Clustering and Visualizing Solution Variation In Massive Programming Classes

by

Elena L. Glassman

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2016

Author . . . . . .
Department of Electrical Engineering
and Computer Science
August 7, 2016

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

Accepted by . . . . . .
_____
Chairman, Department Committee on Graduate Students

Clustering and Visualizing Solution Variation In Massive Programming Classes

by

Elena L. Glassman

Submitted to the Department of Electrical Engineering
and Computer Science
on August 7, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In massive programming-based engineering courses, a single exercise may yield thousands of student solutions. Some solutions are superficially different, while others differ in a fundamental way. Understanding large-scale variation in solutions is a hard but important problem. For teachers, this variation can be a source of pedagogically valuable examples and expose corner cases not yet covered by autograding. For students, the variation in a large class means that other students may have struggled along a similar solution path, hit the same bugs, and can offer hints based on that earned expertise.

This thesis describes three systems that explore the value of solution variation in large-scale programming and simulated digital circuit classes. All three systems have been evaluated using data or live deployments in on-campus or edX courses with thousands of students. (1) OverCode visualizes thousands of programming solutions using static and dynamic analysis to cluster similar solutions. It lets teachers quickly develop a high-level view of student understanding and misconceptions and provide feedback that is relevant to many student solutions. (2) Foobaz clusters variables in student programs by their names and behavior so that teachers can give feedback on variable naming. Rather than requiring the teacher to comment on thousands of students individually, Foobaz generates personalized quizzes that help students evaluate their own names by comparing them with good and bad names from other students. (3) ClassOverflow collects and organizes solution hints indexed by the autograder test that failed or a performance characteristic like size or speed. It helps students reflect on their debugging or optimization process, generates hints that can help other students with the same problem, and could potentially bootstrap an intelligent tutor tailored to the problem.

These systems demonstrate how clustering and visualizing student solutions can help teachers directly respond to trends and outliers within student solutions, as well as help students help each other. The feedback generated by both teachers and students can be re-used by future students who attempt the same programming problem.

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

Preface

I am not a professional software developer, but learning how to write programs in middle school was one of the most empowering skills my father could ever have taught me. I did not need access to chemicals or heavy machinery–just a computer and, occasionally, an internet connection. I acquired datasets and wrote programs to extract interesting patterns in them. It was–and still is–a creative outlet.

More recently, the value of learning how to code, or at least how to "think computationally" has gained national attention. Last September, New York City Mayor Bill de Blasio announced that all public schools in NYC will be required to offer computer science to all students by 2025. In January of this year, the White House released its Computer Science For All initiative, "offering every student the hands-on computer science and math classes that make them job-ready on day one" (President Obama, 2016 State of the Union Address).

The economic value and employment prospects associated with knowing how to program, the subsiding stigma of being "a computer geek," and the popularity of movies about (now rich) programmers like The Social Network, has driven up enrollment in computer science classes to unprecedented levels. Hundreds or thousands of students enroll in programming classes at schools like MIT, Stanford, Berkeley and the University of Washington.

One-on-one tutoring is considered a gold standard in education, and programming education is likely no exception. However, most students are not going to receive that kind of personalized instruction. We, as a computer science community, may not be able to offer one-on-one tutoring at a massive scale, but can we create systems that enhance the teacher and student experiences in massive classrooms in ways that would never have been possible in one-on-one tutoring? This thesis is one particular approach to answering that question.

Acknowledgments

Thank you, thank you, thank you to my advisor, Rob Miller, who took a chance on me, and my other co-authors on this thesis work, Rishabh Singh, Jeremy Scott, Philip Guo, Lyla Fischer, Aaron Lin, and Carrie Cai. The past and present members of our User Interface Design group were instrumental in helping me feel at home in a new area and get up to speed while having fun. I'm also grateful to my past internship mentors at Google and MSR, specifically Dan Russell, Merrie Ringel Morris, and Andres Monroy-Hernandez, who gave me the chance to try out my chops in new environments. I also want to thank my friends outside MIT, especially the greater Boston community of wrestling coaches, who helped me learn important lessons outside the classroom. And, last but not least, my partner Victor, my parents, and my brother, who have each supported me in their own way, from hugs, to proof-reading papers, to being technical sounding-boards, and finally, to demonstrating, as my brother has, how to switch fields, pick up a whole new set of skills, and look good doing it.

add accents to Andres' name

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Given growing demand, programming class sizes at major universities like MIT, Stanford, Berkeley, and University of Washington are reaching hundreds or thousands of students. However, one-on-one tutoring is considered a gold standard for the educational outcomes it regularly produces [10]. The rapid, personalized feedback and attention that is possible within a one-on-one tutoring relationship or a small class becomes prohibitively difficult or expensive at larger scales. Systems and techniques that scale the benefits of low student-to-teacher ratios to arbitrarily many students is an on-going challenge in modern education.

Computers are a natural ally in this challenge. Even though computers do not yet have a human's ability to design curriculum, compose an entirely new question, or handle a novel answer, their attention, computation, and memory scales much better than humanity's. This thesis work supports a vision of teaching computer science education where humans–teachers and students–are front and center, discussing approaches, design choices, and trade-offs, augmented by computation behind the scenes.

The challenge of scaling up computer science education is usually framed as simply coping with the problems of large class sizes. This thesis, however, explores how the volume of students and their solutions can be exploited to enable new types of rapid personalized feedback, as well as recover aspects of a traditional small class.

Learners individually solving programming or digital hardware design problems can collectively generate a wide variety of possible bugs and solutions. A single programming or digital hardware design exercise may yield thousands of student solutions that vary in

many ways, some superficial and some fundamental. For teachers, this variation can, for example, be a source of pedagogically valuable examples and corner cases that slipped past an automatic grader. For students, the variation in a large class means that other students may have struggled along a similar solution path, hit the same bugs, and can offer hints based on that earned expertise.

Understanding large-scale variation in student solutions requires identifying appropriate features of solutions, developing ways to automatically extract those features from each solution, designing user interfaces to communicate the results to teachers or students, and, when appropriate, collecting additional information from students through crowdsourcing techniques, i.e., learnersourcing [61]. This thesis includes the development of three systems that take advantage of the solution variation in large classes: OverCode for viewing thousands of Python solutions, Foobaz for delivering personalized feedback on variable names, self-reflection and comparison workflows for learnersourcing curcuit debugging and optimization hints. These systems and workflows, as well as an extension of OverCode adapted to grading, have been evaluated using data or live deployments in on-campus or edX courses with hundreds or thousands of students.

1.1 The Status Quo

Several years ago, I was hired as a teaching assistant in Computation Structures (6.004). This class is famous for its culminating semester-long project; every student must make an entire processor out of simulated logic gates. Much of my time was spent helping approximately 200 enrolled students debug their simulated circuits. This was fun, but it had several frustrating elements:

1. The simulated circuits were created by students using a language developed by the lecturing professor. It was difficult to read or skim.

2. Since student solutions were only evaluated based on input-output behavior, they could create many different innovative (or inefficient) designs. Established optimizations, like ripple carry adders, were discussed in class, but the diversity of good and

bad designs found within our own crowd of students' solutions were never systematically examined. One student had the courage to publish his own innovative design on the course forum, and we teaching assistants spent much of the rest of the week helping everyone else understand and implement it. How many other students came up with an innovative idea but were afraid to share it, or did not even realize that it was new and different? Some students had no idea whether their solutions were "normal" or unusual.

3. Students failed many of the same teacher-provided tests for the same reasons. These (test failure, problem) patterns would evolve in my mind as I helped more and more students. I was sometimes reduced to an inefficient middleman of debugging advice serving a long queue of customers: "Oh, you failed test 286? Check your Z logic!" The class forum did not seem to alleviate this problem, possibly because students did not encode their test failure problems in a consistent manner for easy searchability.

The professor, who had created the course decades ago, had created multiple implementations of every major circuit design assignment over the years. He was the real resident expert. I shadowed him helping students like medical student might shadow a senior physician. And yet, in order to have the perspective he had without also spending decades teaching the same class, I wanted to augment my cognition with technology.

1.1.1 A Tale of Two Turing Machines

I was wrapping up my late-night shift helping students in the computer lab, when a student asked for help with the Turing machine lab. His Turing machine could not detect whether a string of open and closed parentheses was properly balanced, i.e., had an appropriate closing parenthesis for every open parenthesis. He was running out of time. The semester was almost over, and if he was not successful within the next day or two, he would fail our course.

On that late night, I had a hard time helping this student turn his Turing machine into one that would pass all the automatic grader's test tapes. I could not tell whether his approach could work with a little debugging or was fatally flawed. I did not want to de-

moralize him by telling him to abandon his efforts so far and start over unless it was truly necessary. And what if he was on the path to a novel solution no one else had created before? While not a necessary condition for the correctness of his intended approach, finding a similar working solution in the pool of previous student submissions would serve as an existence proof and evidence that he should persevere.

I had copies of hundreds of other students' correct Turing machines, but it was difficult to see by inspection whether any of these solutions were successful implementations of his intended approach. This was due to superficial differences between solutions:

1. Students design their own symbol library and state names for the finite state machine portion of their Turing machine.

2. Using their custom symbol and state names, students can list behavioral specifications in any arbitrary order.

3. After translating these textual statements into a diagram, the diagrams can look very different from one correct solution to the next.

It is difficult to immediately see any differences in behavior or strategy based on these representations.

Eventually, through trial and error, he got his Turing machine to work on all the test tapes, and passed the course. But I was deeply troubled by the experience. I felt like the information was there, in our staff server, but my brain could not see through the superficial textual differences nor mentally execute all the student solutions I had access to in order to have the perspective that would help me help students.

After the semester ended, I looked into this further. I ran all the correct two-state Turing machines collected during the semester on the same test tape containing a string of open and closed parentheses. The movement of the tape-reading head across this input was logged in coordinates relative to the common starting point at the left end of the test tape (see Figure 1-1a). I had watched so many students' Turning machines execute before that I knew there were be some patterns, I just did not know exactly what or how many there would be.

Most machines exhibited one of two common movement patterns. Roughly 50% of students used Strategy A: pairing inner sets of open and closed parentheses (Figure 1-1b). Another 40% used Strategy B: pairing the first open with the first closed parenthesis, the second open with the second closed parenthesis, etc. (Figure 1-1c). The bold trajectories in these figures represent particularly clean representative examples.

The remaining approximately 10% solutions (not shown) include less common strategies, including at least two that are actually incorrect, i.e., cannot handle arbitrarily deep nesting of parentheses, but are marked as correct because they pass all the test cases in the teacher-designed suite. Many teaching staff members:

1. were aware of the Turing machine solution they each had found on their own time.

2. not aware that there were two mutually exclusive common solutions. At least one staff member admitted steering students away from solutions they did not recognize, but in retrospect may have indeed been valid solutions.

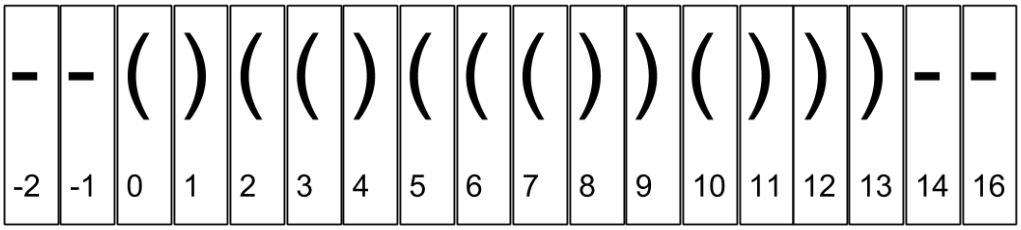
3. not aware that the test suite was insufficient to determine correctness.

I was able to brief fellow staff members on the space of solutions, both good and bad, in preparation for subsequent semesters and suggested additions to the test tape suite to catch more submissions that were previously erroneously marked as correct.

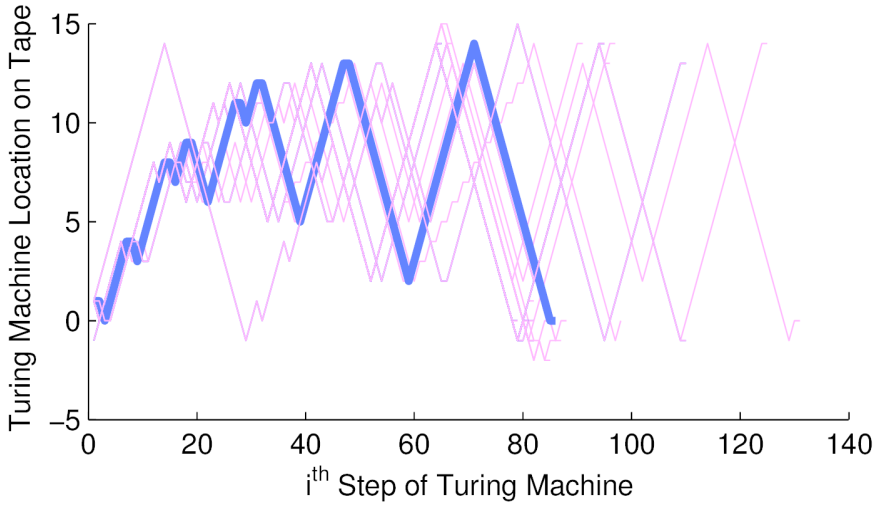
1.2 OverCode

Understanding solution variation is important for providing appropriate feedback to students at scale. In one-on-one scenarios, understanding the space of potential solutions can help teachers counsel students struggling to implement their own solutions. 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.

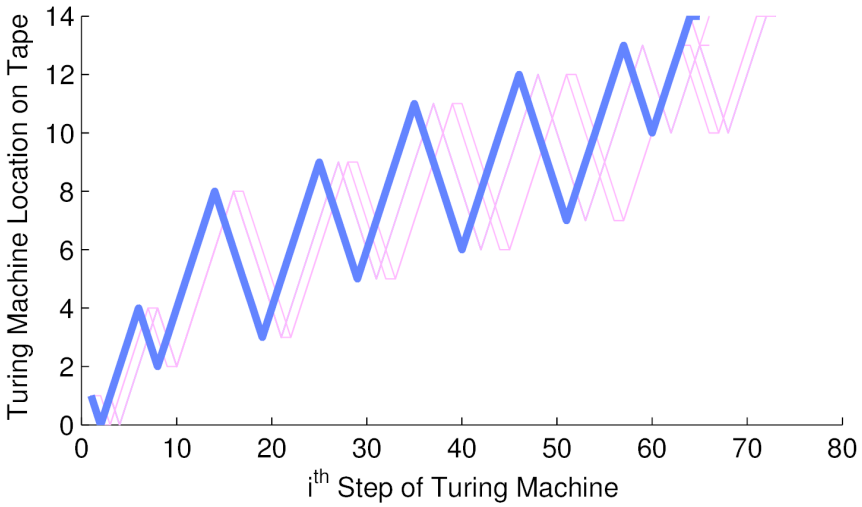
I acquired a dataset of thousands of correct student-written Python solutions to the same programming exercise, freshly collected from MIT's introductory Python programming course on edX. Analogous to tracking Turing machine movement, my colleague Rishabh



(a) Tape on which all 148 two-state Turing machines were tested, and the numbering system by which locations along the test tape are identified.



(b) Strategy A Turing machines: those which paired inner sets of open and closed parentheses, as is standard in mathematical notation. (73 out of 148 Turing machines)



(c) Strategy B Turing machines: those which paired the first open with the first closed parenthesis, the second open with the second closed parenthesis, etc. (58 out of 148 Turing machines)

Figure 1-1: The two most common strategies for a two-state Turing machine to determine if a string of parentheses is balanced. Figures 1-1b and 1-1c show tape head position over time on the tape illustrated in Fig. 1-1a. The bold trajectories represent particularly clean examples.

Singh and I realized that we could identify meaningful patterns in Python solutions by tracking the values of variables during execution on test cases. With the help of Philip Guo, who contributed variable value-logging code from his Online Python Tutor [46] and Jeremy Scott, who built the user interface, we created and tested OverCode, a system for visualizing and exploring thousands of programming solutions.

OverCode uses both static and dynamic analysis to cluster similar correct solutions, and lets teachers further filter and cluster solutions based on different criteria. OverCode rewrites solutions to make them understandable as a collection. To do this, OverCode runs all the solutions to the same programming problem on the same test case(s) and extracts the sequence of values every variable in every solution takes on during execution. Variables that take on the same sequence of values across multiple correct solutions are assumed to be the same *common variable*. The most *common name* given to each *common variable* across all solutions is then used to rename all instances of that common variable in all solutions. At the same time, students' individual comments are removed and formatting is standardized. The resulting set of solutions are rendered in such a way that variable names are semantically meaningful and shared across solutions, as a common vocabulary. All solutions whose *set* of lines are the same after this normalization process are clustered together, with a single normalized solution as the cluster's representative.

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 student understanding and misconceptions, and to provide feedback that is relevant to more student solutions.

OverCode addressed the first two of my three frustrations as a teaching assistant:

1. It canonicalizes students' code so that I can skim their answers as a group or as individuals without first adjusting to each of their variable naming, statement ordering, and style choices.

2. The diversity and distribution of good and bad designs generated by our students is made explicit and explorable. For example, OverCode shows common solutions, which are often reasonable since many students independently arrived at them, and

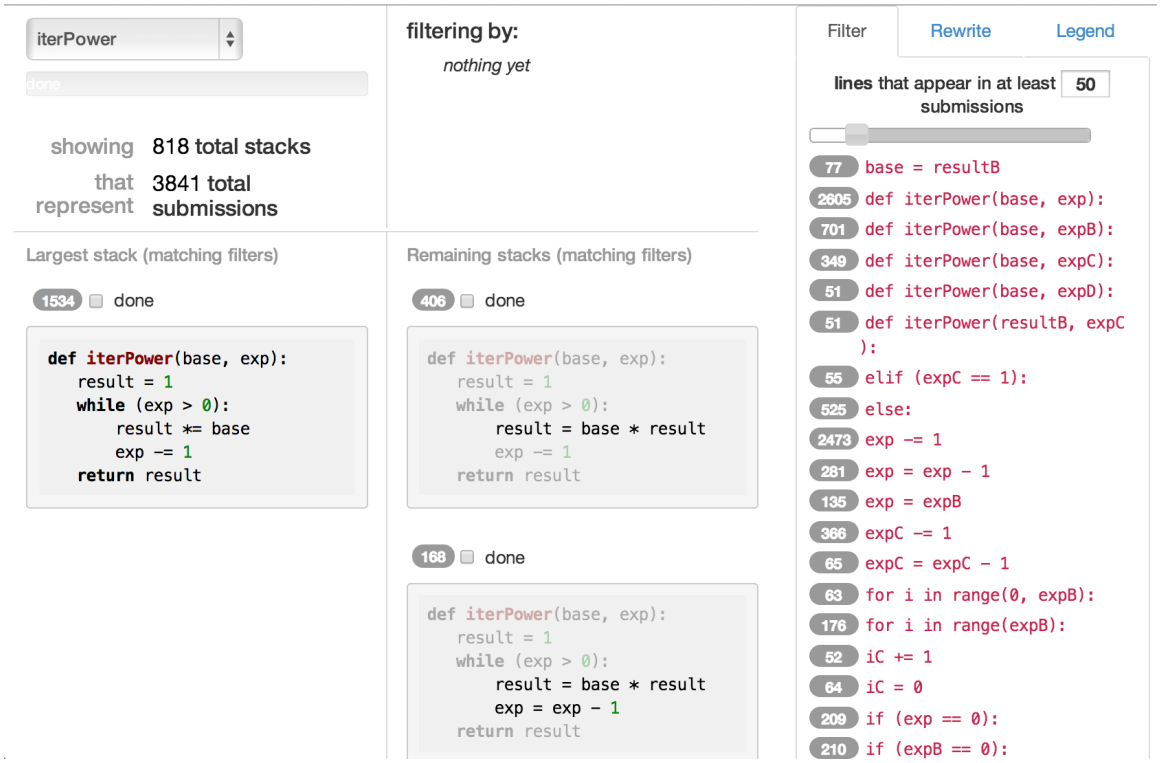


Figure 1-2: The OverCode user interface. The top left panel shows the number of clusters, called *stacks*, and the total number of solutions visualized. The next panel down in the first column shows the largest stack, while the second column shows the remaining stacks. The third column shows the lines of code occurring in the normalized solutions of the stacks together with their frequencies.

> as the user scrolls to less common solutions, solutions can become more convoluted, include unnecessary code, use reasonable alternative Python language keywords and constructs, or solve the problem using teacher-forbidden strategies, e.g., recursively rather than interatively.

1.3 Foobaz

I approached Ana Bell, the lecturer who served both MIT's residential and online introductory Python programming courses. She found the OverCode interface interesting and hoped to use it, then asked if OverCode could also show her more about students' variable name choices. In a second meeting, her co-teacher, Professor John Guttag, explained that some introductory students mix themselves up by creating names that suggest something

other than what the variable contains, e.g., swapping names for the index into an array with the name for the value in the array at that index. It could be an innocent mistake that lengthens debugging time or indicative of a flawed mental model, and it can throw a wrench into their program comprehension process.

Some people undervalue variable names, but they are the most basic form of documentation that helps humans read programs. Donald Knuth compares a good programmer to an essayist who, "with thesaurus in hand, chooses the names of variables carefully and explains what each variable means" [63]. Without modifying execution, names can express to the human reader the type and purpose of an object, as well as suggest what the kinds of operators used to manipulate it [53]. Various naming conventions, like Hungarian notation, have evolved to help developers use their freedom wisely. The Google C++ Style Guide authors assert that their most important consistency rules govern naming, which are arbitrary but consistent in order to increase human readability.

Programmers can develop their own heuristics for good variable names through the experiential learning process of building, debugging, and sharing increasingly large programs with others and their future selves. During informal interviews with programmers in my social network, one professor explained an elaborate set of guidelines that she personally developed and teaches to her students, e.g., all method names must be verbs [29].

In practice, very little time is spent on variable naming in introductory classes. One student-taught month-long introductory Python course offered at MIT spends one lecture on it, and does not systematically assess students' naming choices afterwards. With new tools, giving feedback on variable names could become time-efficient and scalable enough to become common place.

Building on top of OverCode and the support of my co-authors Jeremy Scott and Lyla Fischer, I designed and implemented Foobaz, a system that enables time-efficient, scalable, personalized variable name feedback within the context of existing assigned programming exercises. OverCode systematically hid the variation in variable names. Foobaz reveals that variation, in context.

Like OverCode, Foobaz distinguishes variables by their behavior, recognizing *common variables*. For each common variable, Foobaz allows teachers to comment not only on poor

names, but also on names that mislead the reader about the variable's role. As shown in Figure 1-3a, the teacher's view is a variable name browser where names are shown along with all the context necessary to judge their quality. Teachers only need to annotate a small number of variable names with qualitative judgements in order to generate personalized formative assessments about variable naming for the majority of their students.

In our first user study, the Foobaz system helped each of 10 teachers give personalized variable name feedback on thousands of student solutions from MIT's edX introductory Python programming course. In the second study, students composed solutions to the same programming assignments and immediately received personalized quizzes composed by the teachers in the previous user study, like the one shown in Figure 1-3b.

Traditional feedback methods, such as hand-grading student code for substance and style, are labor intensive and do not scale. Foobaz addresses feedback at scale for a particular and important aspect of code quality. Foobaz also addressed the second of my three frustrations as a teaching assistant: making the diversity and distribution of good and bad choices explicit.

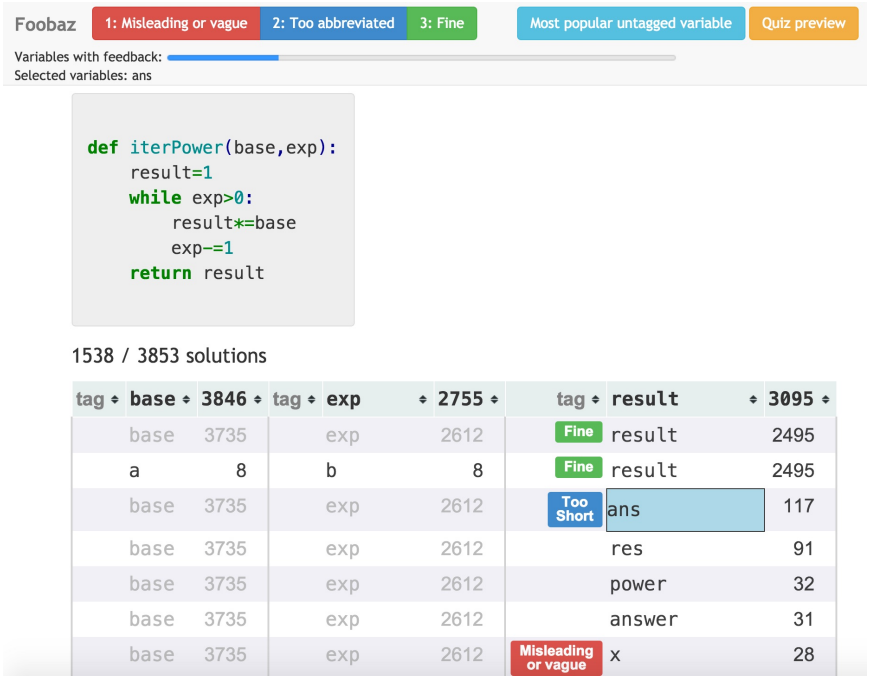
place quiz
reenshot with
stem screen-
ot

1.4 Self-Reflection and Comparison Workflows

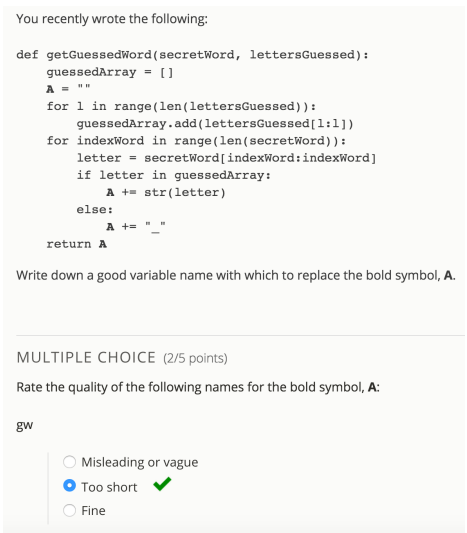
The final section of this thesis is a response to another personal experience in the lab as a teaching assistant, paraphrased here:

> Student A: "I cannot figure out why I'm having this bug."
>
> Teacher (me): "Oh, I have not seen this behavior before. Strange..."
>
> *We sit together, brainstorming, trying things, debugging, stuck...*
>
> Student B walks in, overhears us: "Oh, I just had that buggy behavior. Have you considered...?"

They had separately created the same bug, one after the other. Student B helped Student A debug it. By random chance or because I am not a novice, I had not generated that problem in my own code before. I could model good debugging skills, but Student B could be just



(a) The Foobaz teacher interface. The teacher is presented with a scrollable list of normalized solutions, each followed by a table of student-chosen variable names. Some names shown here have been labeled by the teacher as “misleading or vague,” “too short,” or “fine.”



(b) A personalized quiz as seen by the student, delivered by an edX-based system maintained by a large university. Students are shown their own code, with a variable name replaced by an arbitrary symbol, followed by variable names for the student to consider and label using the same labels that were available to the teacher. After the student has submitted their own judgments, the teacherâĂŹs labels are revealed, along with their explanatory comments.

Figure 1-3: Snapshots of the Foobaz teacher interface and student variable name quiz

as, if not more, helpful to Student A. Student B had earned expertise through experience that I had not. As demonstrated in the user studies that follow, students can and do often restrain themselves from "giving away the answer" when asked to generate a hint after struggling to correct a bug or finish an implementation.

Shortly after my experience of coincidental peer teaching, I stopped thinking of myself as a "resident expert" and started thinking about how to help students more directly share their earned expertise with other students, in a way that would be beneficial for both the giver and the receiver. After helping each student fix a bug and observe a test case go from failed to passed, I asked the student to compose a hint for other students like them who are still failing the same test. I then asked them to enter that hint, indexed by the id of the test case it might resolve, in a web application I wrote, where it would be accessible to all students in the course. Other teaching assistants started pointing students to the web application before helping them, to decrease repeating themselves. I also released it to edX students taking the same course online.

When I saw two students with complementary optimizations of an adder circuit, I made them teach each other their respective optimizations before I gave them their required lab check off. Then I pushed on toward automating this peer-to-peer help on optimalizing circuits. I used the number of transistors in the circuit (circuit size) as an approximate metric of optimality, examined the space of solutions as a function of circuit size, and picked out several key points on that landscape.

During a later semester, when each student enrolled in Computation Structures submitted a correct adder circuit, they were automatically shown a slightly more optimal student solution and a slightly less optimal student solution. They then had to write hints about (1) how to turn a solution like theirs into the more optimal solution, forcing them to engage with and hopefully learn from a more optimal solution and (2) how to turn a solution like the less optimal solution into a more optimal solution like theirs, using their earned implementation expertise in the process.

My co-authors and I hand-analyzed hundreds of optimization hints collected through these targeted prompts and categorized them into the types of hints an expert might write when going through the expensive process of building an intelligent tutoring system for

this domain. In a small lab study, we found that students enrolled in later semesters of the same course could improve their own solutions based on these hints written for solutions like theirs by other students who had the experience to write them.

In general, students in engineering courses can create many different solutions to the same problem, and the space of possible bugs accumulated over all paths to each solution can be very large. Teachers can model good debugging strategies and give feedback on a solutions they have not themselves already created, but students, through their own experience struggling with a particular problem, can become experts on the particular optimizations they implement or bugs they resolve.

In summary, I designed two workflows to harvest and organize students' collective knowledge and advice for helping fellow novices through design problems in engineering (see Figure 5-1). Unlike general learnersourcing, this targeted form of learnersourcing allows or prompts specific students to help current or future students on challenges they have already conquered. In the process of participating in the workflows, students engage in self-reflection, self-explanation, and comparison with alternatives, hopefully within their zone of proximal development. The literature described in the chapter on Related Work suggests that these activities promote learning. Both workflows were evaluated in an undergraduate digital circuit design class with hundreds of students. Within these targeted learnersourcing workflows, students can create helpful hints for their peers that augment or even replace teachers' personalized assistance, especially when that assistance is not available.

By taking better advantage of the knowledge distributed across many students, these workflows addressed the third and final of my three frustrations as a teaching assistant. Using these workflows, I hope future teaching assistants feel less like inefficient middlemen and more like the efficient debugging and optimization consultants they can be, augmented by explicit knowledge of the space of common and uncommon bugs and optimizations.

1.5 Additional Clustering and Visualization

To further explore the power of clustering and visualizing solutions, several extensions of the OverCode methods were explored and one variant, GroverCode for grading, has

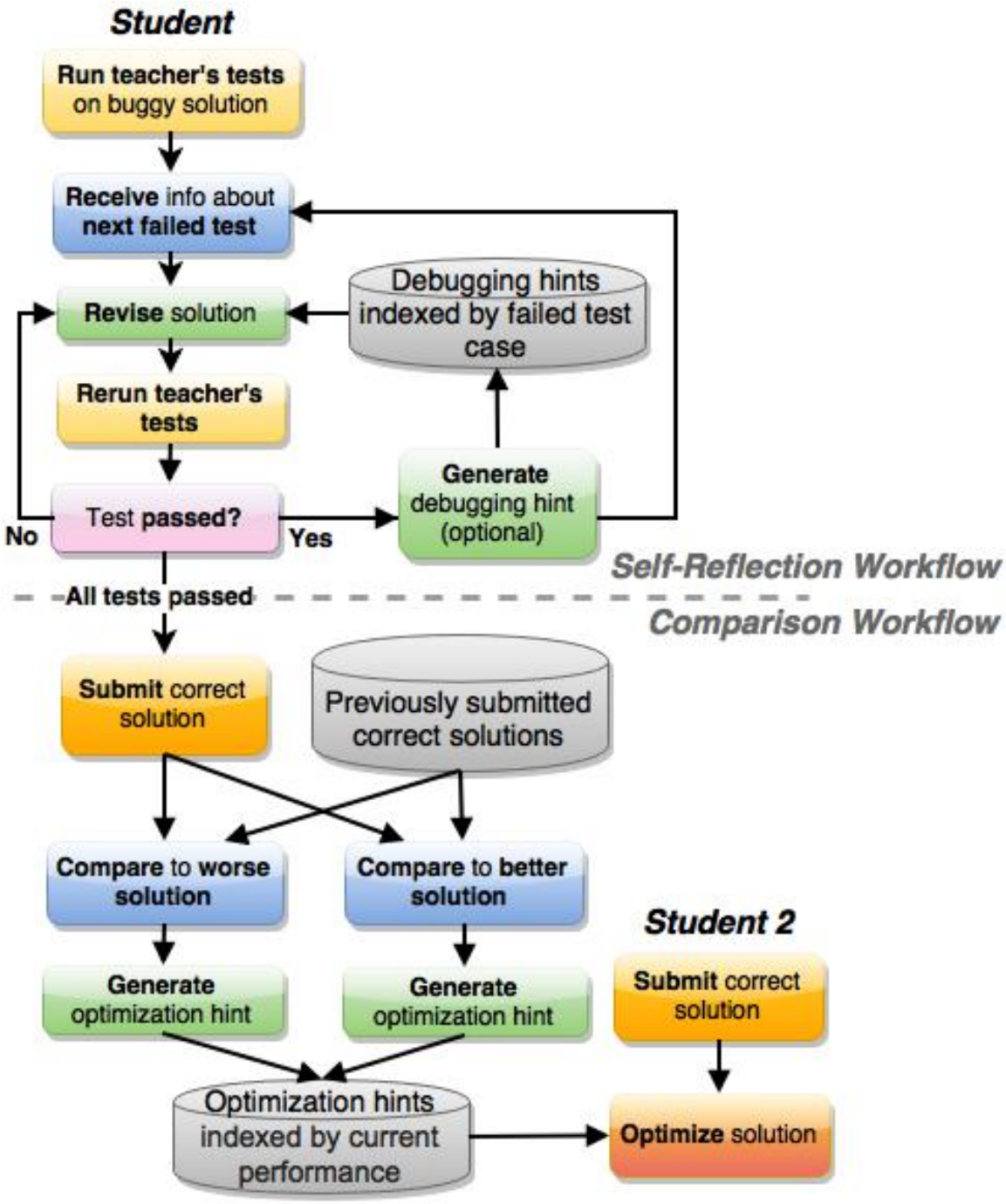


Figure 1-4: In the *self-reflection* workflow, students generate hints by reflecting on an obstacle they themselves have recently overcome. In the *comparison* workflow, students compare their own solutions to those of other students, generating a hint as a byproduct of explaining how one might get from one solution to the other.

been fully implemented. Adapting to the needs of a team of graders grading a smaller number of more complex solutions, many of which were incorrect, required significant changes to the pipeline and interface. The resulting system was deployed to the staff for use during two grading sessions. The results of this field deployment are also promising; the staff welcomes its continued use in future semesters. More preliminary explorations of statistical methods, such as Bayesian inference, for clustering solutions are explored (see Chapter 6.

1.6 Thesis Statement and Contributions

The systems described in this thesis show various mechanisms for leveraging learners in massive programming-based engineering classes. Learners produce many variations of solutions to a problem, running into common and uncommon bugs along the way. Learners can be pure producers whose solutions are analyzed and displayed to teachers to augment their cognition and help them generate feedback. Alternatively, learners can be prompted to generate analysis of their own and others' solutions, for the benefit of both themselves and current and future students.

My thesis statement is:

> Clustering and visualizing solution variation collected from programming courses can help teachers gain insights into student design choices, detect autograder failures, award partial credit, use targeted learnersourcing to collect hints for other students, and give personalized style feedback at scale.

The main contributions of this thesis are:

- A novel visualization that shows similarity and variation among thousands of Python solutions, with normalized code shown for each variant.
- An algorithm that uses the behavior of variables to help cluster Python solutions and generate the normalized 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' Python solutions.

- A grading interface that shows similarity and variation among Python solutions, with faceted browsing so that users can filter solutions by error signature, i.e., the test cases they pass and fail.
- Two field deployments of the grading interface within an introductory Python programming staff's exam grading sessions.
- A technique for displaying clusters of Python solutions with only an aspect, i.e., variable names and roles, of each cluster exposed, revealing the details that are relevant to the task.
- A workflow for generating personalized active learning exercises, emulating how a teacher might socratically discuss good and bad choices with a student while they review the student's solution together.
- An implementation of the above technique and method for variable naming.
- Two lab studies which evaluate both the teachers' and students' experience of the workflow applied to variable names.
- A self-reflection learnersourcing workflow in which students generate hints for each other by reflecting on an obstacle they themselves have recently overcome while debugging their solution to a Python or digital design exercise.
- A comparison learnersourcing workflow in which students generate design hints for each other by comparing their own solutions to alternative designs submitted by other students.
- A deployment of the self-reflection workflow in a 200-student class and a lab study of the comparison workflow with 9 participants.

1.7 Thesis Overview

Chapter 2 summarizes prior and contemporary relevant research on systems that support programming education. It also briefly explains theories from the learning sciences and

psychology literature that influenced or support the pedagogical value of the design choices made within this thesis.

The four chapters that follow describe, in detail, the four systems developed, as well as their evaluation on archived data or in the field.

- OverCode (Chapter 3) visualizes thousands of programming solutions using static and dynamic analysis to cluster similar solutions. It lets teachers quickly develop a high-level view of student understanding and misconceptions and provide feedback that is relevant to many student solutions.

- Foobaz (Chapter 4) clusters variables in student programs by their names and behavior so that teachers can give feedback on variable naming. Rather than requiring the teacher to comment on thousands of students individually, Foobaz generates personalized quizzes that help students evaluate their own names by comparing them with good and bad names from other students.

- Chapter 5 describes two workflows that collect and organize solution hints indexed by (1) the autograder test that failed or (2) a performance characteristic like size or speed. It helps students reflect on their debugging or optimization process, generates hints that can help other students with the same problem, and could potentially bootstrap an intelligent tutor tailored to the problem.

- OverCode Extensions (Chapter 6) describes (1) GroverCode, an extension to OverCode optimized for processing and displaying incorrect as well as correct student submissions and (2) clustering and mixture modeling algorithms applied to the OverCode pipeline's output for additional insight into patterns within student solutions.

Chapter 7 discusses the design lessons that apply to the entire collection of systems in this thesis. Chapter 8 outlines avenues of future work this thesis work, in combination with the complementary work of others in this space, paves the way for.

Chapter 2

Related Work

Systems that help students in massive programming courses may build on work from any or all the following related fields: program analysis, program synthesis, crowd workflows, user-interface design, machine learning, intelligent tutoring systems, natural language processing, data mining, and learning science. First, I present prior work and theories of how people learn that later inspired key design decisions. I then clarify how this thesis work is novel while describing related work that achieves similar goals or uses similar methods.

2.1 Scalable teaching principles

There are many factors inside and outside the classroom that have significant effects on learning. I will focus on a few techniques and theories from the learning sciences that a human may be able to execute better with a computer than without. Therefore, peer groups, home environment, learning communities, and identity formation [17, 124] are beyond our consideration.

2.1.1 Tutoring

Tutoring has been held up as a gold standard in education since 1984 when Bloom published a collection of his lab's work demonstrating tutoring's efficacy relative to other experimental and conventional methods at the time [10]. For example, his lab found that,

after 11 sessions of instruction in probability or cartography, elementary and middle school students who received tutoring in groups of one to three were, on average, two standard deviations better than their counterparts in a conventional 30-person classroom. Given the expense of scaling up one-on-one tutoring, Bloom challenged the academic community to find a method of group instruction that was just as good, or better. That challenge still stands as a benchmark that modern systems and techniques can compare against. While I do not measure the relative learning gains associated with the systems presented in this thesis, I do relate key design decisions to the tutoring best practices that are able to be scaled up or automated, such as prompting for self-explanations.

Reflection and Self-Explanation

Effective tutors often have characteristics described by Lepper and Wolverton's INSPIRE model: superior domain and pedagogical content knowledge, nurturing relationships with students, progressive content delivery, Socratic styles that prompt students to explain and generalize, and feedback on solutions, not students [129]. Turns et al. [118] argue that the absence of reflection in traditional engineering education is a significant shortcoming. This thesis contributes systems designed to address that shortcoming.

Self-explanations are generated by the student for themselves; they are a form of reflection, which is a critical method for triggering the transformation from conflict and doubt into clarity and coherence [26]. Students self-explanations foster the integration of new knowledge, while some tutors' explanations may be insufficient or flawed, due to the curse of knowledge [9]. Effective tutors encourage self-explanation by prompting students with questions like *Why?* and *How?* [20]. Students of tutors who fostered self-explanations had learning gains similar to those whose tutors provided their own explanations and feedback[21].

2.1.2 Deliberate Practice

Deliberate practice is generally accepted to be goal directed, effortful, not enjoyable, repetitive, accompanied by rapid feedback, and only sustained as long as the learner can be fully

concentrated on the task, i.e., no more than a few hours [43]. For example, rather than just playing pickup basketball games in the neighborhood, an aspiring professional player might design specific drills to work on his/her weaknesses. Teachers help facilitate deliberate practice, because they can design appropriate exercises and provide feedback until the student can differentiate between good and bad performance and provide that feedback to themselves.

Recent work incorporating deliberate practice in large classrooms has demonstrated great benefits. A recent study of undergraduate physics classrooms found that, with deliberate practice as a base of the instructional design, improvements can approach and exceed Bloom's 2-sigma threshold [25]. Another recent study confirms the value of rapid feedback in foundation engineering classrooms [19]. This thesis contributes systems designed to provide feedback on some aspects of programming tasks where it was not feasible to do so before, such as Foobaz, which stimulates reflection and provides feedback on one of the most basic forms of program readability: variable names.

2.1.3 Zone of Proximal Development and Scaffolding

The concept of the zone of proximal development (ZPD) was first introduced in the mid 1920's by the Soviet pyschologist Lev Vygotsky. It refers to the gap between what a learner can do without help and what a learner cannot yet do, no matter how much help they are given. It is implied that an object of learning strictly outside the ZPD is either too easy or too hard, and little or no learning will occur.

Wood et al. [128] introduced a complementary process called scaffolding. Scaffolding enables a learner to solve problems or achieve goals that would ordinarily be beyond their grasp because the teacher controls the aspects of the task that are initially outside the learner's abilities. Recent work suggests that the maximum learning gains come from giving students the hardest possible tasks they are able, with the assistance of scaffolding, to complete [125]. Formative feedback [108] can be helpful as part of the scaffolding. It should non-evaluative, supportive, timely, and specific. It usually arrives as a response to a student's action, e.g., a hint, an explanation, or a verification based on the student answer.

Based on this research, one of the components in this thesis, the comparison workflow, identifies where a student solution is on the spectrum of optimality and prompts the student to reflect on the *next most optimal* solution.

2.1.4 The Role of Strategic Variation in Examples

Studies of tutors and their students help us identify characteristics and styles of interaction that help explain the effectiveness of tutoring. Some of these can be successfully deployed in large classrooms. However, the way we frame content can also have large effects on how students understand, generalize, and transfer their learning to new contexts.

Concrete examples of an object of learning–like how to apply an appropriate statistical test in a statistics word problem–vary in ways that are superficial, e.g., irrelevant, and fundamental, e.g., relevant. In the language of educational psychologists, these are often called surface and structural features [94]. A simple compare and contrast exercise when solving equations [97] or examining case studies in negotiation [75] can bring this variation to the fore, and yield learning benefits.

Learning in the presence of variation in these features helps learners generalize and transfer their knowledge to new situations, such as better transfer of geometric problem solving skills [91, 120]. Several educational models, e.g., Variation Theory and the 4C/ID Model [119], build on the value of variability by suggesting specific ways for how it should be deployed in the classroom. The three primary components in this thesis, OverCode, Foobaz, and the targeted learnersourcing workflows, are all designed to make the natural variability present in student solutions useful to teachers and students, in accordance with recommendations from the educational theories that follow.

Analogical Learning

Analogies are central to human cognition; they can help learners understand and transfer knowledge and skills to new situations. Analogical learning is at play both when learners have a base of knowledge that they bring with them to a novel target and when they compare two partially understood situations that can illuminate each other, serving as both a

source and recipient of information [68, 75]. However, in order to reap the full benefits of analogical learning, learners must engage deeply. Reading two cases, serially, in a session is not enough; learners will not necessarily make the necessary connections unless there are explicit instructions to compare [18, 75]. Kolodner [64] suggests creating software tools that align examples to facilitate analogical learning. This thesis is, in part, a response this suggestion.

Novices may become confused if asked to compare their solution to a fellow student's solution; this is not necessarily bad for learning outcomes. Piaget theorized that cognitive disequilibrium, experienced as confusion, could trigger learning due to the creation or restructuring of knowledge schema [57]. D'Mello et al. maintain that confusion can be productive, as long as it is both appropriately injected and resolved [28]. Similarly, reflecting on a peer's conceptual development or alternative solution may bring about cognitive conflict that prompts reevaluation of the student's own beliefs and understanding [56]. The comparison workflow component of this thesis is designed to stimulate this kind of productive confusion, comparison, and resolution through self-explanation.

Analogical learning can be very difficult. For example, the structural features may be aligned between a base and the new target situation, but large differences in surface features will hurt the learner's ability to see any connection [67]. This may be explained by how human memory works. For novices, the most reliable form of retrieval is based on surface similarity, not deep analogical similarity; experts can more easily retrieve situations that are structurally similar and therefore more relevant for a new situation at hand [74].

Variation Theory, discussed next, is specifically designed to help students more deeply appreciate structural features, which may help them transfer their learning to new situations instead of feeling lost, confused by superficial differences.

add Towards Providing Feedback to Student in Absence of Formalized Domain Models by Sebastian Gross, Bassam Mokbel, Barbara Hammer, Niels Pinkwart

Variation Theory: Preventing Human Overfitting

This aphorism captures the ideas at the heart of Variation Theory (VT) well: *He cannot, England know, who knows England only.* VT is concerned with the way in which students are taught from concrete examples. It is relatively new and still being investigated for its usefulness in a broad range of disciplines, including mathematics and computer science.

VT asserts that human learning can suffer from overfitting for some of the same reasons that machine learning algorithms do. Overfitting is a term I am borrowing from the machine learning community. If a machine learning algorithm selects the the color of the sky as the key difference between photos of cats and dogs (which is obviously unrelated to distinguishing between photos of cats and dogs), then a possible explanation is that the algorithm was trained on photos with insufficient or biased variation. If all the cat photos it ever saw were taken on a cloudy day, and all the dog photos it ever saw were taken on a sunny day, could you blame this naive program for latching on to this obvious differentiator of housepet species? Humans can make the same inferential mistake when not exposed to a sufficient variety of examples of an object of learning. VT catalogues a hierarchy of patterns of variation designed to immunize the learner to this kind of mistake.

More abstractly, VT is built on the understanding that learning is not possible without being able to discern what the object of learning is [78]. Discernment is not possible without experiencing variation in the object of learning and the world in which its situated [80]. This variation is described in terms of aspects and features. An aspect refers to a dimension of variation, and a feature is a value of that dimension of variation [71]. Some features are irrelevant, while critical features collectively define the object of learning.

For a more concrete discussion of variation, consider the following examples:

1. The phrase *a heavy object* might not make sense to the reader unless they have interacted with objects of various weights.

2. Consider a child who recently learned how to add numbers, but always starts with the larger number: 2+1=3, 4+2=6, etc. Asking the child to add the numbers in the opposite order, smaller number first, and verify that the result is the same introduces the commutative feature of addition.

3. No matter how wildly a cup diverges from a prototypical example of a tea cup, if it does not have the critical feature of being able to hold something, it is not a cup.

Marton et al. [78] identify four patterns of variation: contrast, separation, generalization, and fusion. If a child is learning the concept of *three*, then contrast refers to being

introduced to three apples, as well as a pair of apples, or a dozen. Generalization refers to being introduced to different groups of three: three apples, three dogs, three beaches, and three languages. This clarifies that it is not the apples that give *three* its meaning. Separation refers to a pattern of examples that helps the learner separate a dimension of variation from other dimensions of variation. A child could be introduced to a litter of nearly identical puppies that only differ in coat color, for example. Fusion is the final pattern of variation, where the learner is exposed to examples that vary along all the dimensions of variation at once, since this is most commonly encountered in the real world. These patterns of variation are intended to reveal which aspects of a concept or phenomenon are superficial and irrelevant and which are innate and critical to its definition.

VT is a framework that has guided teaching materials and been used as an analytic framework in a variety of contexts, including lessons on critical reading [90], vocabulary learning [31], the color of light [70], mathematics [83], chemical engineering [14], Laplace transforms [16], supply and demand [79] and computing education [115]. It has been the subject of a government-funded three-year longitudinal study in Hong Kong, with promising results [73].

In this thesis, Overcode and Foobaz are explicitly designed to discover and make human-interpretable the variation naturally present in student solutions. All the systems in this thesis demonstrate ways in which extracted variation, in solutions or errors, can be used in massive classes. In the next section, I describe efforts to extract, answer questions about, or otherwise make usable the natural variation in corpuses of webpages, apps, and code that do not all have the same purpose. In the final section, I describe similar efforts specifically for corpuses of student solutions to the same programming exercise.

2.2 Exploring and Mining Variation in the Wild

One important aspect of both engineering and design is that there are multiple ways to solve the same problem. There are many means to an end, and the optimality of each solution may be context-dependent. Sub-optimality can have high personal, safety, and economic costs. For example, in the software industry, maintenance dominates the cost of producing

software [37]. Poorly designed code may be the culprit because it requires significantly more maintenance.

If solutions can be indexed in useful ways, they can be mined to ask basic questions like,

1. In the face of a common design choice, what do people most commonly pick? Has this changed over time?
2. What are popular design alternatives?
3. What are examples of design fails that should be learned from and never repeated?
4. What are examples of design innovations that are clearly head-and-shoulders above the rest?

The answers to these questions could fuel instruction that complies with Variation Theory's recommendations. With OverCode, Foobaz, and the targeted learnersourcing workflows, I try to answer the same questions for students tackling the same programming challenge.

2.2.1 Webpages

Ritchie et al. [96] describe a user interface for finding helpful, i.e., relevant or inspiring, design examples from a curated database of web pages. This work is intended to support designers who like to refer to or adapt previous designs for their own purposes. Traditional search engines only index the content of web pages; this system indexes web pagesâĂŹ design style by automatically extracting global stylistic and structural features from each page. Instead of manual browsing, users can search and filter a gallery of design-indexed pages. Users can provide an example design in order to find similar and dissimilar designs, as well as high-level style terms like "minimal."

Kumar et al. [66] defined *design mining* as "using knowledge discovery techniques to understand design demographics, automate design curation, and support data-driven design tools." This work goes beyond searching and filtering a gallery of hundreds of curated webpages. Their Webzeitgeist design mining platform allows users to query a repository

of hundreds of thousands of web pages based on the properties of their Document Object Model (DOM) tree and the look of the rendered page. A 1679-dimensional vector of descriptive features are computed for each DOM node in each page.

Webzeitgeist enables users to ask and answer some of those originally highlighted questions, with respect to this large web page repository:

1. What are all the distinct cursors?
2. What are the most popular colors for text?
3. How many DOM tree nodes does a typical page have? How deep is a typical DOM tree?
4. What is the distribution of aspect ratios for images?
5. What are the spatial distributions for common HTML tags?
6. How do web page designers use the HTML canvas element?

To dig into examples of a particular design choice, users can, for example, query for all pages with very wide images. The result is a set of horizontally scrolling pages. Alternatively, users can query for webpages that have a particular layout, like a large header, a navigational element at the top of the page, and a text node containing greater than some threshold of words, in order to see all the examples of pages that fit those layout specifications. Specific combinations of page features can imply high-level designs as well so, with careful query construction, users can query for high-level ideas. For example, querying for pages with a centered HTML input element AND low visual complexity retrieves many examples that look like the front pages of search engines.

2.2.2 Android Apps

Shirazi et al. [104] and Alharbi and Yeh [2] describe automated processes for taking apart and analyzing Android app code as well as empirical analyses of corpuses of Android Apps available on the Google Play app store. Shirazi et al. analyzed the 400 most popular free

Android applications, while Alharbi and Yeh tracked over 24,000 Android apps over a period of 18 months. Alharbi and Yeh caputured each update within their collection window, as well. They decompiled apps into code from which UI design and behavior could be inferred, e.g., XML and click handlers, and tracked changes across versions of the same app. Both papers analysed population-level characteristics of their corpuses, answering questions like:

- What is the distribution of layout design patterns, among the seven standard Android layout containers?
- What are the most common design patterns for navigation, e.g., tab layout and horizontal paging? Have any apps switched from one pattern to another?
- How quickly are newly introduced design patterns adopted?
- What are the most frequent interface elements? And combinations of interface elements? How many applications does that combination cover?

2.2.3 Open-Source Code Repositories

Ideally, code is not just correct, it is simple, readable, and ready for the inevitable need for future changes [35, 93]. How can we help students reach this level of programming composition zen? How can we learn from others' code, even after we become competent, or even an expert, at the art of programming?

For the same reason we look at patterns in design across web pages and mobile apps, we can look at the design choices already made by humans who share their programs. Rather than using web crawlers or app stores, we can process millions of public repositories hosted online. What can we learn about good and bad code design decisions from these collections?

These code analysis techniques that follow are most closely related to those developed for OverCode and Foobaz. OverCode and Foobaz's pipelines are optimized for user interfaces that help users answer design mining questions about their students' compositions and put the code front and center. The advantages and disadvantages of the OverCode and

Foobaz pipelines with respect to the systems designed in this section will be discussion in later chapters after the techniques developed in this thesis are fully explained.

Regularity In Code

Several papers make similar observations, arguments, and emperical validation of the regularity that can be found in code. Hindle et al. [48] were motivated by the assertion that human-produced natural language and human-produced program language may both be "complex and admit a great wealth of expression, but what people write ... is largely regular and predictable." The authors argue that the assertion may be even more true for code than for natural language. Allamanis and Sutton [3] observe that there are syntactic fragments, i.e., idioms, that serve a single semantic purpose and recur frequently across software projects. Fast et al. [36] observe that poorly written code is often syntactically different from well written code, with the caveat that not all syntactically divergent code is bad.

Mining Idioms

Idiomatic code is written in a manner that experienced programmers perceive as normal or natural. Idioms are roughly equivalent to mental chunks, i.e., the memory units characterized by George Miller [84]. I will borrow an example from Allamanis and Sutton [3]

- `for (int i=0;i<n;i++) ...` is a common idiom for looping in Java.
- do-while and recursive looping strategies are not.

An experienced Java programmer will be able to understand the code whether it's idiomatic or not, but it may take longer. They may even be distracted by questions, e.g., *Why did the author make this choice?*

Fast et al. [36] break the definition of idioms into two levels. An example of a high-level idiom is code that initializes a nested hash. An example of a low-level idiom is code that returns the result of an addition operation. Some languages support a variety of different, equally good ways to do the same thing; others encourage a single, idiomatic way to achieve each task.

Idioms can and do recur throughout distinct projects and domains (unlike repeated code nearly verbatim, i.e., clones) and commonly involve syntactic sugar (unlike API patterns). In general, clone detectors look for the largest repeated code fragments and API mining algorithms look for frequently used sequences of API calls. Idiom mining is distinct because idioms have syntactic structure and are often wrapped around or interleaved with context-dependent blocks of code, like the block of statements within an the idiomatic for loop in the previous paragraph.

There are enough idioms for some languages that they have lengthy, highly bookmarked and shared online guides. StackOverflow has many questions asked and answered about the appropriate language or library-specific idioms for particular, common tasks. It is difficult for expert users of each language or library to catalogue all the idioms. It is much more practical to simply look at how programmers are using the language or library and extract idioms from the data.

Hindle et al. [48] used statistical language models from natural language processing to identify idiom-like patterns in Java code. They found that corpus-based n-gram language models captured a high level of project- and domain-specific local regularity in programs. Local regularities are valuable for statistical machine translation of natural language; they may prove useful in analogous tasks for software as well. For example, the authors trained and tested a corpus-based n-gram model token suggestion engine that looks at the previous two tokens already entered into the text buffer and predicts the next one the programmer might type.

Allamanis and Sutton [3] automatically mine idioms from a corpus of idiomatic code using nonparametric Bayesian tree substitution grammars. The mined idioms correspond to important programming concepts, e.g., object creation, exception handling, and resource management, and are, as expected, often library-specific. They found that 67% of the idioms mined from one set of open source projects were also found in code snippets posted on StackOverflow.

Fast et al. [36] computed statistics about the abstract syntax trees (ASTs) of three million lines of popular open source code in the 100 most popular Ruby projects hosted on Github. AST nodes are normalized, and all identical normalized nodes are collapsed into a

single database entry. The unparsed code snippets that correspond to each normalized node are saved. Codex normalizes these snippets by renaming variable identifiers, strings, symbols, and numbers to `var0`, `var1`, `var2`, `str0`, `str1`, etc. Note that this fails when primitives, like specific strings and numbers, are vital to interpreting the purpose of the statement.

The resulting system, Codex, can warn programmers when they chain or compose functions, place a method call in a block, or pass an argument to a function that is infrequently seen in the corpus. It is fast enough to run in the background of an IDE, highlighting problem statements and annotating them with messages like, "We have seen the function split 30,000 times and strip 20,000 times, but we've never seen them chained together." Codex can be queried for nodes by code complexity; type, i.e., function call; frequency of occurrence across files and projects; and containment of particular strings.

Mining Larger Patterns in Code

In the code of working applications, Ammons et al. [4] observed that common behavior is often correct. Based on that observation, they use probabilistic learning from program execution traces to infer the programâĂŹs formal correctness specifications. Inferring formal specifications for programs is valuable because programmers have historically been reluctant to write them. During program execution, the authors summarize frequent patterns as state machines that can be inspected by the programmer. As a result, the authors identified correct protocols and some previously unknown bugs.

Buse and Weimer [13] go beyond idioms to mining API useage. Based on a corpus of Java code, they find examples that reference a target class, symbolically execute it to compute intraprocedural path predicates while recording all subexpression values, identify expressions that correspond to one use of the class, capture the order of method calls in those concrete examples, then use K-mediods to cluster these extracted concrete use examples with a custom formal parameterized distance metric that penalizes for differences in method ordering and type information. Concrete use examples within the same cluster are merged into abstract uses represented as graphs with edge weights that correspond to counts of how many times node X happens before node Y. Finally, they have a synthesis method to express these abstract use graphs in a human-readable form, i.e., representative,

well-formed, and well-typed Java code fragments.

Mining Names

Without modifying execution, names can express to the human reader the type and purpose of an object, as well as suggest the kinds of operators used to manipulate it [52]. Perhaps as a direct result, variable names can exhibit some of the same regularity exhibited by code, in general. Høst and Østvold [49] go so far as to call method names a restricted natural language they dubbed Programmer English.

Høst and Østvold [49] ran an analysis pipeline on a corpus of Java that performs semantic analysis on methods and grammatical analysis on method names; it generates a data-driven phrasebook that Java programmers can use when naming methods. In a second publication [50], they formally defined and then automatically identified method naming bugs in code, i.e., giving a method a name that incorrectly implies what the method takes as an argument or does with an argument.

They did this by identifying prevalent naming patterns, e.g., contains-*, which occur in over half of the applications in the corpus and match at least 100 method instances. They also determined and cataloged the attributes of each method body, such as whether it read or wrote fields, created new objects, or threw exceptions. If almost all the methods whose names match a particular pattern, e.g., contains-*, have an attribute or do not have some other attribute, it automatically determined to be an implementation rule that all names in the corpus should follow. On a large corpus of Java projects, this analysis pipeline found a variety of naming bugs.

Fast et al.'s Codex [36] produced similar results; by keeping track of variable names in variable assignment statements, it can warn programmers when their variable name violates statistically-established naming conventions, such as the (probably confusing) naming of a Hash object "array."

Foobaz can go beyond Fast et al. [36] and Høst and Østvold's [49, 50] work in providing feedback on variable names because all solutions are known to address the same programming exercise.

2.3 Exploring and Mining Variation in Student Solutions

All the work in this thesis was composed during an eruption of interest in teaching massive numbers of people through online exercises and courses. Computer-based, scalable teaching environments, such as Intelligent Tutoring Systems, were not new, and neither was putting coursework up online. For example, MIT OpenCourseware opened its virtual doors to provide free access to static MIT course materials nearly 15 years ago, in 2002. However, it was not until 2011, that the first full university level course was made available to the public, complete with college-level programming exercises. Soon after, several organizations, e.g., edX, Coursera, and Udacity, started up to provide similar experiences for a range of university courses. I was acutely aware of edX because it was first organized from within the same basement lab of Stata in which I was spending my days and nights helping students complete assignments for Computation Structures.

These organizations were soon sitting on top of piles of student activity and responses that could be studied by researchers. It became possible to formulate and answer new questions. The ACM Learning at Scale conference was started to give these a dedicated venue to talk about this new work. Multiple research groups began working on making sense and making use of this data to enhance learning outcomes and experiences. The work in this section is relevant to processing large corpuses of student solutions to the same programming exercise.

The common purpose of the code in the corpus almost certainly contributes to the regularity already found in code from large corpuses of open source projects. However, this regularity from a common purpose may be more than counter-balanced by the fact that the code is generated by novices who are still learning how to program and are not yet well-versed in the common idioms and programming constructs.

Unlike corpuses of open source projects, it is often not difficult to go beyond static analysis and perform dynamic analyses while executing each solution in the corpus. A set of teacher-designed tests of expected behavior may already exist, some available to students while developing their solutions, some hidden from students' view, and some generated on the fly, through fuzz testing [113].

Also unlike corpuses of open source code, the solutions in large corpuses of student code often require feedback, so that the author can learn. Automated feedback on solutions to programming exercises is still an area of active research. For example, assigning grades based solely on the number of teacher-designed tests the code passes may not capture what teachers care to grade on. A single error in a solution can cause a near-perfect solution to fail all test cases. A solution can perform perfectly on test cases but be poorly written or violate instructions, e.g., use the wrong algorithm to achieve the right results. The test cases themselves may be poorly designed. For these reasons, the teaching staff of 6.0001 at MIT review every exam solution from each student by hand.

When teachers have hundreds or thousands of students, it can be challenging to provide feedback to each solution quickly and consistently by hand. Design mining techniques and interfaces can help teachers explore and understand the whole space of solutions as well as distribute feedback to specific subsets of solutions, or subsets within solutions. It could be an important tool to assist in the sometimes difficult, manual task of identifying pedagogically valuable examples for illuminating a concept or principle. Distributing feedback can also be approached as an unsupervised or supervised, possibly interactive, machine learning problem, by leveraging clustering, classification, or mixture modeling methods. Clustering also need not be done by machine learning methods; clone detection methods can identify clones at a variety of levels [102].

2.3.1 Regularizing Code

Before comparing solutions, it is common to preprocess the code to remove token variability that is irrelevant to later stages in the analysis pipeline. Light preprocessing might include normalizing white space and removing comments. Slightly more preprocessing might include systematically renaming variable and method names, i.e., those chosen by the code author, to generic placeholders. OverCode and Foobaz also preprocess solutions by normalizing whitespace and removing comments. However, since variables are central to the both systems, variables are carefully tracked and strategically renamed, rather than replaced by generic placeholders. OverCodeâĂŹs 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.

More aggressive code preprocessing can remove syntactic differences between solutions that are semantically similar by using semantics-preserving transformations. This can include standard compiler optimizations, such as dead code removal, constant folding, copy propagation, and the inlining of helper functions [99]. It can also include transformations, like changes in the order of operands to a commutative function, that make one solution closer to another solution with respect to some definition of edit distance. Applying semantics-preserving transformations, sometimes referred to as canonicalization or standardization, has been used for a variety of applications, including detecting clones [6, 55], automatic "transform-based diagnosis" of bugs in students' programs written in programming tutors [130], and self-improving intelligent tutoring systems [99].

A schema, in the context of programming, is a high-level cognitive construct by which humans understand or generate code to solve problems [111]. For example, two programs that implement bubble sort have the same schema, bubble sort, even though they may have different low-level implementations. While powerful, these semantics-preserving transformations will not change the schema(s) within a solution. This is not true for the probabilistic semantically equivalent AST subtrees algorithmically mined from a corpus of solutions by Nguyen et al. [89]. These are probabilistic equivalences because the different AST subtrees are only verified to be semantically equivalent *with respect to* the problem and the specific test cases provided.

2.3.2 Features and Distance Functions

The challenge is designing features or distance functions that capture what the teacher cares about. Teachers may want to give feedback on correctness, design decisions (a.k.a., code style), or both. Other applications of code similarity functions are clone detection and plagiarism. Solutions can be represented as sets or vectors of hand-crafted computed features, sequences of tokens, or graphs. In the context of software, tokens could be characters or tokens corresponding to the lexical rules of the programming language. Drummond et

al. [32] catalogues additional distance measures that are potentially helpful for clustering interactive programs.

Features

Just as the authors of Webzeitgeist defined sets of features to be computed for each node in a web page's DOM [66], there are many sets of features used to estimate program similarity in the literature. Aggrawal et al. [114], Elenbogen and Seliya [34], Roger [101], Rees [95], Huang et al. [51], Kaleeswaran et al. [54], and Taherkhani et al. [116] each defined a set of features that could be computed automatically for each solution and represented as a numerical feature vector. These feature sets each include a mixture of static and/or dynamic features.

Static features include but are not limited to counts of the solution's various keywords and tokens, e.g., control flow keywords, operators, constants, and external function calls; counts of each type of expression in the solution; measures of code complexity; length of commented code; scores from linting scripts; function name lengths; line lengths; and entire solution lengths. To quantify the goodness of a solution's style, AutoStyle [22] used a pre-existing metric called the ABC score, a weighted count of assignments, branches, and conditional statements in a block of code.

Dynamic features include but are not limited to data collected from running a solution on each test case, e.g., the solution's output and whether or not it is correct with respect to the teacher's specification, the evolution of values assigned to each variable within the solution, and the order in which statements in the solution are executed. For example, Huang et al. [51] create an 'output vector' for each solution: a binary vector representing the solution's correctness on each test case. For approximately one million solutions to 42 programming exercises Stanford's original Machine Learning MOOC, the authors found that a teacher could cover 90% of solutions or more in almost all problems by annotating the top 50 output vectors. However, this could be difficult; the authors acknowledge that many different mistakes can produce to the same output vector.

Role of Variables Theory Variable behavior is another specific kind of dynamic feature that merits additional description. Just as there is evidence that experts subconsciously

internalize *control flow plans*, like the Running Total Loop Plan that accumulates partial totals, there is evidence that experts subconsciously internalize *variable plans* [33]. Variable plans are characterized by the function or role that it serves in a function, how it is initialized and updated, and conditional statements on the variable's value.

Empirically, the number of distinct variable roles found in introductory-level programs is small. While reviewing three introductory programming textbooks written for Pascal, Sajaniemi [105] hand-labeled the role of variables within each provided example program, based only on the pattern of successive values each variable took on. Nine variable roles, e.g., 'stepper' and 'one-way flag', covered 99% of the variables in all 109 programs found in those textbooks. Sajaniemi notes that a single variable can switch roles during execution, 'properly' or 'sporadically'. A proper switch is when its final value while serving in one role is its initial value when serving in the next role. A sporadic switch is one in which the variable is reset to a new value at some point, to serve a new role that may or may not have anything to do with its previous role. Sajaniemi has been credited for creating what is now called in the literature as role of variables theory.

find and look a
green1985progr
The Programmer's Torch–to
summarize how
it's different,
based on sajaniemi2002em
description

Further work independently confirms and operationalizes Sajaniemi's insights. Taherkhani et al. [116] found that 11 variable roles cover all variables in novice-level programs, including object-oriented, procedural, and functional programming styles, and went further to automatically classify algorithms based on variables and some additional features. Gulwani et al. [44] also depend on variable behavior to recognize different algorithmic approaches; they ask teachers to annotate source code, by hand, with key values that differentiate algorithmic approaches. Gulwani et al.'s work was published at a conference six months after the submission of the OverCode manuscript to TOCHI.

OverCode and Foobaz make use of a pipeline that characterizes each solution as (1) a set of variables, which are distinguished from each other by their dynamic behavior during execution on test cases, (2) a set of one-line statements canonicalized in a novel way by those identified variables and (3) the solution's outputs in response to each test case. After OverCode's publication, Gulwani et al. [45] used the same variable value tracing method to cluster solutions, augmenting it with information about control flow structure to overcome syntactic differences between solutions.

Alternative distance functions based on tokens or solution-derived graphs, like ASTs, are included in the next sections for completeness, and so that the techniques can be revisited in discussions of future work.

Token-based distance functions

Many token-based statistical natural language processing techniques can also be applied to code, preferably after preprocessing. For example, Biegel et al. [7] described how w-shingling can capture local patterns within a solution. The w-shingling of a solution is the set of all unique subsequences of w tokens it contains [11]. The resemblance r between two solutions is defined as the number of unique subsequences of w tokens both solutions contain, normalized (divided) by the union of unique subsequences of w tokens contained in either solution. It other words, it is the Jaccard coefficient of the two solutions' w-shinglings. The resemblance distance is defined as $1 - \mathrm{r}$; it is a metric that obeys the triangle inequality. n-grams models are like w-shinglings except instead of capturing just the *set* of unique subsequences of a certain length, they also capture a more global feature: the relative frequencies of these unique subsequences in the entire solution or corpus. Similarly, representing Python programs as TF-IDF vectors calculated from counts of tokens, e.g., keywords, collected across the entire corpus of solutions can capture individual solution's deviations from whole-corpus trends [38].

CCFinder [55] and MOSS [106] (Measure Of Software Similarity) are both pair-wise similarity (clone) detectors. Like w-shingling and n-gram models, MOSS extracts all subsequences of tokens of a specified length; unlike them, the order of these subsequences is preserved. CCFinder [55] is an exception to this pattern; after aggressive pre-processing, it considers all sub-strings in both solutions and looks for matches.

Structure-based features and distance functions

The structure of solutions can be represented as trees, i.e., ASTs, and graphs, e.g., data dependency and control flow graphs. Binary or numerical feature vectors can be computed from the graphs, as well as graph-to-graph metrics of similarity, for use in feedback or assessment [100, 114]. Recent literature uses the full AST for computing pairwise distance

metrics, e.g., the tree edit distance (TED). TED is defined as the minimum cost sequence of node edits that transforms one AST into another, given some set of costs on types of edits.

While a naive TED algorithm scales very poorly with tree size, an optimized TED algorithm [107] makes this computation more feasible; it is only quadratic in both the number of solutions and the size of their ASTs [51]. In contrast, the analysis pipeline used by both OverCode and Foobaz clusters solutions scales linearly with the number and size of solutions.

Huang et al. [51] used the optimized TED algorithm to process approximately a million solutions while executing on a computing cluster. The same analysis pipeline was used by Roger et al. [101]. Yin et al. [131] defined a normalized TED that weighs edits associated with nodes closer to the tree root more heavily than nodes closer to the leaves. This prioritizes high-level structural similarities between solutions and de-emphasizes minor differences in syntax near the AST leaves.

add http://web.stanf papers/progra- mEncoding.pdf

2.3.3 Clustering Solutions

Gaudencio et al. [38] recently investigated whether computers might be able to compare student code solutions as well as teachers. In the process, they found that teachers only agreed with each other between 62% and 95% of the time. Similarly, Rogers et al. [101] found that official graders for a large programming course at Berkeley agreed on solution grades only 47.5% of the time even when there were only 3 possible grades to choose from. In spite of the lack of agreement across expert code evaluators, several research efforts have focused on automatically clustering and grading solutions.

Luxton-Reilly et al. [76] suggests that identifying distinct clusters of solutions can help instructors select appropriate examples of code for helping students learn, e.g., in accordance with the systematic variation suggested by Variation Theory. They also suggest that it is helpful for teachers' own understanding and quality of feedback and guidance. Clustering can also be used to guide rubric creation.

Luxton-Reilly et al. [76] develop a hierarchical clustering taxonomy for types of solu-

tion variations, from high- to low-level: 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. However, it is not yet fully implemented.

The analysis pipeline behind OverCode and Foobaz cluster solutions based on whether or not they have the same output vector and the same set of variables and canonicalized statements. Other systems rely on clustering algorithms applied to solutions whose pairwise distances are determined by TED scores. AutoStyle [22] uses the OPTICS clustering algorithm to cluster solutions based on normalized TED scores. That work was inspired by early published OverCode work [41] on hierarchically clustering solutions. Huang et al. [51] and Rogers et al. [101] cluster solutions by creating a graph where nodes are solutions and an edge between each pair of nodes exists if and only if the TED between their ASTs is below a user-specified threshold. Modularity is used to infer clusters within the graph.

Gross et al. [**?**] use prototype-based clustering, specifically, the Relational Neural Gas technique (RNG) for clustering graded submissions and finding prototypes. Feedback on new submissions is provided by highlighting the differences between the new submission and the closest prototype. Seeing these differences can help students debug their code.

Unlike the work in this thesis, Gross et al. focus on providing feedback to a single student at a time. Graded submissions are clustered, and these clusters help identify problems in news submission. In contrast, OverCode, Foobaz, and GroverCode process ungraded submissions and help staff compose feedback for the whole class or assign grades or feedback to a whole body of submissions at one time. OverCode and GroverCode do highlight differences between submissions to help pinpoint problems, although it is staff, rather than students, who view these differences.

Another approach to clustering solutions comes from the field of Bayesian inference. The following methods are particularly relevant to clustering code:

- Bayesian Case Models (BCM) Kim et al. [59] learn a pre-set number of subspace clusters, where each cluster is represented by an example and a small set of features that play an ”important role” in identifying that cluster. This representation of the clusters has been designed to increase human interpretability of the results. This model has has an interactive version, iBCM [58], where humans can directly modify the cluster example and important features that characterize a cluster.

- Mind the Gap model (MGM) by Kim et al. [60] clusters data while also learning a ”global set of distinguishable dimensions to assist with further data exploration and hypothesis generation.” This is another interpretable clustering algorithm, which may explain itself in ways teachers can understand and base grades on with confidence.

- Dirichlet Process Mixture Models (DPMMs) [] do not require the number of clusters to be set beforehand; the number of clusters solutions are assigned to can grow as the number of solutions grow. However, every data point belongs to one cluster.

get DPMM citation

Maybe add Classification for grading: Aggrawal et al. [successful] Rogers [unsuccessful] Taherkhani [successful]

add http://ethanfast.com/papers/paper.pdf

add https://arxiv.org

2.3.4 Identifying Common Components Across Solutions

Since teacher’s holistic grades can be so inconsistent, they may be internally relying on different rubrics, not equally sensitive to minor differences, and/or weighing factors differently. Teachers may be more consistent if, rather than generating holistic grades, they can annotate or grade particular components, mistakes, or design choices within solutions. Three existing approaches support this goal: (1) Create a classifier for every component, mistake, or design choice teachers are interested in, similar to what was done for web pages in the Webzeitgeist dataset by Lim et al. [69]. (2) Model solutions as mixtures of components or design choices using mixture modeling. They have already been applied to source code [] and student solutions to open-response mathematical questions [] [8, 72].

(3) Create a 'code search engine' which takes AST nodes or subtrees as queries and retrieves solutions in the database that contain them, as Nguyen et al. [89] did for solutions to the same problem and Fast et al. [36] did for general open source code.

There is a particularly rich literature on Bayesian mixture models. The following methods are particularly relevant to modeling code as mixtures of choices:

- Latent Dirichlet Allocation (LDA) [**?**] is a mixture model that is typically applied to natural language. In that context, it learns topics, i.e., distributions over words, as well as the distributions over topics found in each document.

- Correlated Topic Models (CTM) [] are like LDA but topics are no longer assumed to be independent. In other words, the topics learned by the model can be correlated. If it it possible to formulate the inputs such that the learned latent topics represent design decisions, this could capture the reality that design choices are not independent of each other in code composition.

- Hierarchical Dirichlet Process Models (HDPs) [], like DPMMs, do not require a preset number of clusters. Unlike DPMMs, solutions do not belong to a single cluster. Like LDA, solutions contain features, and each feature belongs to a cluster. Solutions can contain features from multiple clusters. This method emulates LDA with no preset number of clusters.

- Models and inference algorithms built on Indian Buffet Processes (IBPs) by Doshi-Velez et al. [30] are like HDPs, but individual features can belong to multiple clusters.

t CTM and
DP citations

2.3.5 Visualization and Interfaces

There are several existing visualizations or interfaces that help users understand how solutions vary within a large corpus of solutions to a common problem and/or why a given set of solutions are grouped together. Even tools that are not built for this purpose, such as file comparison tools, do have useful features to consider when designing new interfaces. Most

highlight inserted, deleted, and changed text. Unchanged text is often 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.

In contrast, the solutions maps in Cody, a Matlab programming game[1], do help users pick and compare pairs of solutions from hundreds of solutions to the same programming exercise. The 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. Users can select pairs of points to see the code they correspond to side-by-side beneath the solution map. Despite the simplicity of this metric, solution maps can provide quick and valuable insight when exploring differences among large numbers of solutions [40]. This can help game players learn alternative, possibly better, ways to solve a problem using the Matlab programming language, including its extensive libraries.

Huang et al. [51] and Rogers et al. [101] have an alternative but still point-based representation of hundreds or thousands of solutions. They create graphs where each node is a solution and links between nodes indicate similarity scores beneath a certain threshold. Inter-Node and inter-cluster distances correspond to syntactic similarity. Clusters are colored using modularity, a measure of how well a network decomposes into modular communities. Rogers et al. [101] built a grading interface on top of this clustering process; graders graded one solution at a time, grouped by cluster.

AutoStyle [22] also uses a point-based display of solutions. The authors visualize all solutions on the screen using a t-SNE 2D visualization. Similar to the previous clustering interfaces, each point represents a solution and its color indicates its cluster. Hovering over

[1] mathworks.com/matlabcentral/cody

a point reveals the solution it represents. Using this interface, teachers hand-annotate each cluster with a label, i.e., 'good', 'average', or 'weak' and a hint about how to improve the solution. For each cluster, the teacher must also choose an exemplar solution from a *slightly better* cluster as an example of what to shoot for.

In contrast to point-based displays, the OverCode interface puts the code front and center, borrowing display techniques from file comparison tools, adding filtering mechanisms and interactive clustering through rewrite rules on top of the clustering done by the analysis pipeline. The pipeline's clustering criteria allows the user interface to efficiently name clusters with a representative exemplar solution automatically.

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. The powergrading paradigm [5] enables teachers to assign grades or write feedback to many similar answers at once. Their interface focused on powergrading for short-answer questions from the U.S. Citizenship exam. After machine learning clustered answers, the frontend allowed teachers to read, grade, or provide feedback on similar answers simultaneously. 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" [12].

OverCode was also inspired by information visualization projects like WordSeer [87, 88] and CrowdScape [103]. WordSeer helps literary analysts navigate and explore texts, using query words and phrases [86]. CrowdScape gives users an overview of crowdworkersâĂŹ performance on tasks.

2.4 Personalized Support

Several types of solutions have been deployed to help students get the personalized attention they need. These solutions span the spectrum from recruiting more teaching assistants from the ranks of previous students [92] to automating hints using program synthesis or intelligent tutoring systems.

Singh et al. [110] use a constraint-based synthesis algorithm to find the minimal

changes needed to make an incorrect Python 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. The system can automatically deliver hints to students about these changes at various levels of specificity.

Intelligent tutoring systems can provide personalized hints and other assistance to each student based on a pre-programmed student model. For example, previous systems sought to provide support through the use of adaptive scripts [65], or cues from the studentâĂŹs problem-solving actions [27]. Despite the advantage of automated support, intelligent tutoring systems often require domain experts to design and build them, making them expensive to develop. Furthermore, domain experts who generate these hints may also suffer from the *curse of knowledge*: the difficulty experts have when trying to see something from a novice's point of view [15].

Unlike intelligent tutoring systems, the HelpMeOut system [47] does not require a pre-programmed student model. It assists programmers during their debugging by suggesting code modifications mined from debugging performed by previous programmers. However, the suggestions lack explanations in plain language unless they are added by experts (teachers), so the limits imposed by the time, expense, and curse of knowledge of experts still apply.

Rivers and Koedinger [98] 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.

Discussion forums derive their value from the content produced by the teachers and students who use them. These systems can harness the benefits of peer learning, where students can benefit from generating and receiving help from each other. However, as the system has no student model, the information is available to all students whether or not it is ultimately relevant. Students can receive personalized attention only if they post a question and receive a response.

Include "curren
ITSs require an
exact formal-
ization of the
underlying do-
main knowledg
which is usu-
ally a substanti
amount of work
researchers hav
reported 100-
1000 hours of
authoring time
needed for one
hour of instruc-
tion [MBA03]
from 'Feedback
Provision Strate
gies in Intelli-
gent Tutoring
Systems Based
on Clustered So
lution Spaces"'

Peer-pairing can stand in place of staff assistance, to both reduce the load on teaching staff and give students a chance to gain ownership of material through teaching it to someone else. Weld et al. speculate about peer-pairing in MOOCs based on student competency measures [127], and Klemmer et al. demonstrate peer assessments' scalability to large online design-oriented classes [62]. Peer instruction [82] and peer assessment [117] have been integrated into many classroom activities and have also formed the basis of several online systems for peer-learning. For example, Talkabout organizes students into discussion groups based on characteristics such as gender or geographic balance [1].

Recent work on learnersourcing proposes that learners can collectively generate educational content for future learners while engaging in a meaningful learning experience themselves [61, 85, 126]. For example, Crowdy enables people to annotate how-to videos while simultaneously learning from the video [126]. The targeted learnersourcing workflows presented in this thesis expand on learnersourcing by requesting the contributions of specific learners who, by virtue of their work so far, are uniquely situated to compose a hint for fellow learners.

onsider adding
dditional papers
om notes

ld
tp://research.microsoft.com/en-
/um/people/sumitg/pubs/fse14.pdf
Our key insight
that the algo-
hmic strategy
mployed by
program can
identi ed by
bserving the
lues com-
uted during the
ecution of the
ogram."

Chapter 3

OverCode

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 [81], and understanding this variation is important for providing appropriate, tailored feedback to students [5, 51]. 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 [39, 51, 89] 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

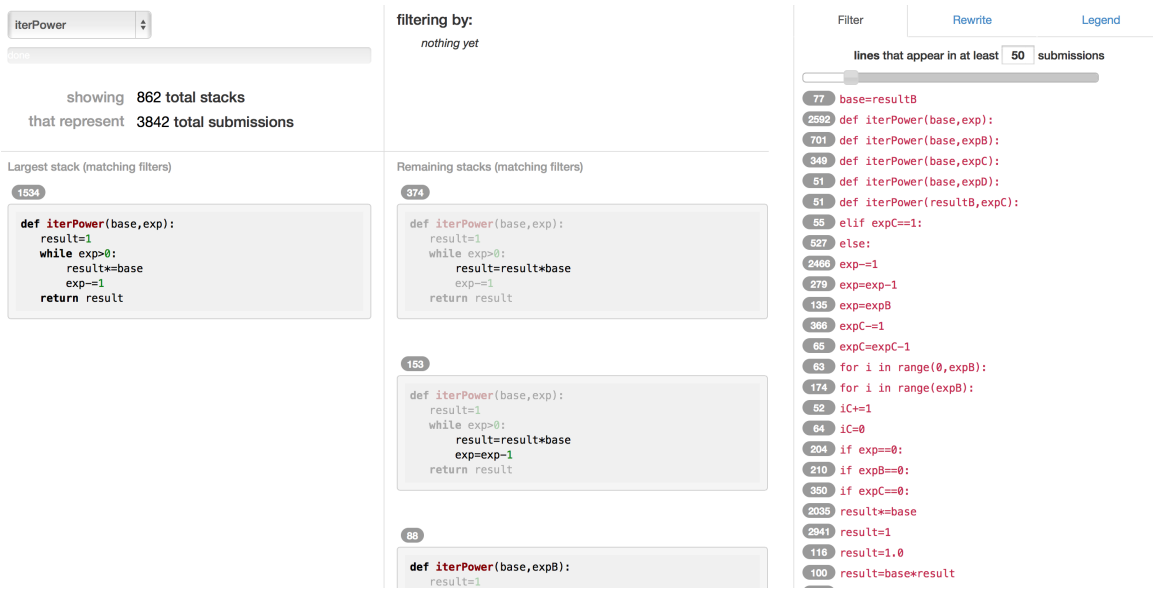


Figure 3-1: The OverCode user interface. The top left panel shows the number of clusters, called *stacks*, and the total number of solutions visualized. The next panel down in the first column shows the largest stack, while the second column shows the remaining stacks. The third column shows the lines of code occurring in the normalized solutions of the stacks together with their frequencies.

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.

This chapter presents 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.

OverCode uses a novel clustering technique that creates clusters of identical normalized code, in time linear in both the number of solutions and the size of each solution. The normalized code is readable, executable, and describes every solution in that cluster. The

normalized code is shown in a visualization that puts code front-and-center (Figure 3-1). 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, the OverCode interface is compared 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.

3.1 OverCode

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

The OverCode interface was iteratively designed and developed 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 3-1. In this section, the intended use cases and the user interface are described. In Section 3.2, the backend program analysis pipeline is described in detail.

3.1.1 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.

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 normalized 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.

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.

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 [81], students can learn through concrete examples of these multiple solutions, which vary along various conceptual dimensions.

3.1.2 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 3-1, the default tab shows a list of lines of code that occur in different normalized 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.

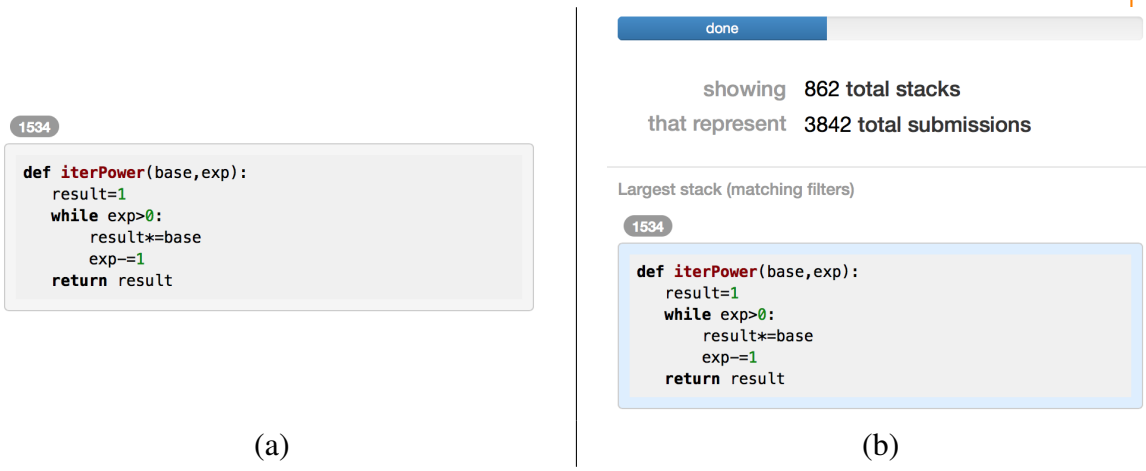
Stacks

A stack in OverCode denotes a set of similar solutions that are grouped together based on a similarity criterion defined in Section 3.2. For example, a stack for the `iterPower` problem is shown in Figure 3-2(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 normalized solution that describes all the solutions in the stack. See Section 3.2 for details on the normalizing 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 3-2(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.

add that now th
raw solutions in
side the stack a
also displayed,
but this was no
included during
user study tests



(a) (b)

Figure 3-2: (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.

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 3-3 shows the differences between the normalized solutions of the two largest stacks. In earlier iterations of the user interface, lines in stacks that were not shared with the largest stack

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

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, the visible noise is reduced, while still keeping differences between stacks salient.

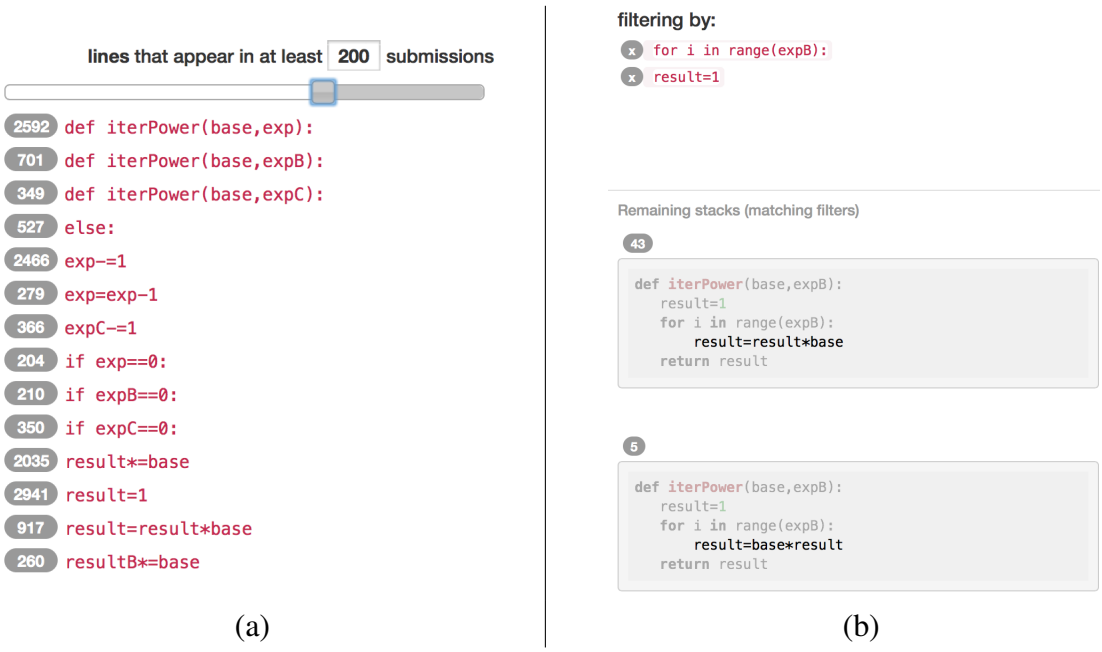
Filtering Stacks by Lines of Code

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 3-4(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 normalized solutions have those selected lines of code are shown. Figure 3-4(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`.

Rewrite Rules

There are often small differences between the normalized 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



(a) (b)

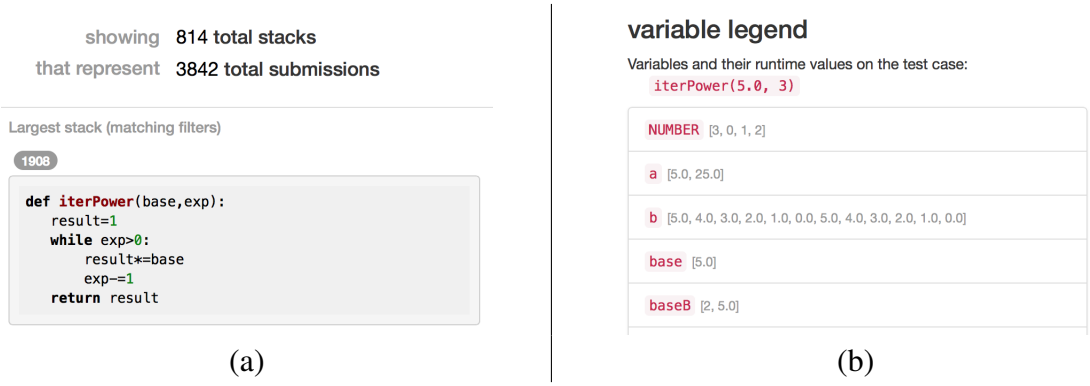
Figure 3-4: (a) The slider allows filtering of the list of lines of code by the number of solutions in which they appear. (b) Clicking on a line of code adds it to the list of lines by which the stacks are filtered.

more easily find other differences.

A rewrite rule is described with a left hand side and a right hand side as shown in Figure 3-5(a). The semantics of a rewrite rule is to replace all occurrences of the left hand side expression in the normalized solutions with the corresponding right hand side. As the rewrite rules are entered, OverCode presents a preview of the changes in the normalized solutions as shown in Figure 3-5(b). After the application of the rewrite rules, OverCode collapses stacks that now have the same normalized solutions because of the rewrites. For example, after the application of the rewrite rule in Figure 3-5(a), OverCode collapses the two biggest iterPower stacks from Figure 3-1 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 3-6(a), the number of stacks now drop from 862 to 814.

(a) (b)

Figure 3-5: (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 normalized solutions because of the application of the rewrite rule.



(a) (b)

Figure 3-6: (a) The merging of stacks after application of the rewrite rule shown in Figure 3-5. (b) The variable legend shows the sequence of dynamic values that all program variables in normalized solutions take over the course of execution on a given test case.

Variable Legends

OverCode also shows the sequence of values that variables in the normalized solutions take on, over the course of their execution on a test case. As described in Section 3.2, a variable is identified by the sequence of values it takes on during the execution of the test case. Figure 3-6(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 normalized 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.

3.2 Implementation

The OverCode user interface depends on an analysis pipeline that canonicalizes solutions in a manner designed for human readability, referred to here as *normalizing*. The pipeline then creates stacks of solutions that have become identical through the normalizing process. The pipeline accepts, as input, a set of solutions, expressed as function definitions for $\mathtt{f}(\mathtt{a}, ...)$, and one test case $\mathtt{f}(\mathtt{a}_1, ...)$. Solutions that enter the pipeline are referred to as *raw*, and the solutions that exit the pipeline as *normal*. Examples, beginning with `iterPower`, are presented below to illustrate this pipeline.

3.2.1 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 with the PythonTidy package.[1] 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 normalizing 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. Included for the purpose of illustrating this pipeline, the following examples are derived from executing definitions of `iterPower` on a `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 log-

[1] Created by Charles Curtis Rhode. https://pypi.python.org/pypi/PythonTidy/

ger [46] 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, the pipeline extracts 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. The pipeline analyzes all program traces, identifying which variables' sequences are identical. Variables that have identical sequences across two or more program traces are referred to as *common variables*. 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)	Unique 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)	• `3,2,1,0`: `exp` (Student C)
• `0,1,2`: `i` (Student A) `k` (Student B)	

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 3.2.2 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	Unique Variables, Named
• `base: 5.0`	
• `exp: 3`	
• `result: 1, 5.0, 25.0, 125.0`	• `exp__: 3,2,1,0`
• `i: 0,1,2` (common name tie broken by random choice)	

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

Normal Student A Code (After Renaming)

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

Normal Student B Code (After Renaming)

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

Normal Student C Code (After Renaming)

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

6. Make stacks. The pipeline iterates through the normal solutions, making stacks of solutions that share an identical *set* of lines of code. The pipeline compares *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 normal solutions within the stack are identical, with the possible exception of the order of their statements.

In the examples below, the normal C and D solutions have the exact same set of lines, and both provide correct output, with respect to the autograder. Therefore, the difference in order of the statements between the two solutions is not communicated to the user. The two solutions are put in the same stack, with one solution arbitrarily chosen as the visible normalized 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 Normal Student A (After Renaming)

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

Stack 2 Normal Student B (After Renaming)

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

Stack 3 Normal Student C (After Renaming)

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

Stack 3 Normal 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 processed in this pipeline have already been marked correct by an autograder, the program tracing [46] 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, they are excluded from our analysis. Less than five percent of the solutions in each of the three problem datasets are excluded.

3.2.2 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, referred to here 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, the pipeline appends 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`.

nange to sub-
ripts now?

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
```

Normal 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
```

Normal Student E (After Renaming)

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

Multiple-instances collision. The pipeline identifies 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, the pipeline preserves the solution author's variable name choice for all the instances of that common variable in that solution. If a solution 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 solution authors preserved variable name, so that the interface, and the human reader, do not conflate them.

In the following example, the solution 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)
```

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
```

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)
```

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 the following example, 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
```

3.2.3 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 [51], 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. Given the assumption that all solutions in the pipeline are correct, 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 the corresponding variable sequences can be hashed and then checked 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.

3.3 Dataset

For evaluating both the analysis pipeline and the user interface of OverCode, the dataset of solutions is collected from 6.00x, an introductory programming course in Python that was offered on edX in fall 2012. Three exercise problems were chosen. This dataset consists of student solutions submitted within two weeks of the posting of the those three problems. There are thousands of submissions to these problems in the dataset, from which all correct

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

Figure 3-7: Number of solutions for the three problems in our 6.00x dataset.

solutions (tested over a set of test cases) were selected for analysis. The number of solutions analyzed for each problem is shown in Figure 3-7.

- **`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 3-8 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 3-9 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 3-10 for examples.

The three exercises were selected for analysis because they are representative of the typical exercises students solve in the early weeks of an introductory programming course. The three exercises 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.

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
```

Figure 3-8: Example solutions for the `iterPower` problem in our 6.00x dataset.

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)
```

Figure 3-9: 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
```

Figure 3-10: Example solutions for the `compDeriv` problem in our 6.00x dataset.

3.4 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 normalizing process. The evaluation was performed on a Macbook Pro 2.6GHz Intel Core i7 with 16GB of RAM.

move we's

Running Time The complexity of the pipeline that generates stacks of solutions grows linearly in the number of solutions as described in Section 3.2.3. Figure 3-11 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 normal thousands of student solutions and generate stacks within few minutes for each problem.

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

Figure 3-11: Running time and the number of stacks and common variables generated by the OverCode backend implementation on our dataset problems.

Distribution of Stacks The distribution of initial stack sizes generated by the analysis pipeline for different problems is shown in Figure 3-12. 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 normalized solutions for each problem are shown in Figure 3-13.

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 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 Over-

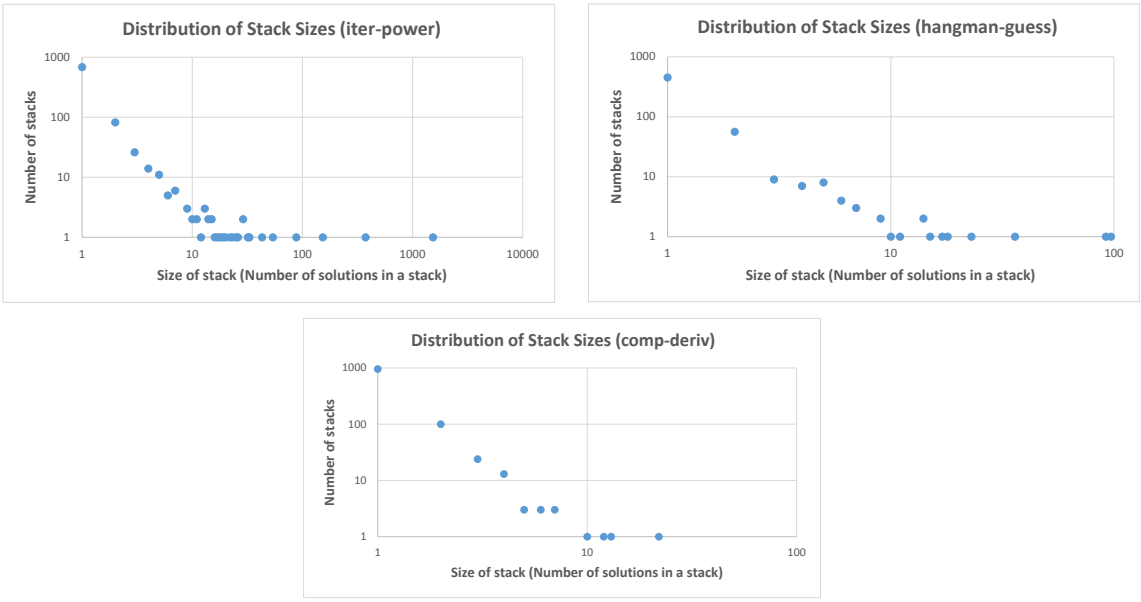


Figure 3-12: The distribution of sizes of the initial stacks generated by our algorithm for each problem, showing a long tail distribution with a few large stacks and a lot of small stacks. Note that the two axis corresponding to the size of stacks and the number of stacks are in logarithmic scale.

Code 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 3-11 and some examples of these common variable names are shown in Figure 3-14. Figure 3-14 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 vari-

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
```

(a)

97

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

92

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

(b)

22

```
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
```

13

```
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
```

(c)

Figure 3-13: The two largest stacks generated by the OverCode backend algorithm for the (a) `iterPower`, (b) `hangman`, and (c) `compDeriv` problems.

able renaming is shown in Figure 3-15. A large majority of the collisions were Common/Common Collisions. For example, Figure 3-14 shows the common variable name `exp` for two different sequences of 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`.

Common Variable Name	Occur-rence Count	Sequence of Values	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, ···
exp	749	[3]	exp, count, temp, exp3, exp2, exp1, inexp, old_exp, ···
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],··· ,[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, ···

Figure 3-14: 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 of 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

Figure 3-15: The number of common/common, multiple instances, and unique/common collisions discovered by our algorithm while renaming the variables to common names.

3.5 User Study 1: Writing a Class Forum Post

The 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. Two user studies were designed 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 the control condition, so instead of measuring time to finish the entire set of solutions, the experiment measures what 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, the first hypothesis is tested:

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

3.5.1 OverCode and Baseline Interfaces

Two interfaces were designed, referred to here as OverCode and the baseline. The OverCode interface and backend are described in detail in Section 3.1. The baseline interface was a single webpage with all student solutions concatenated in a random order into a flat list (Figure 3-16, left). This design emulates 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 [12]. 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.

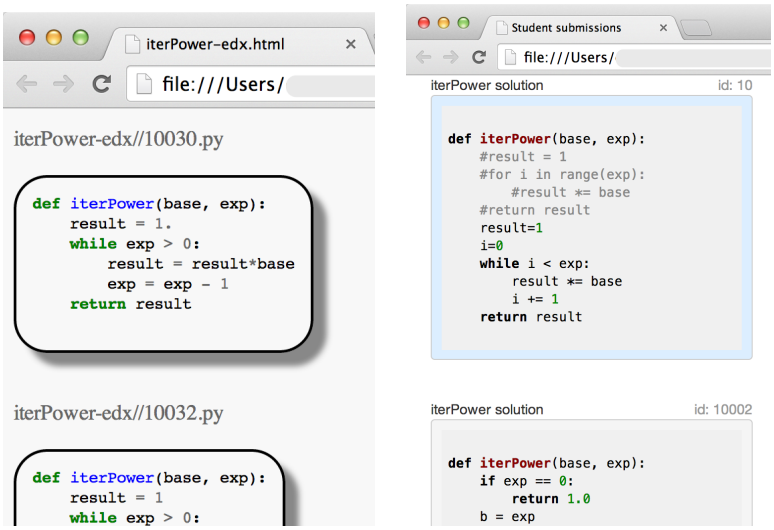


Figure 3-16: The baseline interface used in the Forum Post study (left) and the Coverage study (right).

3.5.2 Participants

Participants were recruited 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.

3.5.3 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.

3.5.4 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 3.3) 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.

3.5.5 Procedure

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[2] 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.

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.

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 task completion times were not measured, 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

[2] 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)

by teachers, e.g., recursive solutions to `iterPower` when its prompt explicitly calls for an iterative solution. As a secondary task, subjects were asked 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.

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.

3.5.6 Results

H1 is supported by ratings from the post-study survey (see Figure 3-17). 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).

From the surveys conducted after subjects completed each interface condition, 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!

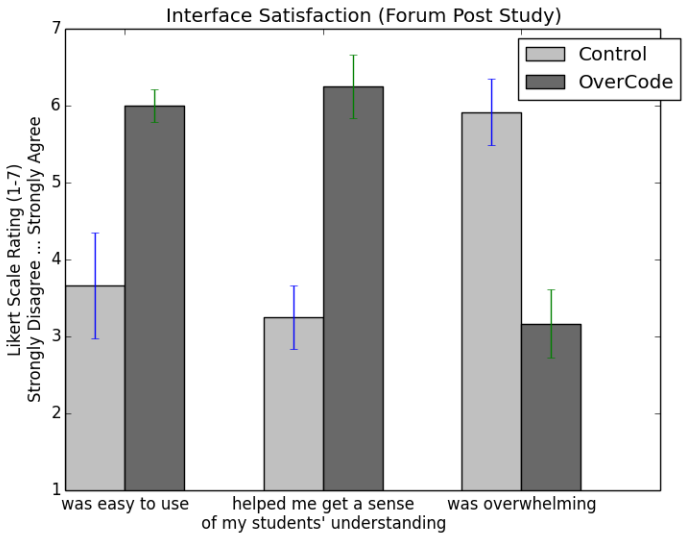


Figure 3-17: H1: Interface satisfaction Mean Likert scale ratings (with standard error) for OverCode and baseline interfaces, after subjects used both to perform the forum post writing task.

- 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.

3.6 User Study 2: Coverage

A second 12-person study was designed, 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. The following hypotheses were tested:

- **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.

3.6.1 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.

3.6.2 Interface Modifications

Prior to the second study, both the OverCode and baseline interfaces were slightly modified (see differences in Figure 3-16) 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*[3] 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 3.6.4.

- All interface interactions were logged in the browser console, allowing for the tracking of 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.

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

- Where it differed slightly before, the styling of code in the baseline condition was changed to exactly match the code in the OverCode condition.

3.6.3 Procedure

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.

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.

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 representa-

tive strategies. Combined with interaction logging done by the system, this enabled us to measure read coverage.

Surveys

move we's
om here on
own

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.

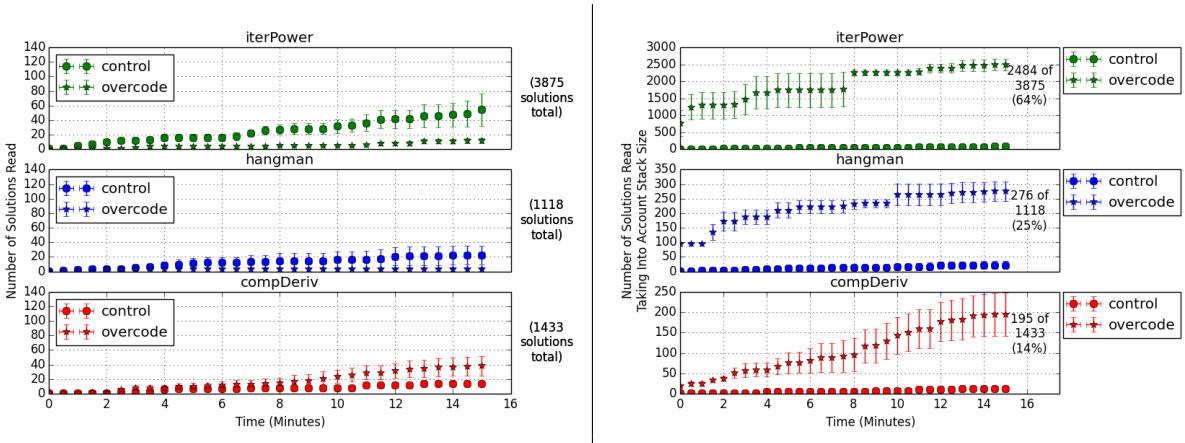


Figure 3-18: In (a), we plot the mean number of normalized solutions representing stacks read in OverCode over time versus the number of raw solutions read in the baseline interface over time while performing the 15 minute long Coverage Study task. In (b), we replace the mean number of normalized solutions 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.

3.6.4 Results

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 normalized 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 3-18 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.

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:

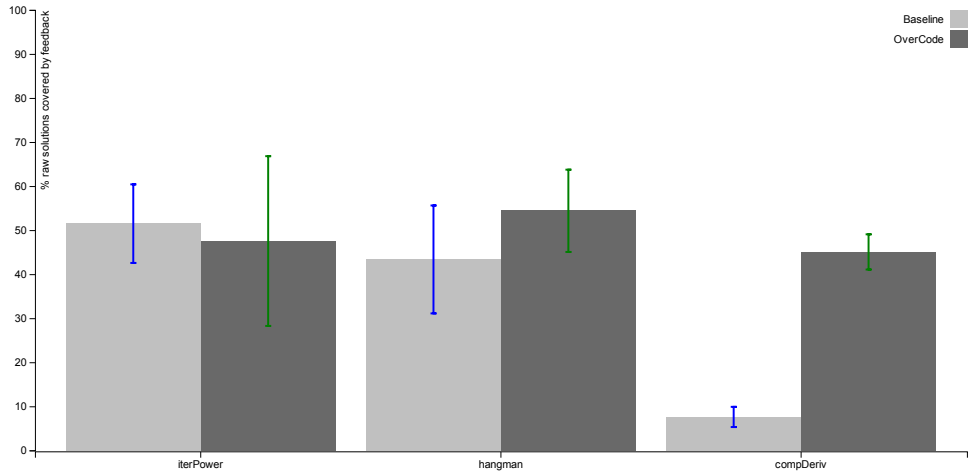


Figure 3-19: Mean feedback coverage (percentage of raw solutions) per trial during the coverage study for each problem, in the OverCode and baseline interfaces.

- 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 normalized 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 normalized code of its source stack, we found the normalized version of each line, with variable names and whitespace normalized.

- Find the raw solutions that include the set of normalized lines, using a map from stacks to raw solutions provided by the backend pipeline.

Figure 3-19 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 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 $U = 0$, $n1 = n2 = 4$, $p < 0.05$).

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' under-

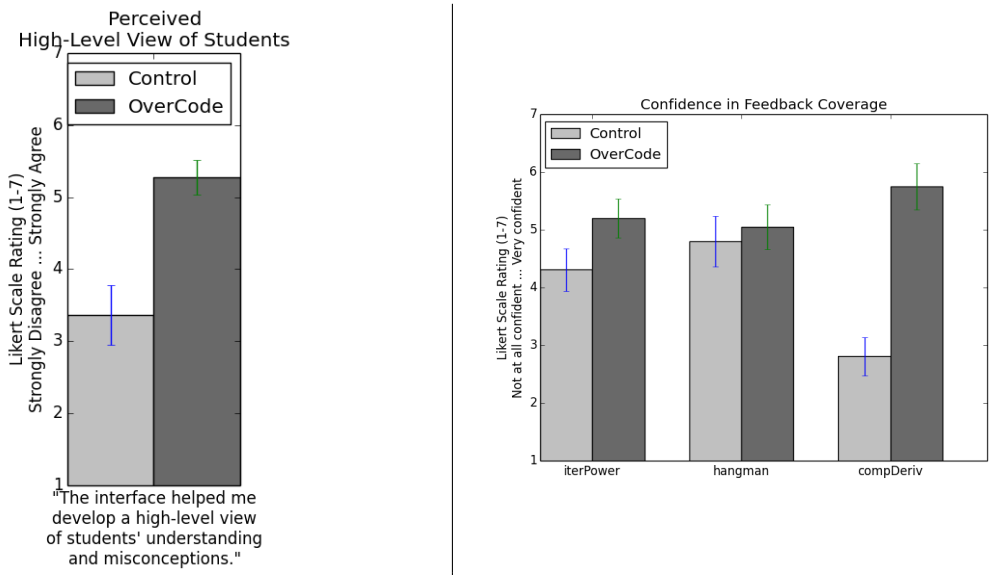


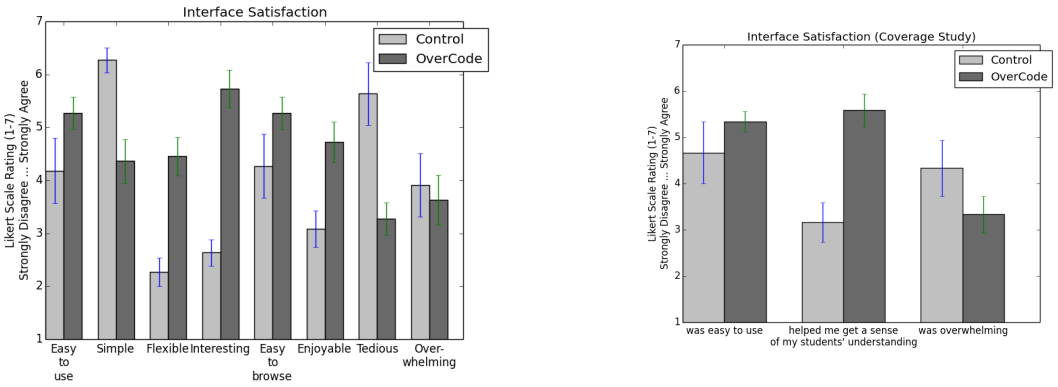
Figure 3-20: (a) Post-condition perception of coverage (excluding one participant) and (b) Confidence ratings that identified strategies frequently occurred (1-7 scale).

standing 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 3-20(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 3-20(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).

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



(a) (b)

Figure 3-21: 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.

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 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 3-21.

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 3-21.

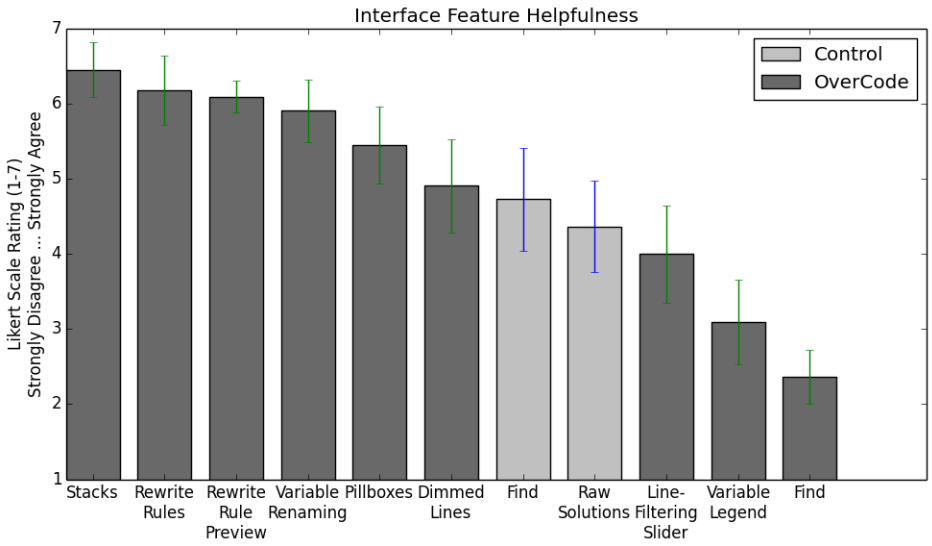


Figure 3-22: Mean Likert scale ratings (with standard error) for the usefulness of features of OverCode and baseline. "Find" is the web browser's built-in find, which appears twice because users rated its usefulness for each condition.

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 3-22.

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 3-4) 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 3-4a) and the variable legend (Figure 3-6b) 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.

3.7 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.

3.7.1 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 normalized 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 3-18(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 3-18(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.

3.7.2 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.

3.7.3 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.

3.8 Post-Publication Modifications

3.8.1 Multiple Test Cases and Efficiency

Since publication, the OverCode pipeline has been modified in several ways. It has been split into two phases: solution preprocessing and batch processing. During preprocessing, each solution's formatting is standardized and all comments are removed. It is then executed on one or more test cases, which makes the resulting canonicalization more robust to any individual poorly chosen test case. The full program trace and output is recorded for every test case.

This logging of solution execution can be one of the longer steps in the OverCode analysis pipeline, so preprocessing occurs once were solution and is saved in its own pickle file. New solutions can be preprocessed once, without affecting previously preprocessed solutions. Batch processing takes, as input, all the existing preprocessed results and canonializes variable names. This must occur as a batch operation because variable renaming depends on the behavior and name of every variable in every solution.

3.8.2 Collecting Additional Information per Line

It was also noticed that a single-line difference between two solutions could, by affecting a variable's behavior, cause the variable renaming process to propagate that difference as a

variable name change throughout all the lines that mention the affected variable. This non-locality of difference made it harder to spot in the user interface where the actual syntactic difference between two canonicalized solutions was. As a result, stacks are rendered in the user interface slightly differently. Now, even if the canonicalized variable names are different between two syntactically identical lines of code in two different stacks, if the variables take on the same sequence of values *just within that line of code*, it will not be highlighted as different in the user interface. Only the line with the syntactic difference will be highlighted.

In order to implement this, the modified pipeline examines the program trace collected during preprocessing and the AST for each solution and compiles the following information for each line of code in each solution:

1. a template, i.e., the line of code with variables replaced by blanks, e.g., `__ += 1` and `for __ in range(__):`

2. an ordered list of the common variables' behavior and their canonicalized names corresponding to each blank in the line, e.g., the blank in `__ += 1` can be associated with one of several common variables, depending on the rest of the solution

3. an ordered list of the sequences of values each blanked-out variable in the line took on during execution, e.g., the first blank in `for __ in range(__):` might iterate through `1,2,3` while, with each visitation of this line during execution, the second blank stays constant at `[1,2,3],[1,2,3],[1,2,3]`

Items 1 and 2 in this list together contain the information in the original canonicalized lines in OverCode and information about the line's syntax, regardless of the variables' identities. Items 1 and 3 capture the information necessary to tell what the variable behavior is just within that one line and whether two lines in two different stacks will be highlighted as different from each other.

This information is also used for variable renaming in incorrect solutions, as described in Section 6.2.2.

For clarity in the OverCode user interface, the modifiers appended to common variables to resolve Common/Common collisions are no longer capitalized letters. Instead, they are

rendered in the interface as numerical subscripts indicating whether they are the second or third or fourth (etc.) most frequently occurring common variable with that name across all the solutions in the collection analyzed.

3.9 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 normalized 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.

Chapter 4

Foobaz: Feedback on Variable Names at Scale

Current traditional feedback methods for solutions to programming assignments include hand-grading student code for substance and style. Unfortunately, those methods are labor intensive, potentially inconsistent across graders, and do not scale to the sizes associated with Massive Open Online Courses (MOOCs). The scaling difficulty is particularly important when considering that some residential course enrollments at prominent universities, like UC Berkeley's CS61A, are rising above the thousand-student mark [77]. Some Computer Science teachers, such as MIT's John Guttag and Ana Bell, are simultaneously teaching hundreds of students in residential programming courses and thousands of online students in MOOC versions.

Variable naming is a specific, important aspect of writing readable, maintainable code, and many teachers want to give feedback on it. The quality of a name is most easily judged when its role within the surrounding code is known. However, at scale, teachers cannot read every solution. Programming education at scale opens up new challenges for processing and presenting thousands of solutions so that teachers can more easily view them. Teachers also cannot write comments on each solution. This difficulty motivates the creation of tools that help teachers give customized feedback to subsets of students for whom that feedback is relevant.

This chapter introduce Foobaz, a user interface for giving tailored feedback on student

variable names at scale. Foobaz enables teachers to explore and comment on the quality of student-chosen variable names, given the role the variables play in students' code (see Figure **??**). A variable's role is a function of the sequence of values that the variable contains during program execution. The variety of student-chosen variable names for each role makes evaluating every one prohibitive. Using Foobaz, the teacher can label a small subset of good and bad names for each role.

Foobaz then uses these labeled variable names to create pedagogically valuable active learning exercises in the format of multiple choice quizzes. These quizzes are a form of feedback for many more students than just those whose variable names receive a teacher's label. The quizzes also allow students to see examples of good and bad alternatives, rather than just receive a label on one of their own names.

Foobaz personalizes teacher quizzes for each student, so that students can consider good and bad names for variables that exist in their own solutions. Personalized quizzes render the student's original code submission with a specific variable replaced by an arbitrary symbol. The quiz presents the student with several variable names as candidate replacements for the symbol, one of which may be the student's original choice. The student selects appropriate labels for the variable names before checking their labels against teacher labels and comments.

In two user studies, the capabilities and workflow enabled by this novel interface are demonstrated. The first study shows that the interface helped teachers give personalized variable name feedback on thousands of student solutions from an introductory programming MOOC on edX. In the second study, students composed solutions to the same programming exercises and received personalized quizzes generated with Foobaz by teachers in the first study.

move we from
e rest of this
apter

4.1 User Interface

Consider an introductory programming MOOC where thousands of students submit correct solutions to a programming exercise. Imagine that many of these solutions include a

variable that takes on the same sequence of values, like a running sum of the elements in an input argument. While most students decide to name this variable `result`, others decide to give it obscure or less descriptive names like `p`, `val1`.

The quality of a variable's name is most easily judged when the teacher understands its algorithmic role and relationship to other variable names within the surrounding code. At scale, this can be difficult and frustrating, because while variable roles can be repetitive across many solutions (e.g., a particular running sum), their names can be unpredictable (`result`, `val1`, `s`). Instead of browsing student submissions in a linear fashion, it would be better if the teacher could provide feedback on the basis of variable roles.

In Foobaz, teachers can browse *stacks* of student submissions. A stack is a set of solutions whose code is identical after normalizing formatting and variable names, removing comments, and ignoring the exact order of statements. Within a stack, the teacher can browse the different sets of variable names that students chose, label some of them as, e.g., "too short" or "misleading," and add comments.

Teacher labels and comments are used to provide students with tailored feedback in the form of personalized quizzes on variable names. By providing feedback on only a few variable names, personalized quizzes are generated that are potentially relevant to thousands of students. Foobaz is distinct from powergrading systems: instead of grading as many names as possible, the system produces personalized quizzes that have excellent examples of good and bad variable names students would not otherwise get to see and learn from.

4.1.1 Producing Stacks and Common Variables

Foobaz enables feedback at scale because teachers can browse solutions on the basis of *stacks of similar solutions* and *common variables*. These groupings are automatically produced by the OverCode analysis pipeline, which starts by executing each student's solution on a single test case and tracking the sequences of values that variables take on. Common variables are identified as those that behave the same way (i.e., take on the same sequence of values) across multiple solutions. Raw solutions are then normalized by renaming instances of common variables with their most popular names, as well as removing comments

and extra whitespace. Normalized solutions that have the same set of lines can be grouped together into a *stack*, with a single representative on top for the teacher to read.

The stacking performed by the system directly reduces the number of implementations that a teacher needs to analyze in order to provide feedback to the majority of the class. Furthermore, Foobaz is sensitive to *variable roles*, meaning that variables that behave the same across different stacks are linked together as a single common variable. The result is that feedback provided for a variable in one stack is propagated to variables that play out the same behavior in *other* stacks.

4.1.2 Rating Variable Names

The Foobaz interface lets the teacher rate variable names in the context of their role in the program. Figure **??** shows the teacher's view while they perform this task.

The teacher is presented with a scrollable list of stacks. Each stack is represented as normalized stack code followed by a table of alternative variable names. Since some of the tables are taller than the screen, the normalized stack code is pinned to the screen in such a way that it remains visible until the next stack is scrolled into view.

Each column of the table represents the common variables occurring in the stack, where each common variable's most popular name serves as the column header. Each row of the table represents a unique set of variable names used in a solution (e.g., `secretWord`, `lettersGuessed`, `guessedWord`, `char`). We show *sets* of variable name choices, rather than independent columns of variable names, because we found in early pilot testing that variable names can at times make more sense when seen as a group, rather than as individual naming decisions. This helps give teachers the context and confidence to assign quality judgments.

As the teacher brushes over the names of a common variable in the table, its occurrences in the normalized code are highlighted, so they can develop an understanding of the variable's role in the solution. The teacher can then go down the list of student-chosen names, rating as many of them as they desire using three different labels: "misleading or vague," "too short," or "fine." These labels were based on early pilot testing with beginner

programmers but future iterations of Foobaz may support teacher-added labels. Next to each name, they can also see how frequently it was given to that common variable across all solutions in the dataset, and can sort the entire table by this frequency. In order to draw teachers' attention to student-chosen variable names, variables with names that match one of given parameter names provided with the homework prompt are greyed out. The long tail of infrequently used names can be a place where both the best and worst examples of names can be found.

It is important to note that each common variable is likely to occur in multiple stacks. When the teacher selects a particular name, or set of names, for a common variable, occurrences in other stacks are highlighted as well. When they assign a label to a name, the label is propagated to all uses of the name for that common variable, across all the stacks. This has the effect of "filling down" the teacher's annotations. As teachers scroll down, they see that much of their feedback has been propagated for them, letting them focus on the remaining best and worst examples they might find.

A progress bar at the top of the page indicates the coverage of their feedback across all variable names. Since teachers were motivated by the progress bar to maximize their coverage, we provided a button which would select and scroll into view the next most popular, yet unlabeled name for a common variable. Also included to maximize efficiency, the interface supports navigation through the variable names in the table using arrow keys. Teachers can also press hotkeys to rate variables instead of clicking on one of the three quality judgments, e.g., press 2 to rate a variable name as "too short."

4.1.3 Making Quizzes

Each quiz is an active learning exercise that asks the student to think about good and bad names for a common variable. Quizzes begin by showing a solution with that common variable's name replaced by an arbitrary symbol everywhere it occurs. In a personalized quiz, the solution is the student's own, as shown in Figure 4-1. The quiz presents the student with several variable names as candidate replacements for the symbol, one of which may be the student's original choice. The student labels these names before checking their labels

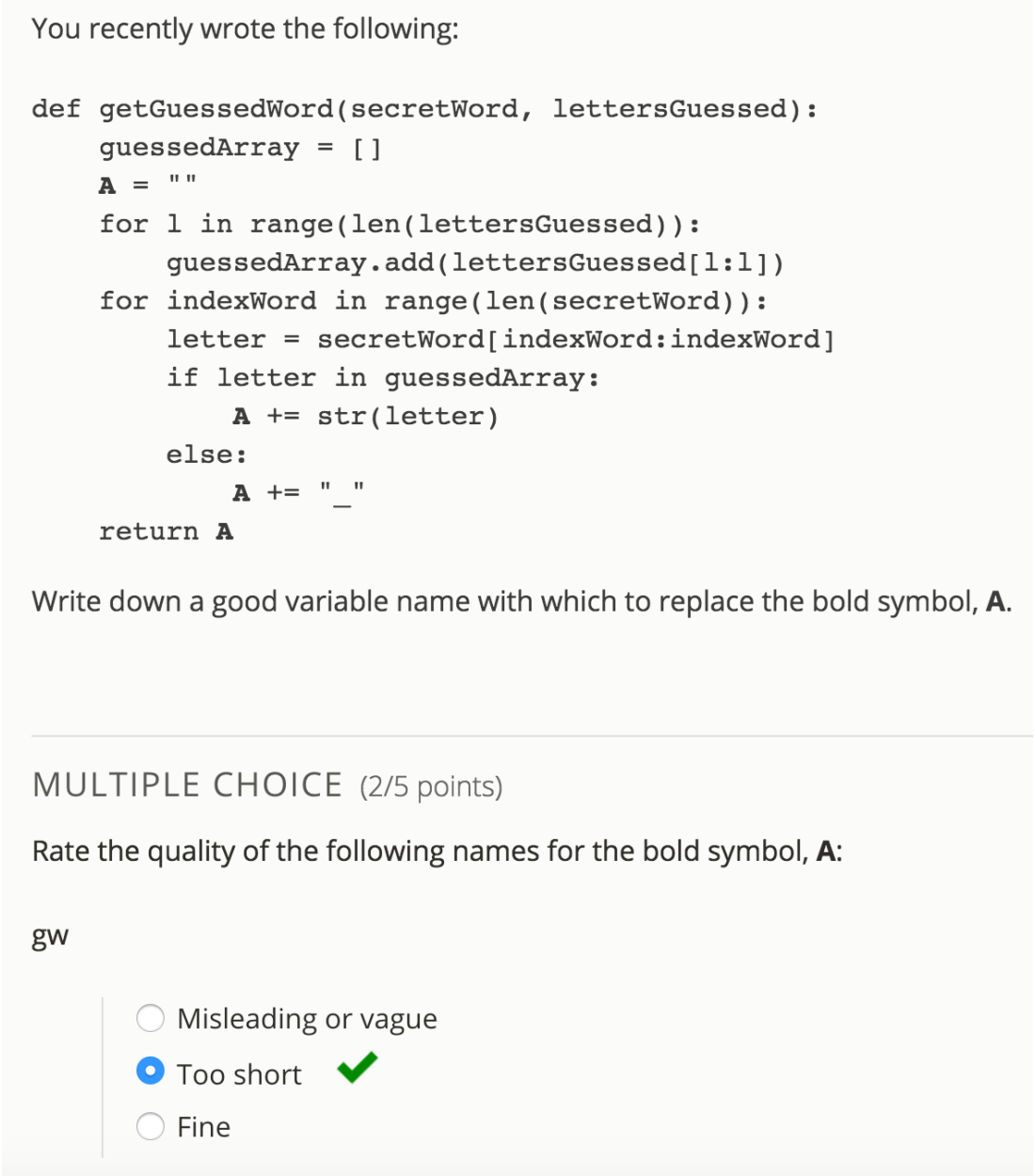


Figure 4-1: A personalized quiz as seen by the student, delivered by an edX-based system maintained by a large university. Students are shown their own code, with a variable name replaced by an arbitrary symbol, followed by variable names for the student to consider and label using the same labels that were available to the teacher. After the student has submitted their own judgments, the teacher's labels are revealed, along with their explanatory comments.

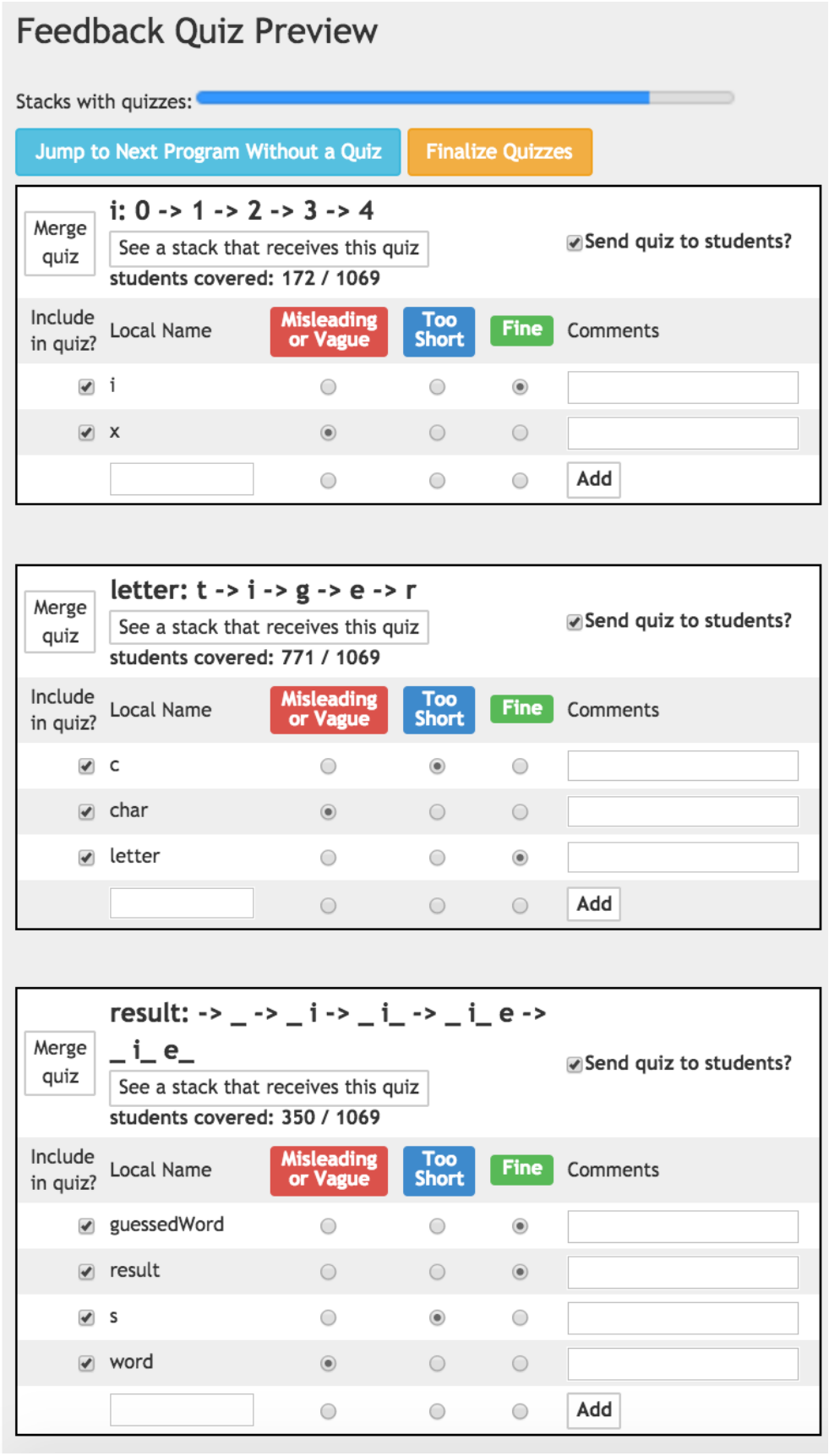


Figure 4-2: The quiz preview pane of the Foobaz teacher interface. Variable behavior was logged by running all solutions on a common test case. This particular teacher created quizzes for the common variable `i` that iterates through indices of a list, the common

against the teacher's. If a student's solution includes a particular common variable, then that student can receive a personalized version of the quiz about that variable.

As teachers rate variables by attaching labels to them, quizzes are created with these names as their good and bad examples. Using the Toggle Quiz Preview button, teachers can see a preview of the quizzes (Figure 4-2) alongside the scrollable list of stacks and watch the quizzes grow as they rate more names. They can hide the quiz preview to reduce visual clutter while they explore all the stacks, common variables, and interesting alternative names.

If two different common variables perform the same conceptual role in student solutions but do not go through the exact same sequence of values, then the teacher can use the "Merge" button to combine the quizzes about each common variable into a single quiz. This quiz becomes relevant to students who have either common variable in their own solution and can be sent out to both groups.

Ultimately, the teachers' goal is to provide pedagogically valuable personalized quizzes to as many of the hundreds or thousands of students in the course as possible. Analogous to the progress bar for variable names, the quiz preview pane includes a progress bar for the number of stacks of solutions that will receive at least one quiz. Like the previously discussed button for selecting the next most popular unlabeled name, the quiz preview pane also includes a button for jumping to the next largest stack of student solutions that do not yet have any quizzes. If the teacher deems one of their automatically populated quizzes to be not pedagogically valuable, then they can uncheck the option to send that quiz back to students. To provide more illustrative examples that might not have been produced in student solutions, teachers can add their own custom good and bad variable name examples and write explanations in the comment field associated with each alternative name.

4.2 Evaluation

We evaluate Foobaz's teacher and student-facing interfaces with two consecutive user studies, one for each population. In order to evaluate Foobaz's scalability, the solutions seen by teachers were collected from MOOCs with thousands of students and a residential college

class of several hundred students.

4.2.1 Datasets

We evaluated Foobaz on sets of correct solutions to four different programming exercises, ranging in size from a couple hundred to several thousand solutions, collected from 6.00x, an introductory programming course in Python that was offered on edX in the fall of 2012, and 6.0001, a residential introductory programming course in Python offered at MIT in the fall of 2014 (see Figure 4-3). The four exercises, referred to here as `iterPower`, `hangman`, `dotProduct`, and `computeDerivative`, are representative of typical exercises that students solve in the early weeks of an introductory programming course. They have varying levels of complexity and ask students to perform loop computation over three fundamental Python data types, integers, strings, and lists.

4.2.2 Teacher Study

During the initial briefing, teachers were informed that they would be looking at solutions that had already passed an autograder and instructed to focus only on variable names, ignoring other aspects of code style, structure, and correctness. Teachers were invited to look over a page in a browser with all solutions concatenated in a random order into a flat list of boxed, syntax-highlighted code. We chose this design as our baseline to emulate existing methods of reviewing student functions.

Using this baseline interface, teachers were asked first to rate as many good and bad variable names as possible, with an eye toward maximizing coverage of names (Task Part 1). Next, the teachers were shown an example of a quiz and were asked to compose their own by listing variable names and labeling them as good or bad with whatever short descriptors and explanatory comments they wished (Task Part 2). Participants were given 5 minutes for each task, and then asked to fill out a survey about their experience. (The answers to two of these surveys were lost so we only report survey results from 8 of the 10 participants.)

Participants learned about the Foobaz interface by watching a tutorial video. This train-

ing process took between 10 and 15 minutes depending on the dataset shown in the video. Participants were encouraged to hold their questions to the end, and answer them by interacting with the interface.

Participants performed both Task Part 1 and 2 on a third dataset of solutions in the Foobaz teacher interface. Participants were asked to spend 5 minutes to perform each task, though some decided to spend more time. They filled out the same surveys about their experience again, followed by a final survey about which features of the Foobaz interface they found helpful.

Apparatus

In all sessions, we used a laptop with a 15.4-inch 2880x1800 pixel Retina screen. All participants' interactions with the system were logged with timestamps using Meteor collections.

Participants

We recruited 10 participants (6 female) with ages between 20 and 29 ($\mu = 23.1$, $\sigma = 2.7$) through computer science-specific and campus-specific mailing lists and Facebook groups. All participants self-reported that they had been a grader, lab assistant, or teaching assistant for a Python course.

Results

Problems with Baseline. When asked to comment on good and bad variable names based on the baseline interface, most teachers immediately began scrolling through solutions one by one, taking notes as they went, fully aware that they would only be able to skim a small, random fraction of the total number of solutions. The sheer volume of solutions was overwhelming to some.

Results of the post-baseline survey reinforce critical usability issues with the status quo that Foobaz was designed to address. In these survey responses, teachers expressed an appreciation for the simplicity, readability, and searchability of the baseline interface

but wished that the endless stream of often very similar solutions could be summarized or "de-duped" before they had to read through them. One teacher requested that variables be automatically identified, so that all references to a variable can be highlighted. This may have been a direct consequence of the fact that it was not possible to search for all the occurrences of the variable name `i` without the browser also highlighting all the `i`'s within the rest of the names and keywords, e.g., the `i` in "if." Another teacher requested an automated count of common variable names. These teachers anticipated three critical features of the Foobaz interface: deduplication, variable name counts, and highlighting all occurrences of selected names.

Two teachers commented on the importance of understanding the role a variable takes on within the program. One teacher writes, "Many times the variable names meant something but I still had to read the code to make sure that it meant what I thought it meant in the context of the code." The second teacher observed, "Whether a variable name is good or bad depends a lot on its function within the code, and since each code block has a somewhat unique structure, I felt like I should be creating separate categories for good vs. bad variables names, e.g., 'for the derivative result,' 'for a counter in a loop through poly,' etc." This is exactly what the Foobaz interface is designed to support.

Power-law Distribution of Names The approximately power-law-type distribution of stacks of code from the thousands of edX solutions has already been reported in prior work, such as Figure 12 in [42]. In Foobaz, the distribution of unique combinations of variable names and behaviors does not differ significantly from a power-law distribution; the Kolmogorov-Smirnov test for a difference between the best-fitting power-law distribution and each dataset all gave p-values of at least 0.44 (i.e., not significantly different), with the best-fitting exponent of the distribution between 1.79 and 2.13. Within each stack, the names for any particular variable appear to follow the same distribution.

There is no ground truth for which variable names are "bad," but we can report the counts of variables that the users chose to label and the counts of unique names for the top common variables in each problem. In the 3875 `iterPower` solutions, there were 179 different names given to a variable representing the base being exponentiated and 64 different names for a variable representing the exponent. In the 1393 `computeDerivative` solutions,

there were 176 different names given for the variable containing the result and 39 different names for the most commonly used iterator variable. In the 1118 `hangman` solutions, there were 50 names given to the variable that iteratively takes on the characters of the 'secret word' input argument and 99 names given to the variable containing the string most commonly returned by solutions as the answer.

Figure 4-4 shows the fraction of names that subjects labeled in Study 1, and the distribution of those names across various categories of quality. In spite of the unknown underlying distribution of good and bad variable names, the subjects are finding and labeling variables across all available categories in order to make quizzes.

Efficiency of Foobaz. The efficiency of using Foobaz to create feedback came up repeatedly in teachers' survey responses. Teachers appreciated the feeling of doing the task "at scale." One teacher noted that it felt like, "with each action, I was helping a large number of students" without "repeating or wasting effort too much." While using the button that highlights and scrolls the next most popular, untagged variable name into view, a teacher told the experimenter, "I love this button!" The arrow key-based navigation through tables of variable names was appreciated as well. At least one teacher commented that "seeing variable names grouped by their role made the process much more efficient."

The efficiency gains of the interface were hampered by, at times, a noticeably sluggish interface response time to some queries that scaled with the size of the dataset. This sluggishness can be cut down by reducing the many unnecessarily repeated calculations made in the current implementation.

Quiz Composition. In both interfaces, teachers appreciated the potential pedagogical value of making quizzes: "I âĂę liked that with the [quiz] I made, students can actually learn about good alternatives. ... [T]hey can change their variable names after putting extra thought into it." A teacher expressed appreciation that, using the Foobaz interface, they could generate quizzes based on actual submitted code as well as their own comments.

Coverage. Immediately after using the baseline interface, teachers did not strongly agree or disagree with the statement "I was able to give specific, personalized feedback to many students" ($\mu = 3$, $\sigma = 1.4$ on a 7-point Likert scale with 1: strong disagreement and 7: strong agreement). Teachers slightly disagreed with the statement "I saw a large percentage

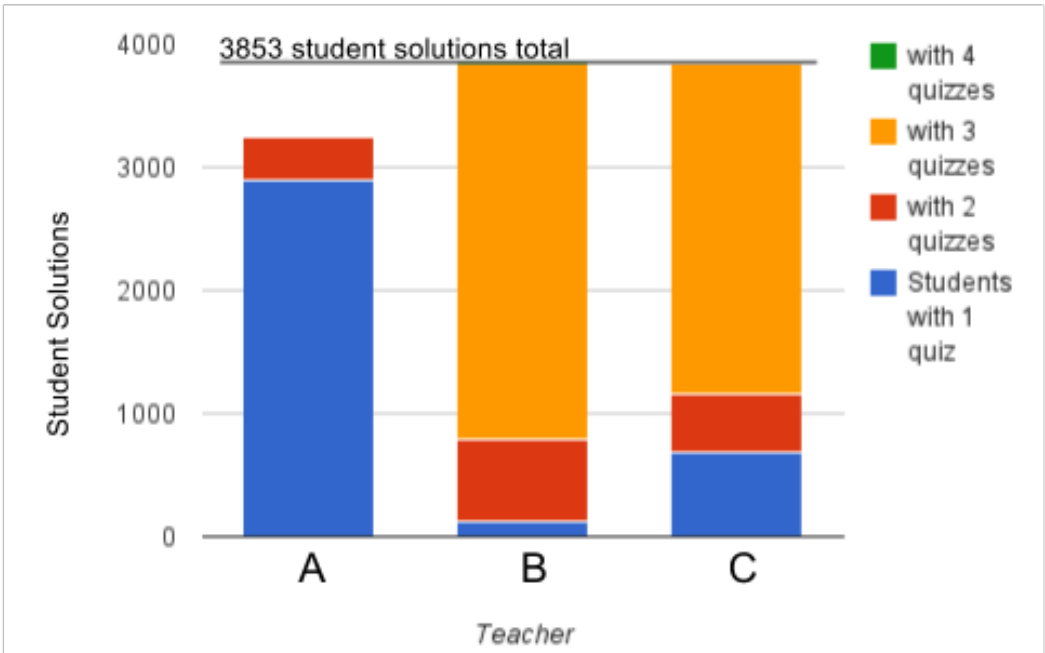
of these students' variable names" ($\mu = 2.6$, $\sigma = 1.2$) and slightly agreed with the statement "This interface helped me provide feedback to many students" ($\mu = 3.6$, $\sigma = 2.0$). After using the Foobaz interface, the mean level of agreement with these statements jumped to 5.5 ($\sigma = 1.7$), 6.3 ($\sigma = 0.46$), and 6.6 ($\sigma = 0.5$).

Figure 4-5 shows that most solutions in the edX datasets received at least one or two quizzes. Solutions in Figure 4-5 have multiple common variables on which they could potentially be quizzed (3, 3, and 4 on average in subfigures (a), (b), and (c), respectively). One teacher used Foobaz to create quizzes for a much smaller dataset, collected from the residential class of only several hundred students. They achieved a similar percentage of coverage (87% of student solutions received at least one quiz) in a similar amount of time, showing that the Foobaz workflow and output appears relatively invariant to the size of the dataset. Figure 4-6 illustrates that, within minutes of their first interaction with the system, teachers can label a significant portion of student-chosen variable names in a dataset, though their progress trails off as they encounter the long tail of names for particular roles and transition to creating better quizzes.

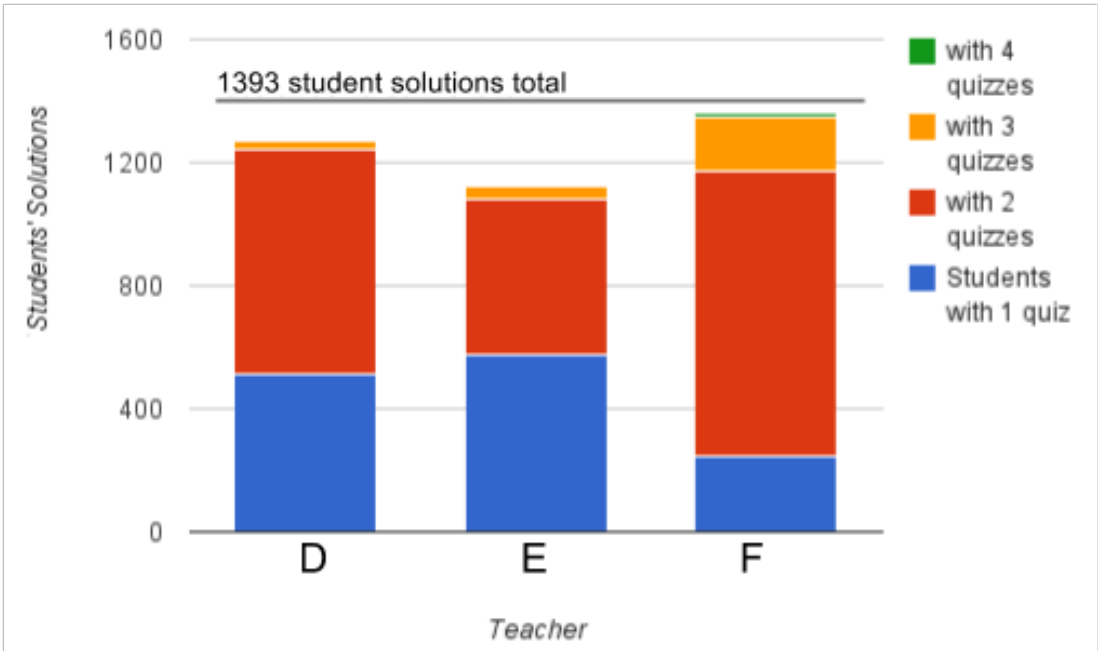
Combined with Figure 4-4, Figure 4-5 also shows that, by only labeling approximately 20 variable names in each of the edX datasets with thousands of student solutions, teachers cover at least 85% of the class with personalized feedback quizzes. Even though Figure 4-6 shows that the coverage of individual variable names with feedback is high, it matters less for Foobaz than in powergrading systems. What matters more is the coverage of students with quizzes that have excellent examples of good and bad variable names students would not otherwise get to see and learn from.

4.2.3 Student Study

We ran a second study on the student side of the workflow in order to (1) find out if the teachers' efforts in the previous study produces quizzes that are relevant to these new students and (2) better understand student reactions to this novel form of feedback. In order to do this, we targeted beginner programming students and invited them into the lab to receive personalized quizzes generated by the teachers in our previous study.



(a) Quiz coverage for student solutions to the `iterPower` exercise



(b) Quiz coverage for student solutions to the `computeDerivative` exercise

Before the start of the study, quizzes composed by teachers using Foobaz in the first user study were rendered using the edX framework, ready to be personalized. Since pilot testing with beginner programming students indicated that six alternative variable names in a quiz is too many, teachers' quizzes were randomly subsampled to include a maximum of five alternative names for students to consider.

Students came into the lab for one hour and composed solutions to one of the four exercises. After receiving each solution, the experimenter mentally executed the solution and compared the behavior of its variables to the variable behavior covered by the teachers' quizzes. If there was a match to one or more teachers' quizzes, one quiz was randomly selected. The experimenter made a copy of the student's code and replaced every instance of the variable to be quizzed on by an arbitrary symbol, e.g., a bold letter A. The experimenter then appended the quiz to the modified copy of the student's solution and delivered it to the student as a personalized quiz. If there was no match to one or more teachers' quizzes, then the student received a generic quiz, about a variable name in a solution other than their own.

After the student completed their personalized quizzes, they took a survey about their experience. Students who were able to complete the coding exercise and quizzes with significant time left in their session repeated this process for a second programming assignment.

Apparatus

In all sessions, we used laptops with 15.4-inch or 13.3-inch screens. All participants' interactions with the system were logged using the edX platform infrastructure.

Participants

We recruited 6 participants (4 female) who were either undergraduate or graduate students, through computer science-specific and campus-specific mailing lists, Facebook groups, and word of mouth advertising. Their ages were between 18 and 27 ($\mu = 20.4$, $\sigma = 3.2$). Four of the participants had taken one or two introductory programming courses on Coursera or at their high school or college campus. The remaining two participants had taken three or

four classes that involved learning programming languages or computer science concepts, and had some experience with Python.

Results

Six students took a total of 12 quizzes, 11 of which were able to be personalized, even though their solutions were, in some cases, significantly different from the prior student solutions seen by teachers during quiz creation. Correspondingly, in the surveys that followed, students agreed with the statement "This quiz felt relevant to me" at an average level of 5.4 ($\sigma = 1.0$) on a 7-point scale. One student's solution did not receive a personalized quiz because its variables behaved in ways that no teacher in the previous study considered. That student was able to understand the new solution and take the quiz, though it had little relation to their own solution.

Students were asked in the post-quiz surveys about what they learned from the exercise. One student replied, "Possible variable names are pretty much synonyms, but the more detailed/specific ones are better." Another wrote, "It's worthwhile to pick good variable names." Students' average level of agreement with the statement "The quiz made me think about what makes variable names good or bad" was 6.2 ($\sigma = 0.9$) on a 7-point scale. Mean levels of agreement with statements about the quizzes being confusing or tedious were 2.9 ($\sigma = 1.5$) and 3.4 ($\sigma = 1.2$), respectively, on a 7-point scale.

Some students observed that this was a very subjective quality of their code to be quizzed on, but all students were able to understand and complete the quizzes they were assigned. Some students disagreed with the instructor-provided ratings. When this occurred and the teacher left no explanation, students had at least one of three reactions: (1) They tried to imagine what the teacher was thinking. (2) They expressed displeasure at the lack of explanation. (3) They decided that they still disagreed with the teacher's judgment. Students did pick up on the teachers' preferred naming conventions through the quizzes. As evidence for this, two students correctly wondered aloud whether a different teacher made each of the quizzes they saw during their session. Given the subjectivity of the task, it may be necessary to grade only on participation, rather than absolute agreement with the teacher.

Students were not informed that quizzes were populated largely by fellow student variable names, but one student volunteered their appreciation for a "wide spectrum" of variable names to consider. However, randomly sampling teachers' quizzes down to 5 variable names created some student confusion when there were no positive naming examples in the resulting quiz. This is evidence that sampling should be constrained to include both good and bad examples and the user interface should provide some additional guidance to remind teachers that students highly value a balance of examples, each paired with an explanatory comment.

4.3 Limitations

The first study establishes the usability, learnability, and efficacy of the main Foobaz interface for teachers. The second study is intended to show that students can understand and use the personalized quizzes that Foobaz produces. However, this evaluation of the student experience does not yet show pedagogical benefit. To measure learning benefits, we plan to deploy the tool in a large Python programming course this fall.

4.4 Conclusion

We have designed and studied both the teacher and student sides of a novel interface and workflow for providing feedback on student variable names at scale. We hope it will serve as an example and a design pattern for future work on user interfaces for teaching programming to thousands of students at once.

Chapter 5

Learnersourcing Debugging and Design Hints

One-on-one human tutoring is a costly gold standard in education. As established in Bloom's seminal work on tutoring, mastery-based instruction with corrective feedback can offer a substantial improvement in learning outcomes over conventional classroom teaching [**?**]. However, personalized support does not scale well with the number of students enrolled. In large classes, it is often not feasible for students to get personalized hints from a teacher in a timely manner. Massive open online courses (MOOCs) can have even worse teacher-to-student ratios, by orders of magnitude. Intelligent tutoring systems have strived to simulate the type of personalized support received in one-on-one tutoring, but they are expensive and time-consuming to build.

In this chapter, learners are asked to generate personalized hints for each other. Prior work on *learnersourcing* demonstrates how learners can collectively generate educational content for future learners, such as video outlines and exam questions, while engaging in a meaningful learning experience themselves [61, 85, 126]. The proposed benefit of learnersourcing is that learners are not only more intrinsically motivated to engage with the learning content to begin with, but may also benefit pedagogically from the task itself.

This work builds upon learnersourcing by exploring how it can be applied to the generation of personalized hints during more complex problem solving. Whereas prior work determined which task to present to the learner depending on what information was still

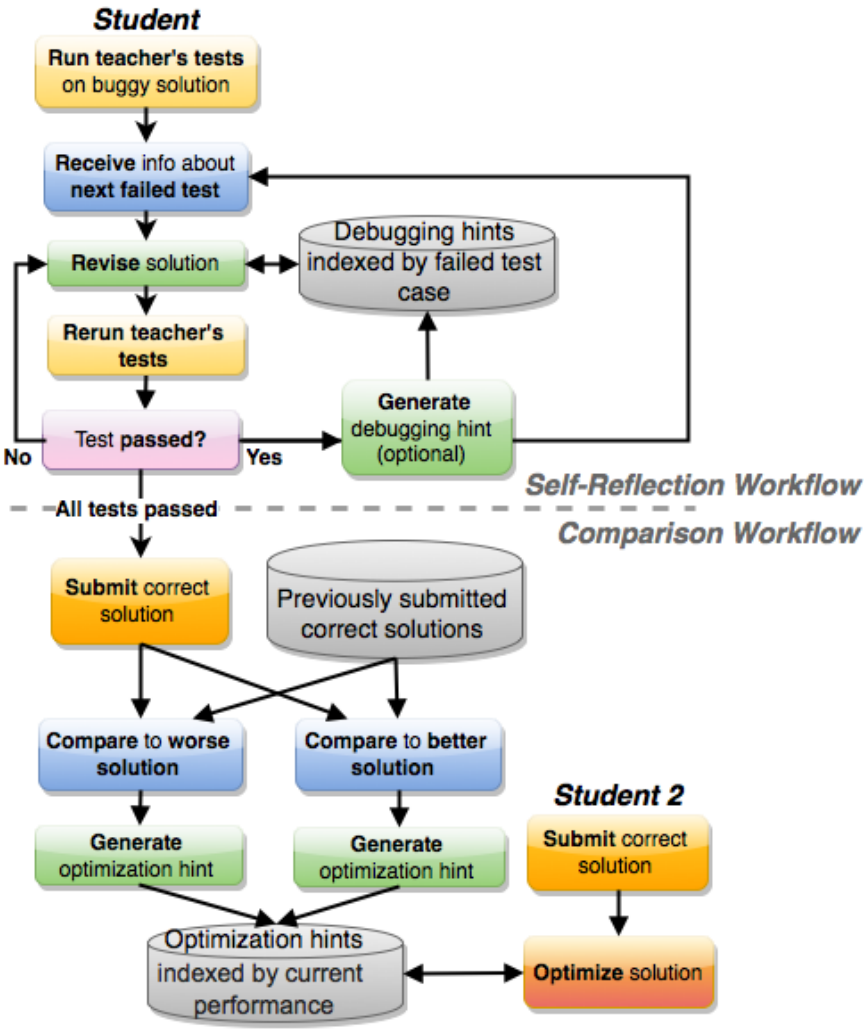


Figure 5-1: In the *self-reflection* workflow, students generate hints by reflecting on an obstacle they themselves have recently overcome. In the *comparison* workflow, students compare their own solutions to those of other students, generating a hint as a byproduct of explaining how one might get from one solution to the other.

needed [126], many educational topics like digital circuit design require more domain expertise, raising the question of which learners should be assigned to generate which hints. Beyond learnersourcing subgoals for how-to videos, this work takes on the challenge of generating content that is tailored to both the *hint-receiver*'s current progress and the *hint-author*'s likely level of understanding.

This chapter describes two workflows for learnersourcing hints that assign hint-generating tasks to learners based on what problems the learner has recently solved. In the *self-reflection* workflow, students generate hints by reflecting on an obstacle they themselves have recently overcome. In the *comparison* workflow, students compare their own solutions to those of other students, generating a hint as a byproduct of explaining how one might get from one solution to the other. The second workflow can operate on the output of the first, as shown in Figure 5-1. In both workflows, the key insight is that, through their own experience struggling with a particular problem, learners can become experts on the particular optimizations they implement and bugs they resolve. The workflows can take pressure off teaching staff while giving students the valuable educational experiences of reflection and generating explanations.

While such workflows could have many applications, this paper presents a specific application within a college-level computer architecture class. In this course, students implement, debug, and optimize simulated processors by constructing digital circuits. During both the debugging and optimization process, hints are one mechanism for teachers to help students fix and optimize their circuits. This paper applies our learnersourcing workflows to two kinds of hints: *debugging hints* and *optimization hints*. A debugging hint is a student's attempt to help a future student change their solution so it generates the expected output for a particular input. An optimization hint is a student's attempt to help a future student get from one correct solution to another, more optimal solution. When hint-receivers encounter that particular situation during their problem-solving process, the hints can be shown to them as if they are the personalized hints an intelligent tutoring system might generate, or that a teacher might provide during a one-on-one interaction.

5.1 System Design

There are a number of necessary decisions to make when designing interventions to collect and deliver hints:

When should learners be asked to provide hints? As soon as a student has resolved a bug, they may have some expertise about that bug that they can share with other students. If too much time has passed between resolving the bug and writing a hint, the student may forget necessary details and context, or forget the bug altogether. A previous learnersourcing system also prompted students to contribute content immediately after having experienced it themselves, for similar reasons [126].

How should learners access hints? Hints can be distributed using either a *push* or *pull* model, and can involve displaying either *all* or *some* of the hints. For example, a push model might display hints as a constantly updating resource, whereas a pull model could dispense hints to individual students just-in-time, when a student needs help. The hints could be algorithmically selected based on the student's work so far and the hints they have already received. I explored both push and pull models, using the push-all model for distributing debugging hints and the pull-some model for distributing optimization hints. Generating optimization hints was a required reflection activity, so the volume and redundancy of these hints made a push-all model potentially overwhelming.

What hints can I ask, or allow, a student to generate? In cases where the student's *start state* (prior to overcoming an obstacle) and *end state* (after overcoming an obstacle) are known, such as when a student fixes a bug, the system can ask the student to create a hint helping other students encountering a similar bug or start state. For example, in the case of circuit design, I consider a student who has recently fixed a bug resolving a particular verification error to be capable of writing a debugging hint associated with that verification error.

In many cases, however, a student might not face any explicit obstacle, or their start state may not be known. For example, a student might naturally arrive at a highly optimal circuit design without having first tried a less optimal design. Regardless of the path to their solution, the student could generate hints by comparing their own solution to a more

optimal solution, or to a less optimal solution. In this chapter, I explore both of these directions by asking each student to do both comparisons. To keep hint generation relevant to the learner's current task and to minimize cognitive load, I did not ask students to generate hints between pairs of solutions when they were familiar with neither solution.

How should hints be indexed? Indexing hints by a meaningful feature of student solutions allows students to more easily find relevant hints in a push model of hint distribution and allows the system to deliver more relevant hints to each student in a pull model of hint distribution. Optimization hints could be indexed by the learner's start state, end state, or both with respect to performance. Debugging hints could be indexed by verification errors.

During debugging, students' solutions are run through sequences of teacher-designed test inputs. A *verification error* occurs when a student's solution does not return the expected values. Because test cases have a specification of actual and expected outputs for each input, I decided to index debugging hints by the tests for which the solution deviates from the expected output. In other words, debugging hints are associated with the verification error that disappears if the bug is resolved.

During optimization, the goal is not simply to attain a correct solution, but rather to arrive at a more optimal correct solution. I decided to index optimization hints by both start and end states: the leap from a less optimal solution to a more optimal solution that the hint is intended to inspire. These states, the solutions themselves, are complex circuit objects; I use the number of transistors in a solution as a metric of its optimality. In this indexing scheme, all hints written with the intent of helping a student with a 114-transistor solution create a 96-transistor solution are binned together.

Which hints should a student receive? In the push-all model of hint distribution, this question is not relevant, but in the pull-some model of hint distribution, it may be critical. Ideally, students would receive a progression of increasingly specific hints, following patterns of adaptive scaffolding established in intelligent tutoring system literature [121], helping them reach a more correct or optimal solution on the spectrum.

This ideal still leaves room for design choices about exactly which hints to deliver in a pull model of distribution. For example, during the optimization stage of a student's assignment, the system could *always* give the student hints that help them create the *next*

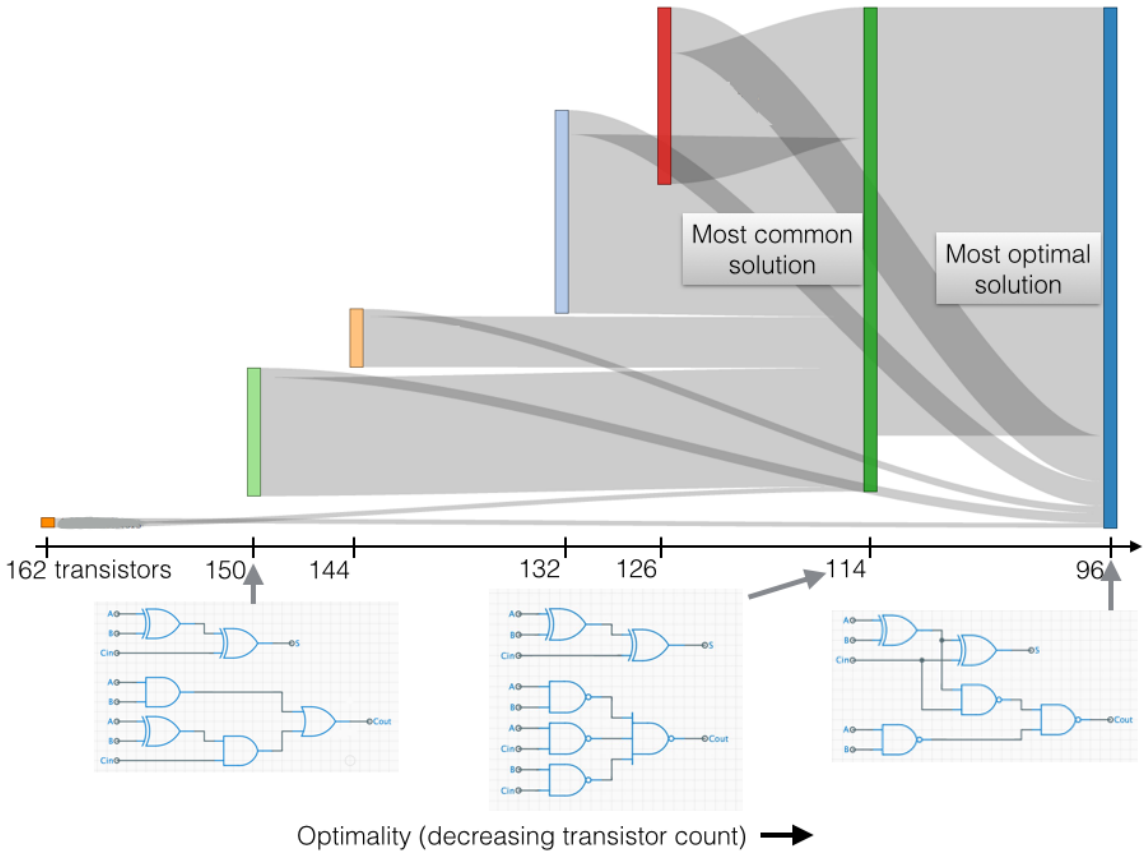


Figure 5-2: Sankey diagram of hints composed between types of correct solutions, binned by the number of transistors they contain. The optimal solution has only 21 gates and 96 transistors while the most common solution generated by students has 24 gates and 114 transistors.

optimal solution found by other students (see Figure 5-2). This would hopefully be an optimization challenge that falls within the student's zone of proximal development [123]. However, this may be too cautious. If a student's solution is far less optimal than the *most common solution*, then the system could give the student hints to help them leap directly to that most common solution, without first creating any intermediate solutions. This strategy ignores how large a leap outside their zone of proximal development this might be, but it ensures that the student is exposed to the ideas, presented in those hints, that are necessary for implementing a solution at least as optimal as the most common one. I chose this latter option, both hinting students toward the most common solution and hinting students with the most common solution toward the most optimal solution.

Should hint creation be a required task? As discussed in Related Work, generating

hints can be a valuable part of the learning process. I required all students to generate optimization hints as part of a reflection activity immediately after submitting their first correct circuit. I did not require students to generate debugging hints. This is because the number of bugs encountered could be large, and unlike optimizations, many bugs also may not lead to significant conceptual gain upon reflection. Because debugging hints are immediately pushed out to all students, I keep both hint creation and upvotes voluntary to minimize the signal-to-noise ratio in hint quality.

How should the variation in hint quality be handled? In the push model of hint distribution, I used users' upvotes to sort hints by quality. In the pull model of hint distribution, I took advantage of the redundancy of the hints, and presented five hints at once. If one or more redundant hints were of poor quality, their aggregate message might still be helpful for a student. If most of the hints were about a feature that is irrelevant to the student receiving the hints, the remaining hint(s) might be about something different and more relevant. I limited each set of hints to a size of five to avoid overwhelming the learner with too many hints.

How public should the hint-author be? Many systems for question-answering have a reputation system, where the author is known and recognized for contributing answers. Previous work at CSCW has examined whether reputation would improve class forum participation [23]. For simplicity, I chose to leave student identities hidden.

Final Workflows In the self-reflection workflow, students iteratively modify their solutions to pass as many teacher-created tests as possible. For any verification error revealed by those tests, students can look up hints for what modifications might cause their solution to pass that test case instead. The hints are stored in a database indexed by the verification errors they are intended to address. When students fix a bug in their own solution, they can reflect on their fix and contribute a hint to the database for others struggling with the same verification error. The self-reflection step turns a successful bug fix into a shareable hint. It is effectively a self-explanation exercise, since the hint is not a conversation starter; it is meant to stand alone for any future student to consult. As studied in prior work [20], self-explanation helps students integrate new information into existing knowledge.

In the comparison workflow, students compare their correct solution to alternative cor-

rect solutions previously submitted by other students or teachers. They are prompted to compare their solution to a solution *W* with worse performance and generate an optimization hint for students who have solutions like *W*. They are then prompted to compare their solution to a solution *B* with better performance and generate an optimization hint for students who have solutions like their own, which are not yet as performant as solutions like *B*. In each case, the student is generating an optimization hint to help students increase the performance of their solutions. Students who receive these hints use them as guidance while optimizing their own solutions. Figure 5-1 illustrates both workflows.

5.2 User Interfaces

I designed two user interfaces, one for each workflow. To learnersource debugging hints with the self-reflection workflow, my co-authors and I built and deployed *Dear Beta*, a Meteor web application that serves as a central repository of debugging advice for and by students in the class. The name alludes to both the "Dear Abby" advice column and the Beta processor that students create in the class. To learnersource optimization hints with the comparison workflow, I designed *Dear Gamma*, a web interface students were required to fill out as a reflection activity after submitting their final correct circuit for a class assignment.

5.2.1 Dear Beta

I applied the self-reflection workflow to processor debugging. Consider students working on their Beta processors within the digital circuit development environment provided by the computer architecture class. Students run a staff-designed test file *x* on their circuit. The development environment alerts them to a verification error: for a particular input (test number *n*), *y* was the expected output and the student's circuit returned *z*.

Students eliminate verification errors by fixing bugs in their circuits. They may use trial and error, methodically examine internal simulated voltages, or experience a flash of insight. On the Dear Beta website, they can post a hint for others derived from their insight and indexed by the verification error it caused. In the process of creating a hint, students

have a chance to reflect on their own process of resolving the error. Other students encountering that verification error can look up these hints, upvote helpful hints, and contribute new hints. Hints for each error are sorted by the number of upvotes they receive.

After further examining their malfunctioning processor with no success, some students may open the Dear Beta website in order to get help (Figure 5-3). Dear Beta displays all errors and hints, sorted by test file name x, then test number n. The student can either scroll to find hints for their verification error or jump directly to them by entering x and n into the bar pinned to the top of the page. If the error was not yet in the Dear Beta system, the error will be added and scrolled into view.

If there are no hints yet, or if the hints are unhelpful, the student can click the "Request" button to the left of the error, which increments a counter. This button helps communicate the need for hints for a particular verification error to potential hint writers. This is analogous to the "Want Answers" button and counter on Quora, a popular question-and-answer site.

If the student resolves their verification error and feels that the existing hints were insufficient or incomplete, they can click in the text box labeled "Add a hint!" so that it expands into a larger textbox sufficient for typing out a paragraph of their own hint text (Figure 5-4). Their new hint may be a clearer rephrase of an existing hint or hint at yet another way to resolve the verification error. Given the variety of processor designs and implementations, there may be several ways any given verification error may be thrown.

Teacher Feedback on Early Prototypes of Dear Beta

finish removing we's

After deploying initial prototypes of Dear Beta for two semesters, we invited Teaching Assistants to share their complaints, requests, and experiences with us. Four TAs were interviewed, in person or by email, and their feedback and experiences informed Dear Beta's final design.

Both TA_1 and TA_3 adapted to Dear Beta's deployment by first asking each help-seeking student if they had already consulted Dear Beta. If they had not, TA_1 came back to them after visiting everyone else in the lab help queue. By then, they had often already resolved

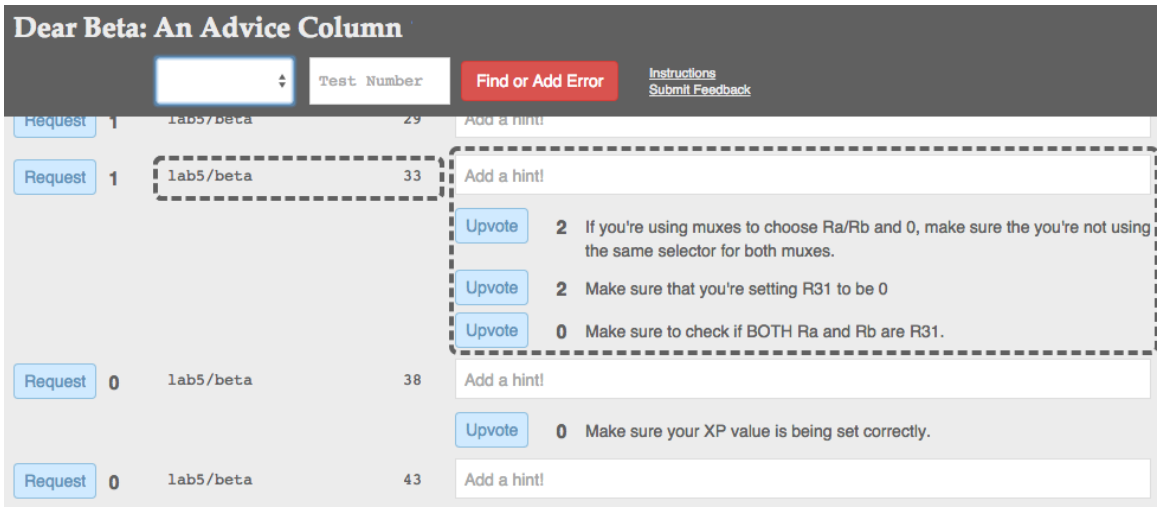


Figure 5-3: *Dear Beta* serves as a central repository of debugging advice for and by students, indexed by verification errors. In this figure, there are three learnersourced hints, sorted by upvotes, for a verification error on test no. 33 in the 'lab5/beta' checkoff file.

Figure 5-4: After fixing a bug, students can add a hint for others, addressing what mistake prevented their own solution from passing this particular verification test.

their problem with Dear Beta's hints, and had a new bug they wanted help debugging.

Dear Beta was used as a debugging aid for both students and teachers. TA_2 described Dear Beta as a "starting point" for students, many of whom used it diligently during debugging. TA_2 appreciated that students who did ask for her help no longer said, "My Beta isn't working. Tell me why." Instead, they used Dear Beta as a starting point, to help them identify potential locations of a bug in many pages of code. Not just helpful for students, TA_3 was able to describe with specific examples how Dear Beta *helped him* help students quickly resolve common bugs.

TA_2 wondered if the extra hints were making it too easy to complete the lab, possibly letting students pass without understanding. TA_3 echoed this concern, but he made sure each student actually understood the Dear Beta hints whenever he personally guided them through the debugging process.

TAs identified both strengths and weakness in Dear Beta's design. TA_1 strongly supported Dear Beta's existing design as a single scannable sorted list for quickly finding hints, rather than a purely search-based hint retrieval mechanism or the more general class forum. However, the affordances for contributing new hints in the initial prototype were not obvious and rarely visible on small screens. As a result, TA_2 was concerned that the level of student involvement in producing hints might be too low. The final Dear Beta design is more externally consistent with other participatory Q&A systems, has more salient buttons for contributing new hints, and a responsive design that accommodates screens as small as that of a cell phone.

TA_4 was absent during most of Dear Beta's deployment but still regularly recommended Dear Beta to students who asked for her help over email. A fifth TA declined to be interviewed; she felt that she had not interacted with Dear Beta enough.

5.2.2 Dear Gamma

In order to learnersource optimization hints, we caught students at a different stage in their learning process: right after they passed all verification tests for a particular digital circuit, the Full Adder. Because students may have arrived at their solution without encounter-

ing any particular optimization obstacles, Dear Gamma uses the comparison workflow for learnersourcing rather than the self-reflection workflow.

The comparison workflow is modified slightly, to accommodate the requirements of the course lecturer, who wanted to make sure that all students get a chance to consider both the most common and the most optimal solutions. The collection of previous student solutions in Figure 5-1 was also curated by the lecturer. If a student's solution is larger than the most common solution, they are not shown solutions larger than their own; instead, they are asked to consider both the most common and the most optimal solutions, so they benefit from seeing both without doing extra work overall. Students with the most optimal solution only consider alternative solutions that are worse than theirs. Figure 5-5 shows an example of the page for a student with a 114-transistor solution.

In this activity, students are given a pair of solutions and asked to give a hint to future students about how to improve from the less optimal solution to the more optimal solution. Students write hints for two such pairs of solutions. In each pair, one of the solutions is always their own. When the student's own solution is the better solution in the pair, then the student can hint at what the peer might have missed. For example, *Remember DeMorgan's Law: you could replace the 'OR' of 'ANDs' with a 'NAND' of 'NANDs.'* When the students' own solution is the poorer solution in the pair, they are challenged to first understand how the better solution uses fewer transistors, and then write a hint about the insight for a peer. To aid the student in comparing solutions, the Dear Gamma interface displays the student's own solution as a reminder of their design, as well as an alternative picked from among other students' solutions.

Specifically, if a student's solution S is just as or more optimal than the most common solution, they are asked to (1) write a hint to help a future student with a less optimal solution reach solution S and (2) write a hint to help a student with solution S reach the most optimal solution. If a student's solution S is less optimal than the most common solution, they are asked to write a hint to help a student with solution S reach (1) the most common solution and (2) the most optimal solution. This scheme ensures that all students are familiar with the most common solution and the most optimal solution and have written two hints to help future students improve the optimality of their solutions.

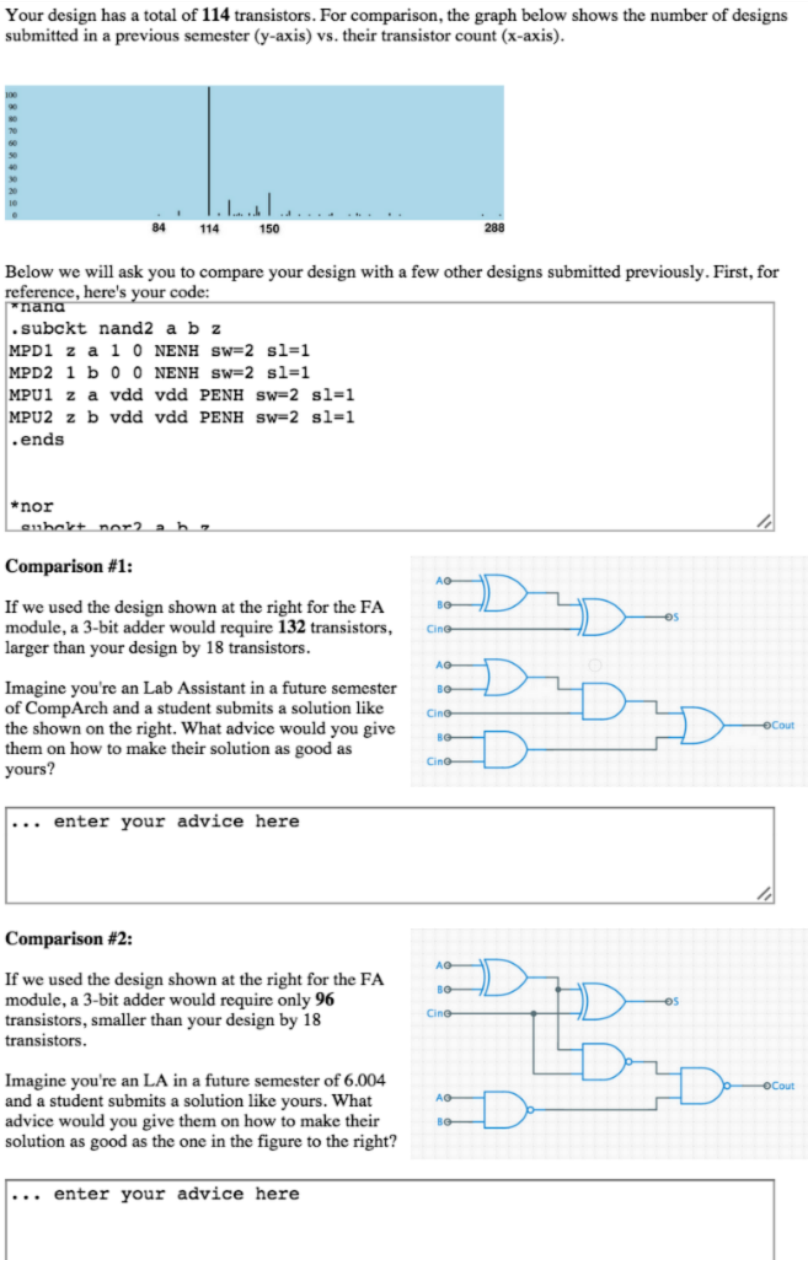


Figure 5-5: This is the Dear Gamma interface for a student with a solution containing 114 transistors. In the first comparison, they are asked to write a hint for a future student with a larger (less optimal) correct solution. In the second comparison, they are asked to write a hint for a future student with a solution similar to their own so that they may reach the smallest (most optimal) correct solution.

5.3 Evaluation

To evaluate the extent to which learnersourced hints can support problem solving, we deployed Dear Beta and Dear Gamma to the computer architecture class, which had an enrollment of more than two hundred students. Dear Beta was deployed for 6 weeks, during which we collected student-generated debugging hints and observed the simultaneous usage of those hints in a real-world setting. Dear Gamma's optimization hint collection interface was released to students as part of a particular lab. We then conducted a lab study with nine students to understand how they solve a typical engineering problem using these learnersourced optimization hints.

The questions our evaluation sought to answer are: (1) *What are the characteristics of student-generated hints?* and (2) *Can learners solve problems using those hints?*

5.4 Dear Beta

The Dear Beta website was released as a stand-alone additional resource for students one week prior to the due date for the final circuit design lab. Students were made aware of its existence through a class forum announcement and signs on chalkboards in the course's computer lab. It was left up for the remainder of the semester for students to refer to, if completing work late. We tracked student logins and engagement with the site's features. An initial prototype of Dear Beta was deployed for two consecutive semesters prior to this final system design and study, as well.

5.5 Dear Gamma

Hint Succession and Categorization

While Dear Beta makes all hints available at all times, Dear Gamma is modeled on the hint-giving mechanism of an intelligent tutoring system. In prior work, sequences of hints have been posited to facilitate learning due to their similarity with sequences used in expert human tutoring, as well as their support of human memory processes [112]. Therefore,

we further decomposed the hints collected with Dear Gamma into the three kinds of hints that typically comprise a hint sequence: 1) *pointing hints* direct the student's attention to the location of error in case the student understood the general principle but did not know to apply it; 2) *teaching hints* explain why a better solution exists by stating the relevant principle or concept; 3) *bottom-out hints* indicate concretely what the student should do [121].

Two researchers independently categorized the 435 collected Dear Gamma hints into six different categories: pure pointing hints (`p`), pointing and teaching hints (`pt`), pure teaching hints (`t`), teaching and bottom-out hints (`tb`), pure bottom-out hints (`b`), and hints that are irrelevant or clearly not helpful. They first independently labeled the first 30 hints. After discussing disagreements and iterating on their understanding of the hint categories, the coders then categorized the remaining 405 hints.

If one coder labeled a hint as a hybrid between two categories (i.e., teaching and pointing) while the other coder labeled it with only one category (i.e., pointing), we assigned the hint to the pure category (i.e., pointing) that was in common between the two coders' labels. If there was no shared category across the two coders, the hint was discarded. We also excluded the minority of hints (3.2%) that were labeled as irrelevant or unhelpful.

Lab Study

Nine out of the 226 current students in the computer architecture course participated in the study. These students were recruited through a course forum post. Participants were given $30 for the study, which lasted one hour. We informed students that we were studying the effectiveness of hints for optimizing circuits so that they use fewer transistors.

During the study, we presented the hints as anonymous, potentially helpful messages. Three batches of hints were shown in the order of pointing, teaching, and bottom-out, but randomly selected within each category. Students began by opening up their previously completed lab and reviewing their solution. The experimenter noted down the number of transistors in their solution, and randomly selected five pointing-type hints for a solution of that size from the Dear Gamma collection. For example, if the student had 114 transistors in their solution, they received five hints previously generated by students who had written

a hint to help improve a 114-transistor solution. Because hints may be of variable quality, the researcher presented hints in batches of five to increase the chances that one of them might be helpful.

The experimenter then asked the student to reduce the number of transistors in their solution. The experimenter explained that there were two more batches of five hints ready for them if they became stuck. These two batches were teaching hints and bottom-out hints. Students could consult outside resources like the course website and Google as well.

After receiving each batch of hints, participants answered the following 7-point Likert-scale questions about each hint (1: strongly disagree, 7: strongly agree): (1) *"This hint taught me something."* (2) *"This hint helped me get to a more optimal circuit."* and (3) *"I feel more confident that I could solve a similar problem in the future."* We placed these questions immediately after each batch of hints to capture user perception of hints at the time they occurred. However, to avoid slowing down the problem-solving process, participants were asked to explain their answers in writing only after the study, in the post-study questionnaire. This rating process was repeated for the teaching hints and bottom-out hints, even if students were able to solve the problem without asking for these hints.

After the study, users completed a post-study questionnaire regarding their overall impressions. Because users were shown a batch of hints at a time, all of which were student-generated, in the post-study questionnaire we added additional Likert-scale items, "I was able to find the most helpful hints and ignore the rest" and "Many hints felt repetitive," to understand whether users felt they could adequately ignore irrelevant hints.

5.6 Limitations

Because these studies do not have control groups, we cannot conclude on the magnitude of the effect on student learning. We can only report qualitative and quantitative measures of teachers' and students' engagement with the system. Some of those observed behaviors and opinions may be derived from the participants' sense of novelty, rather than the underlying value of the system. We deployed Dear Beta in a real classroom setting, and in the context of a real assignment, for the purpose of observing natural interaction with the system.

5.7 Results

5.7.1 Dear Beta Study

For the week prior to the lab assignment due date, the number of registered unique users in the Dear Beta system rose linearly from 20 to 166. It plateaued at 180 by one week after the lab was due. For comparison, the total number of students in the class was 226. 119 students logged in more than once and many students logged in repeatedly.

In the 9 days between Dear Beta's release and the lab's due date, users added 76 verification errors and 57 hints as a response to those errors. Half of the errors received at least one hint. Seven errors received as many as three hints. Figure 5-6 shows users' engagement with the system over time. As soon as the initial stock of hints were available, students began upvoting them.

Users contributed 61 upvotes and 10 downvotes on the hints during the same period. The highest number of upvotes (10) was given to the hint *"When entering constants, 1#4 is 1111 and 1'4 is the 4-bit representation of 1."* Remember that, while this is a teaching-type hint, it is provided as a targeted troubleshooting hint for students whose solution fails to pass a specific test case. The second most upvoted hint (5 upvotes) was *"Make sure your ASEL logic is correct - don't allow the supervisor bit to be illegally set."*

None of the hints appear to be incorrect, though this is difficult to verify, since the teachers do not have copies of the solutions from which these hints were generated. Even within a collection of hints for the same error, not all will be relevant to any particular solution.

5.7.2 Dear Gamma Study

With the Dear Gamma hints, six of the nine laboratory subjects were able to improve the optimality of their circuits within the hour that the study took place. Figure 5-7 illustrates the subjects' revisions. One student only needed one set of pointing hints. Five students successfully revised their circuits after one set of pointing and one set of teaching hints. Four students received a set of final bottom-out hints as well. Three of those four students

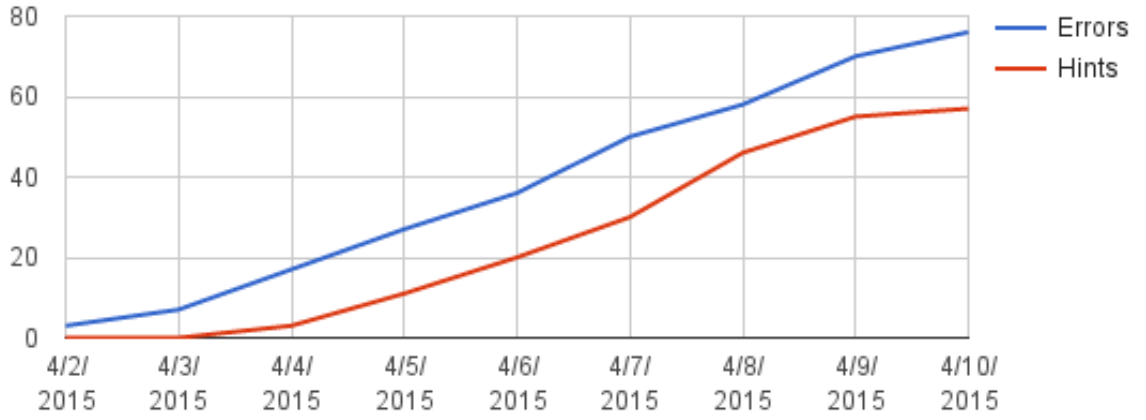


Figure 5-6: Between Dear Beta's release (4/2) and the lab's due date (4/10), verification errors were consistently being entered into the system. The addition of hints followed close behind.

(S_2, S_5, and S_9) were still unable to successfully revise their circuits.

Hint Distribution Figure 5-2 is a Sankey diagram of the optimization hints collected by Dear Gamma. The number of hints between certain key transitions, such as between the most common and the most optimal solutions, was far greater because of the lecturer's requests for pedagogically valuable hint prompts that introduced hint-writers to the common and optimal solutions.

The most common solution size was 114 transistors. Students with that common solution were randomly assigned to generate hints from one of the many different larger solutions down to theirs. These hints are pooled together with the hints written by students with solutions larger than 114 transistors who are seeing the common 114-transistor solution for the first time. Less than five percent of students' solutions were the most optimal (96 transistors), but, at the request of the lecturer, every student was asked to consider that most optimal solution and write a hint for a fellow student on how to optimize their solution into that most optimal solution.

Students in the study were drawn from the same population as the hint-generating students, and all study subjects were offered the same number of hints (5 pointing hints, followed by 5 teaching hints, followed by 5 bottom-out hints) over the course of the hour-long session, regardless of the solution they started with.

Hint Types Table 5.1 shows the breakdown of hints by type, along with representative

Hint type	Count	(%)	Representative examples
Pointing (p)	62	14%	"You don't have to keep S and Cout as two separate/independe
Pointing and teaching (pt)	81	19%	"Instead of making the S and Cout components individual, combine them together to save computation power."
Teaching (t)	111	26%	"Instead of recalculating values, save computation results to sa
Teaching and bottom-out (tb)	19	4%	"Via application of demorgan's theorem, NAND2 (XOR A B) Cin is equivalent to NAND3(NAND2 A
Bottom-out (b)	78	18%	"Use the output of a xor b for one of the nand2 gates."
Unhelpful/irrelevant	14	3%	"Use the hints provided by the lab, but try to improve on them.
No coder agreement	70	16%	

Table 5.1: Breakdown of Dear Gamma hints by type. Students in the Dear Gamma lab study initially received 5 pointing hints (p), followed by 5 pure teaching hints (t), and finally 5 pure bottom-out hints (b), delivered whenever the student was stuck and asked for more help.

examples. The Cohen's κ [24] inter-rater reliability was 0.54, which indicated that the two coders had moderate agreement across the six hint categories [122].

Hint Prompt Hint-authors interpreted the prompt to create a hint in different ways. Some addressed the hint-receiver directly (*"Keep in mind that..."*), while others addressed the teaching staff (*"I would mention [to the student]..."*). Some hint-authors did not directly write a hint, but instead wrote about how they would approach the situation of being a lab assistant for the hint-receiver: *"I think first I'd ask to make sure they knew what a NAND3 was, because I think a solution like this might come from not totally understanding how it works."* Still others took a conversational approach, as if they were having an unfolding conversation with the hint-receiver. Interestingly, a number of hint-authors referred to "here" or "my circuit" in their hints, as if the hint-receiver would be looking at the Dear Gamma interface, with all its examples, rather than just the text generated by the hint-author. This particular assumption on the part of the hint-author was confusing for hint-receivers.

Optimization Issues S_5 was the only student who had a standard, optimizable solution, received hints, and had no insights about how to optimize the circuit within the allotted hour. S_1, S_2, and S_9's forward progress was confounded by having near-optimal top-level

architecture and very large (suboptimal) implementations of the underlying modules. Dear Gamma only shows hint-authors the top-level architecture, not the underlying gate implementations, for the alternative solutions they compare their own solutions to. They therefore found the hints, which were often about fixing high-level architecture, irrelevant and unhelpful. Even so, S_1 was still able to revisit the hints and correctly extract the lesson that only inverting gates should be used. As a result, S_1 successfully optimized their circuit.

While working through their optimizations and hints, S_6 was the one student who significantly deviated from the correct line of thought by removing all inverting gates.[1] As soon as S_6 saw that their transistor count had increased rather than decreased, they revisited the hints, realized their mistake, and correctly optimized their circuit. None of the hints themselves were incorrect, though some were deemed irrelevant or unhelpful.

Hint Progression One student successfully optimized their solution from 150 transistors to the most common solution, 114 transistors, using only pointing and teaching hints. With some time left in the hour-long session, the student opted to optimize their circuit further. The experimenter gave the student one last set of hints, for the transition from 114 to the optimal 96-transistor solution. However, the experimenter did not restart the progression for this next transition; the student was given a set of bottom-out hints. Based on these hints, the student got the final optimization step without understanding, and appeared to feel cheated from the satisfaction of figuring it out himself.

Student Reactions The six subjects without suboptimal gates agreed with the statement *"Overall, these hints helped me get to a more optimal circuit"* (μ=6, σ=1.1 on a 7-point Likert scale). The remaining three subjects with suboptimal gates disagreed with the same statement (μ=2.6, σ=2.1 on a 7-point Likert scale). Regardless of whether a subject's solution included suboptimal gates, subjects on average agreed with the statement *"Hints helped me think differently about the problem, even if they didn't directly help me solve the problem"* (μ=5.4, σ=1.6).

Some subjects commented on the redundancy within each set of five hints of a particular type. This was sometimes expressed as a negative, as in *"These are all hinting at the same thing but I want new information,"* and sometimes expressed as a positive, as in *"Several*

[1] Optimal solutions have *only* inverting gates.

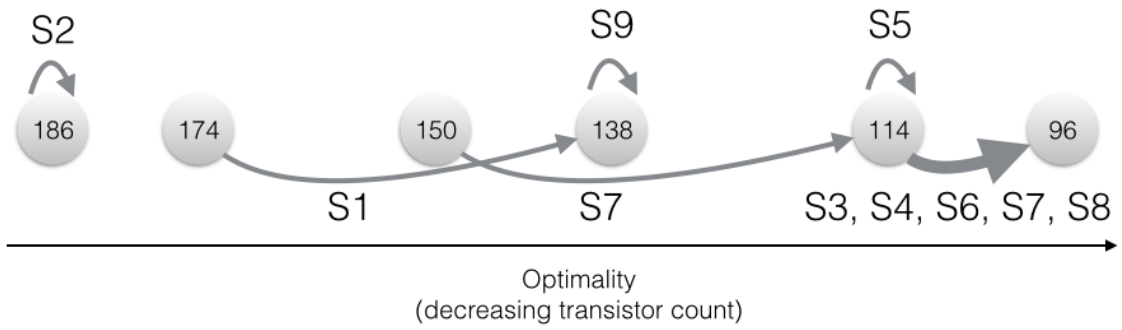


Figure 5-7: Six of the nine lab study subjects were able to improve the optimality of their circuits with the help of the Dear Gamma hints. Subject S7 was able to make two leaps–one to a common solution with 114 transistors and another from the common solution to the most optimal solution at 96 transistors.

hints are mentioning X.... I should look into it." One student told the experimenter that, while the individual hints were hard to understand, together they formed a clearer picture in her mind about what to do.

5.8 Discussion

In this section, we first address the research questions the evaluation was intended to answer. We then explain that, of the design decisions made during the design of Dear Beta and Dear Gamma, the critical factors for success included prompt clarity, the index chosen for hints, how alternative solutions are represented, and the use of hint progressions.

5.8.1 Answers to Research Questions

Our study sought to evaluate the characteristics of student-generated hints. We can see from Table 5.1 that students, without coaching, can naturally produce hints that point, teach, give a bottom-out direction, or provide some combination of those elements. However, the number of pointing hints labeled by *both* coders as purely pointing-type (22) was much smaller than the number of such hints in the teaching (75) and bottom-out (64) categories. Because students were not informed that their hint would be slotted into a progression, it is possible they may have felt that if they were going to give a future student one hint, it

would need to be more substantial than just pointing to a particular location in the solution and hoping the hint-receiver would see the possibility of optimization.

Secondly, the studies sought to evaluate whether learners can solve problems using these hints. Both studies suggest that these student-written hints are helpful. The aggregate activity of students and teachers on Dear Beta indicate that the resource was populated with helpful hints. The Dear Gamma lab study was set up based on the observed sub-optimality of students' circuits at the level of choosing and arranging gates. Students whose solutions were suboptimal in that anticipated way rated the hints as helpful. Students whose solutions were suboptimal in unanticipated ways, i.e., at the level of the gates themselves, were not well-served by the hints. Future optimization hint workflows will need both (1) an optimality metric that accounts for multiple common types of suboptimality and (2) a representation of solutions with an appropriate level of detail about the difference between any two solutions. Regardless, the Dear Gamma study suggests that students are helped by the hints when the optimality metric and representation are appropriate.

5.8.2 Lessons for Self-Reflection and Comparison Workflows

Prompt clarity appears to be critical for soliciting the highest possible quality of hints from students. In Dear Gamma, hint collection and delivery were separate processes. Some students misunderstood the prompt and wrote hints as if their audience was still the teacher, not a fellow student. Others did not understand that the hint-receiver would only see the text of their hint, not the diagrams it was based on. In Dear Beta, hint collection and delivery were all mediated through the same, constantly updated interface. The appropriate audience for a hint was clear. Future instantiations of the self-reflection and comparison workflows should clarify who the audience is for hint-authors, perhaps by displaying what learners will see.

The selection of an index for hints in the self-reflection workflow matters. In Dear Beta, the choice of test file name and test number as an index for hints worked well for a class of hundreds of students. In a MOOC-sized course, the index may need to include an indicator that specifies how the test failed as well. Indices in future systems should have sufficient

information to group related hints into clusters of a manageable size.

The success of the comparison workflow depends not only on the index for solutions, but also on how these solutions are represented in the workflow's prompt for hints. In the comparison workflow, we found that transistor counts sometimes did not account for lower-level reasons for a suboptimal circuit, resulting in hints that were unhelpful for solutions with lower-level suboptimality. Students will not generate hints that account for what has been abstracted away in the representation of solutions in the hint prompt. Likewise, if the definition of optimality used to index solutions does not account for a certain kind of suboptimality, the hints generated will be unlikely to help future students with that kind of suboptimality.

Lastly, when students request hints as they did in the Dear Gamma lab study, conforming to the intelligent tutoring system model of providing progressively specific hints is recommended. To automatically create hint progressions in the future, we could apply machine learning methods to estimate a given hint's type. Alternatively, we could learner-source hint classification.

5.8.3 Generalization

Although we applied these workflows to computer architecture problems, the self-reflection and comparison workflows could be extended to other domains. The workflows can be most readily applied to solutions that can be objectively tested for satisfying a set of requirements, e.g., passing unit tests, or whose optimality can be objectively measured. In domains without objective test cases or definitions for optimality, it may be more difficult to establish indices for clustering hints. In these cases, students could be asked to write what challenge they overcame or select from a growing list, enabling others to search for those terms. The comparison workflow could be modified to simply pair students with solutions different than their own, letting them judge for themselves which they think is better, the alternative solution or their own, and write hints based on that judgment.

add this: "In th next academic year, we plan

5.9 Conclusions

This paper enriches learnersourcing by shaping the design space for learnersourcing personalized hints, and presenting two workflows that engage learners in hint creation while reflecting on their own work as well as that of peers. We built Dear Beta and Dear Gamma, which apply these workflows to the creation of debugging and optimization hints, matching students to the appropriate hint creation task given their current progress. Results from our deployment study and subsequent lab study demonstrate the feasibility of these workflows, and indicate that learner-generated hints are helpful to learners. They also shed light on critical factors that may impact the quality of learnersourced hints, laying the groundwork for future systems in this area.

Chapter 6

Additional Clustering and Visualization

The original vision of OverCode was to discover pedagogically valuable themes of variation within thousands of student solutions to the same programming exercise. While OverCode does normalize and cluster correct solutions so that they can be more easily understood as a group, it does not pull out larger themes of variation as clearly as initially hoped for, nor does it handle incorrect solutions. This chapter describes more recent efforts to address both these shortcomings.

6.1 Clustering Solutions with Statistical Models

One approach to pulling out larger themes of variation within solutions is to cluster more aggressively. OverCode's existing clustering pipeline is a form of deterministic interpretable clustering. What can vary and what is invariant across all the solutions within a cluster is clear, just by reading the normalized solution that represents the cluster. Because it faithfully represents the syntax students used in each cluster, solutions are split apart into smaller clusters by small syntactic differences.

One promising set of methods for more aggressive groupings of solutions are bottom-up and rule-based: they define rules for which solutions or groups of solutions can be merged into a single cluster. Probablistic semantic equality or equality with respect to a teacher's test suite, which is already used by OverCode for variable names, can be used for subexpressions as well []. Programming language-based methods, e.g., compiler optimiza-

tions, could collapse OverCode clusters based on known semantic equality. As a system designer, one must decide or give the teacher control over how many different, semantically equivalent solutions are collapsed into a single cluster since, at the highest level, all correct solutions to the same programming exercise are semantically equivalent. These methods are discussed again in the future work, because the second set of methods were chosen for future investigation in this chapter.

The second set of methods leverage statistical properties of the entire corpus of solutions. Given that Variation Theory is interested in dimensions of variation (and consistency) that characterize all possible instantiations of an idea, statistical methods warrant further investigation. Latent variable models are a type of statistical model that attempt to explain variation in a dataset based on underlying factors. With the right choice of features and model, a latent variable model may be able to capture underlying design choices. In this section, two latent variable models have been explored in a preliminary way for this purpose: the Bayesian Case Model (BCM) [] and Latent Dirichlet Allocation (LDA) [**?**]

BCM was selected because, like the original OverCode pipeline, it clusters solutions and produces a single solution to represent the entire cluster. Like OverCode, it also indicates the features that characterize each cluster. While OverCode displays the set of normalized lines that all members of a cluster deterministically share, BCM learns a subspace of features that are most characteristic of the solutions within cluster, in a probabilistic sense. It also has an interactive variant called iBCM, which allows the user to directly modify the prototype and the subspace chosen by BCM if it disagrees with their domain knowledge or preferences. This user modification triggers a rerun of BCM with the modifications taken into account.

Internally, BCM depends on a mixture model with a Dirichlet prior. Rather than find a cluster for each entire solution, mixture models can learn clusters of features that co-exist across some subset of all solutions. The concentration parameter of BCM's Dirichlet prior is set to promote sparsity, i.e., mixture distributions over solutions that have the majority of their probability mass on a single mixture component. BCM then assigns each solution to a cluster according to its most probable mixture component.

LDA was selected as an alternative statistical approach to BCM because of evidence

collected at UCBerkeley [] that experts were themselves not particularly consistent about how to cluster solutions. While LDA's output is not as optimized for interpretability as BCM's, it preserves the ability to examine solutions through the lens of their mixture components, rather than clusters. LDA learns both mixture components, which are distribution over features, and the distributions of those mixtures over each solution in the set it is trained on. Depending on the features chosen to represent solutions, it may be difficult to interpret exactly what a mixture component is just by looking at its distribution over features. However, sorting solutions based on the degree to which a particular topic is associated with them may pull out a distinctive, human-interpretable themes.

6.1.1 Interpretable Clustering Solutions with BCM

BCM was applied to the normalized cluster-representing solutions generated by OverCode. These normalized solutions encode both static and dynamic information in a readable function, i.e., the syntax carries the static information and the variable name encodes dynamic information. The normalized function body is tokenized and represented as binary vectors indicating the existence of the features, including renamed variables and language-specific keywords, such as specific normalized variable names like `listA` and Python keywords like `assert` and `while`. The result is a BCM clustering of OverCode clusters.

In a small pilot study, three introductory Python teachers were each given sets of Python solutions to three different programming problems selected from those previously analyzed in the chapter on OverCode. For each problem, they were asked to create a grading rubric and provide helpful comments for the students, based on interacting with solutions in one of three different interfaces. The three interfaces were: (1) raw solutions in the browser, similar to the control used in the OverCode studies, (2) OverCode's normalized cluster-representating solutions, and (3) a BCM clustering of OverCode clusters, i.e., the prototypes and characteristic features of each BCM cluster, as well as its cluster members. The BCM interface, shown in Figure 6-1, was running the iBCM variant of the algorithm, so teachers could promote a member of a cluster to be its prototype or click on a token within a prototype–a variable name or keyword–to toggle whether or not it is considered a char-

acteristic feature of that cluster. Both these modifications triggered BCM running in the backend to rerun and send a new clustering to the front end for display.

Pilot users appreciated the fact that BCM gave some structure to the space of solutions; rather than a long list of solutions, the interface suggested distinct subpopulations of solutions within the list. However, subjects did not fully understand the probabilistic nature of the clustering method. The presence of a single "intruder", i.e., a solution that the teacher believed did not belong in a cluster, caused confusion. This could be ameliorated by giving users more ways to modify the clustering, e.g., allowing users the option to kick an intruder out of a cluster and rerun BCM, or by introducing the tool as a mechanism for "discovery" instead of "organization." Subjects also requested richer or higher-level features than variable names and keywords. Kim's PhD thesis [58] describes a follow-up full user study comparing the efficacy of BCM vs. iBCM on clustering OverCode cluster-representing solutions.

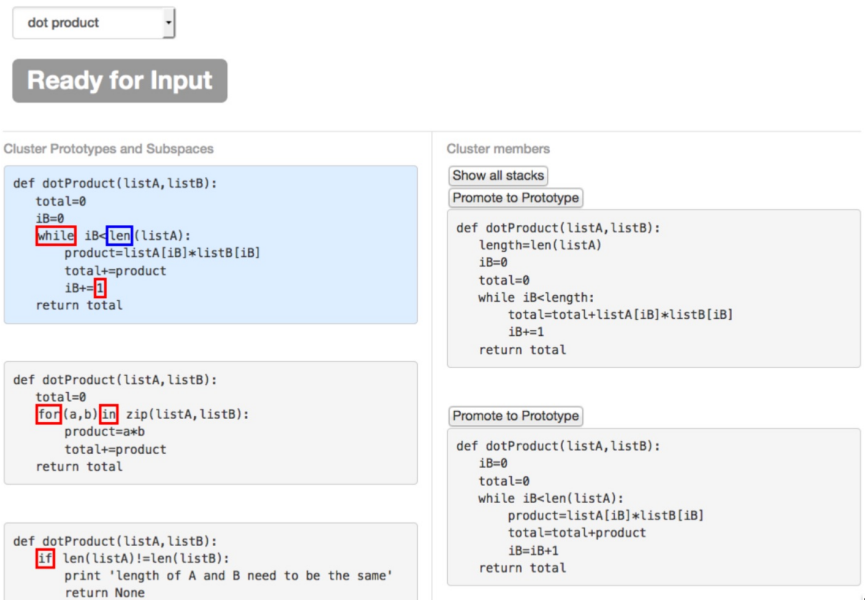


Figure 6-1: The solutions on the left are cluster prototypes. The blue solution is the selected cluster whose members are shown in a scrollable list on the right hand side. The tokens contained in red boxes are features that BCM "believes" characterize the cluster represented by that prototype. When a user hovers their cursor over a keyword or variable name, e.g., `len`, it is highlighted in a blue rectangle, indicating that it can be interacted with, i.e., clicked.

6.1.2 Mixture Modeling Solutions with LDA

As described in the chapter on related work, other researchers have documented a lack of agreement across human-made clusters of student code. One possible explanation for this low consistency across teacher-made clusterings of student code is that solutions are mixtures of design choices and teachers care about different things. As described in [**?**]: the clusters can be as straight forward as "1 pt", "2 pts" and "3 pts". If student A writes a solution with a well-written loop and extraneous statements while student B writes a solution with extra loops but otherwise very clean code, teachers can reasonably disagree about which cluster each whole solution should be placed in, depending on whether they believe inefficient control flow or extraneous statements are worse.

Instead of trying to approximate clusterings that humans do not even agree on, it may be more useful to model solutions as mixtures of good and bad design choices. While more sophisticated mixture models' assumptions may ultimately be more appropriate, LDA [**?**] as implemented in the Gensim toolbox [**?**] was chosen as the model to evaluate in this preliminary work.

Like BCM, LDA was run on the normalized cluster-representing solutions, not raw solutions. However, the representation of these solutions was also changed: in order to pull out higher-level patterns in approach, rather than lower-level patterns in syntax, solutions were represented solely by the behavior of the variables within them.

As described in the chapter on OverCode, the OverCode analysis pipeline executes all programs on a common set of test cases and records the sequence of values taken on by each variable in the program. OverCode assumes that variables in different programs that transition through the same sequence of values on the same test cases are in fact fulfilling the same semantic role in the program.

Below are the sequences of variable values recorded by OverCode while executing `iterPower(5,3)` as defined in Figures 6-2, 6-3, and 6-4:

- **Figure 6-2**

  - `exp: 3, 2, 1, 0, 1, 2, 3`
  - `base: 5`

- **Figure 6-3**
  - `exp: 3, 2, 1, 0`
  - `base: 5`
  - `result: 1, 5, 25, 125`
- **Figure 6-4**
  - `exp: 3`
  - `base: 5`
  - `iter: 3, 2, 1, 0`
  - `result: 1, 5, 25, 125`

In the previous examples, the input argument `base` would be considered variable common to all three programs, but the input argument `exp` would not be. The variable `result` would also be considered a common variable shared across just the definitions in Figure 6-3 and Figure 6-4. This allows us to distinguish between programs that calculate the answer in semantically distinct ways, without discriminating between the low-level syntax-based design decisions.

LDA is often applied to corpuses of textual documents, where the corpus is represented as a $W \times \mathrm{N}$ term-by-document matrix of counts, where W is the vocabulary size across all documents and N is the number of documents. In this representation, the document is represented a bag of word counts, i.e., how many times each word appears in the document. Following this analogy, solutions are represented as a bag of variable behaviors. The matrix representing the solutions in Figures 6-2 through 6-4 executed on `iterPower(5,3)` is shown in Table 6.1.

Note that, while the entries in Table 6.1 only take on values 0 or 1, more complicated definitions may have n instances of, e.g., a variable that takes on the sequence of values `3,2,1,0`. In that case, there could be an n in the `3,2,1,0` column for the row corresponding to that solution, or it can be left as a binary indicator.

In order to run LDA, 3875 student solutions to `iterPower` were first run on a set of test cases within the OverCode analysis pipeline. The OverCode pipeline produced a set of

Table 6.1: Variable-by-Solution Matrix for Programs, where variables are uniquely identified by their sequence of values while run on a set of test case(s)

SOLUTION	5	1,5,... 25,125	3,2,1,0, ...1,2,3	3,2,1,0	3
FIG. 6-2	1	0	1	0	0
FIG. 6-3	1	1	0	1	0
FIG. 6-4	1	1	0	1	1

977 cluster-representing solutions and a set of features for each solution, including which variable sequences were observed during execution. Another script turned this output into a variable-by-solution matrix for the 977 cluster-representing solutions, which were then fed into LDA for analysis. LDA was run repeatedly with multiple values for the parameter that sets the number of latent mixture components. The results were examined by hand since perplexity and held-out likelihoods are not necessarily good proxies for human interpretability [**?**].

Since the learned mixture components are distributions over variable behaviors, it is easier to inspect solutions which have high "amounts" of that mixture component "within" them and infer a theme by comparing them to solutions that contain high "amounts" of a different mixture component. These comparisons were done by hand for a subset of popular mixture components for each LDA model. One topic comparison captured the difference between solutions like Figure 6-4 and 6-3. Another topic comparison exposed the difference between the subpopulation of solutions with extra (unnecessary) conditional statements and the common, more concise solution. In the future, a user interface would be very helpful for this task, especially one which made it easier to compare the output of models with different parameter values.

LDA applied to a variable-by-solution matrix is a promising method for identifying variation within corpuses of solutions to the same programming exercise. However, LDA's assumptions, such as the independence of mixture components, and requirements, such as explicitly setting the number of mixture components beforehand, may mean that other mixture models, such as the Correlated Topic Model [] or the Hierarchical Dirichlet Process []

will ultimately be a better model fit for this purpose.

6.2 GroverCode: Clustering and Normalizing Incorrect Solutions

While understanding the contents of thousands of correct student solutions can be helpful in both residential and online contexts, another application of OverCode's pipeline and interface is supporting the hand-grading of introductory Python programming exam solutions, only some of which are correct. Incorrect solutions are defined as those that do not pass at least one test case in the teacher-designed test suite.

Hand-reviewing solutions is necessary because teacher-designed test suites were found to unfairly penalize some students and award undeserved credit to others. For example, a single typo in an otherwise well-written solution can cause it to fail all the test cases and receive no credit. Conversely, a solution that subverts the purpose of the assignment can still receive credit by returning the expected answer to some or all of the test cases.

For the staff of 6.0001, the residential introductory Python programming courses at MIT, this can be one of the most time-consuming and exhausting parts of teaching the course. It can take an full workday for the entire staff of eight to ten teachers to sit in a room and review several hundred students' solutions by hand in order to assign a single numerical grade to each solution.

There are two main technical contributions found in the GroverCode extension of OverCode: a modified pipeline that can normalize both correct and incorrect solutions and an interface designed for grading. Just as the original pipeline used variable behavior to normalize variable names, the modified pipeline uses variable behavior and the syntax of statements containing each variable to normalize variable names in solutions that are not correct. This comes from the insight that a variable in an incorrect solution can be semantically equivalent to a variable in a correct solution but still behave differently; it can behave differently due to a bug in a line of code that directly modifies its value or a bug in a line of code that affects the behavior of another variable that it depends on. Since many of the

exam solutions are incorrect and do not get clustered together, the new user interface organizes solutions according to their behavior on test cases and compositional similarity to each other, rather than by cluster size.

GroverCode was iteratively designed and evaluated as a grading tool through two live deployments during the Spring 2016 6.0001 staff exam grading sessions. Approximately two hundred students were enrolled in the course, and nine instructors used GroverCode to grade nearly all student code submissions. A total of seven programming exercises of a variety of difficulties were graded over the course of these two sessions. GroverCode was particularly appreciated on simpler problems where correct solutions were clustered together and graded by a single teacher's action. The normalization process applied to more complex solutions, especially those which included student-written objects, was at times more harmful than helpful for teacher's comprehension of student code. Conversely, the feature for grouping solutions based on their behavior on test cases was appreciated regardless of solution complexity.

6.2.1 User Interface

Figures 6-5, 6-6, and 6-7 capture the GroverCode user interface during the second and final deployment of the Spring 2016 term. Unlike OverCode, there are both correct and incorrect solutions in view; they are differentiable by their different background colors, as well as a red "X" next to every failed test. To better explain each test failure, the solution's differing actual and expected outputs are also displayed.

The panel of progress indicators and filters is always in view. The rows of aligned sequences of green checks and red dashes represent "error vectors", the pass/fail results with respect to the ordered list of teacher-specified test cases. The checkbox next to each error vector allows the teacher to selectively view subsets of solutions based on the particular tests they pass and fail. Solutions with the same error vectors may include similar mistakes.

Correct solutions are stacked the same way they are in OverCode. In contrast, incorrect solutions were normalized but not stacked, even if they were identical after normalization. Grades are propagated to all the solutions in a stack; propagating actual student grades

based on a probabilistic normalization process that was not thoroughly vetted before deployment would be unfair to students.

The user interface includes the following features in an attempt to minimize the cognitive load experienced by teachers rapidly switching between grading one solution and the next:

- Horizontally aligned solutions for side-by-side comparison
- Easy filtering of solutions by input-output behavior, i.e., their error vectors
- Normalized variable names
- Solutions ordered to minimize the distance between adjacent solutions with respect to a custom similarity metric defined in Terman **??**
- Highlighted lines of code in each solution which differ from the previous solution in the horizontal list

The hope is that normalization helps teachers switch between solutions more often than it harms reability.

te Stacey's the-
s and Been's
esis for any
orrowed figures

6.2.2 Implementation

The GroverCode implementation is a modification of the OverCode pipeline, including the updates described in Section 3.8.2. Solutions which return an expected output for all test cases during the preprocessing step are categorized as "correct", and all other solutions, having failed at least one test case, are categorized as "incorrect". Correct solutions are normalized by the same process as was used in the original OverCode.

In the original OverCode pipeline for correct solutions, variable behavior is assumed to hold enough semantic information to be the sole basis on which variable names are normalized. In this first version of GroverCode, applies this rule to incorrect solutions

as well, but only for variables whose behavior matches a common variable in the correct solutions.

However, that purely behavior-based renaming strategy may change in future versions of GroverCode because, given that incorrect solutions are known to be wrong with respect to input-output behavior, the behavior of the variables within them is suspect too, regardless of whether it happens to match a common variable in a correct solution.

Consider the following example: in the syntax of an incorrect solution, a variable `i` may be operated or depended on exactly the same way as a common variable in a correct solution. However, if an error somewhere else in the solution causes `i` to behave differently, the original method of variable renaming will be thrown off. Based on this example, variables in incorrect solutions that are not already renamed based on behavior are renamed based on syntax.

The most recently updated OverCode pipeline represents each line's syntax and variables in a separable way (see Section 3.8.2). The line's syntax, e.g., `for __ in __:`, is referred to as a template. For each common variable in correct solutions, GroverCode counts how many times it appears in each template and in which location, represented as an index into the blanks in the template. Examples of templates and locations are shown in Table **??**.

get explicit permission

If a yet-unrenamed variable in an incorrect solution appears in the exact same template-locations as a common variable in a correct solution, it will be renamed to match that common variable.

If a variable in an incorrect solution is still not canoncialized, the counts of template-locations associated with each common variable in the correct solutions are used, as described in detail in Terman [**?**], to infer the most likely common variable it could be renamed to, as long as a threshold for similarity is met. Otherwise, its original name is kept.

6.2.3 Field Deployment

The GroverCode analysis pipeline was run on both the midterm and final exam problems from the Spring 2016 semester of 6.0001, which had approximately 200 students enrolled. These exams contained seven programming problems in total, and between 133 and 189 solutions per problem made it through the analysis pipeline to be displayed in the user interface.

Using the GroverCode user interface, nine instructors, including one lecturer and eight teaching assistants (TAs) graded these solutions as part of their official grading responsibilities. Those solutions that did not successfully pass through the pipeline were graded by hand afterwards.

The TAs' grading events, e.g., adding or applying rubric items and point values to solutions, were logged. An observer took extensive notes during each day-long grading session to capture spontaneous feature requests as well as bugs and complaints. For full disclosure, one of the TAs in these grading sessions was the Masters of Engineering student who implemented most of GroverCode, and the observer was the author of this thesis.

Programming Exam Problems

For each problem processed during the field deployments, an example of a staff-written correct solution is included below. Given the increasing difficulty of these exam problems, the following numbers dropped between the midterm and the final: the number of students present to take the exam, the number of students who submitted any solution to a given problem before the test period ended, and the number of correct solutions that could be clustered.

Midterm Problems

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

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

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

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

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

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

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

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

Final Exam Problems

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

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

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

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

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

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

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

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

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

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

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

Table 6.1: Example: All templates and locations in which the abstract variable `exp`, the second argument to a recursive `power` function, appears. A location represents the index or indices of the blanks that the abstract variable occupies, where the first blank is index 0, the second is index 1, and so on. The second and third columns together form a template-location pair. Copied with permission from Terman [**?**]

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

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

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

Pipeline Evaluation

The number of submissions submitted for each problem and the number that were successfully processed by the GroverCode pipeline is shown in Table 6.2.

Table 6.3 captures the scale of the variation as well as some clustering statistics.

	Quiz			Final			
	q4	q5	q6	q4	q5	q6	q7
Submissions	193	193	193	175	173	170	165
Mean lines per submission	9.9	16.9	19.8	12.3	20.9	50.0	41.8
Solutions successfully processed	186 (96%)	189 (98%)	168 (87%)	170 (97%)	166 (96%)	134 (79%)	133 (81%)

Table 6.2: Number of submissions submitted and successfully processed by GroverCode for each problem in the dataset. Reasons why a solution might not make it through the pipeline include syntax errors and memory management issues caused by students' inappropriate function calls.

Table 6.4 summarizes the counts of the various mechanims by which variables in incorrect solutions were normalized.

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

Table 6.3: The degree of variation in input-output behavior and statistics about stack sizes.

Discussion

For simpler solutions, variable renaming was an invisible and possibly slightly confusing helping hand. One grader remarked, outloud: "Why is everyone naming their iterator variable 'i'?" at which point he had to be reminded of the variable renaming process. As solutions became more varied and structurally complex, graders started immediately looking the raw solutions because the renaming of variables, removal of comments, and standardization of whitespace in the normalized solutions was removing clues they needed in order to understand the student's intent.

	Quiz			Final			
	q4	q5	q6	q4	q5	q6	q7
Vars. in incorrect submissions	15	149	482	289	550	559	859
Vars. renamed based on values	14	84	266	97	246	97	187
Vars. renamed based on templates	0	58	166	136	264	188	489
Vars. not renamed	1	7	50	56	40	274	183

Table 6.4: Statistics about variables renaming based on different heuristics in the GroverCode normalization process.

While it was difficult to get direct feedback on the helpfulness of pair-wise difference highlighting and optimized solution ordering, graders heavily used and appreciated the ability to filter and grade solutions one error vector at a time. At least one grader remarked aloud that it seemed like many of the solution with the same error vector made similar mistakes. Therefore filtering by error vector may have been one of the stronger contributors to any hypothetical decreased cognitive load due to using GroverCode over the status quo of random assignment to solutions in a CSV file.

6.3 Conclusion

The clustering described in the original OverCode work was relatively limited in scope, but it did produce, at least for simple introductory Python programming problems, a concise standardized representation of solutions that can be used for more statistically sophisticated clustering techniques and as a starting point for helping graders understand and grade incorrect student solutions by hand.

```
def iterPower(base, exp):
    '''
    base: int or float.
    exp: int >= 0

    returns: int or float, base^exp
    '''
    # Your code here
    if exp <= 0:
        return 1
    else:
        return base * iterPower(base, exp - 1)
```

Figure 6-2: Example of a recursive student solution.

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

Figure 6-3: Example of a `while`-based student solution.

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

Figure 6-4: Example of a `while`-based student solution, where the student has not modified any input arguments, i.e., better programming style.

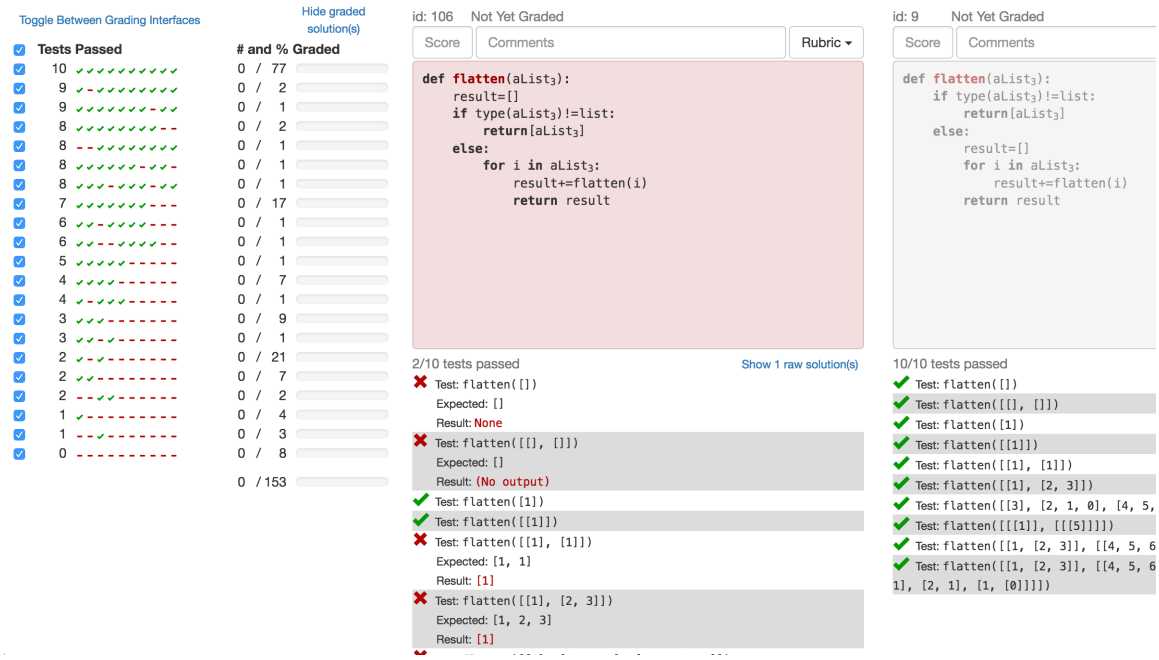


Figure 6-5: The GroverCode user interface displaying solutions to an introductory Python programming exam problem, in which students are asked to implement a function to flatten a nested list of arbitrary depth.

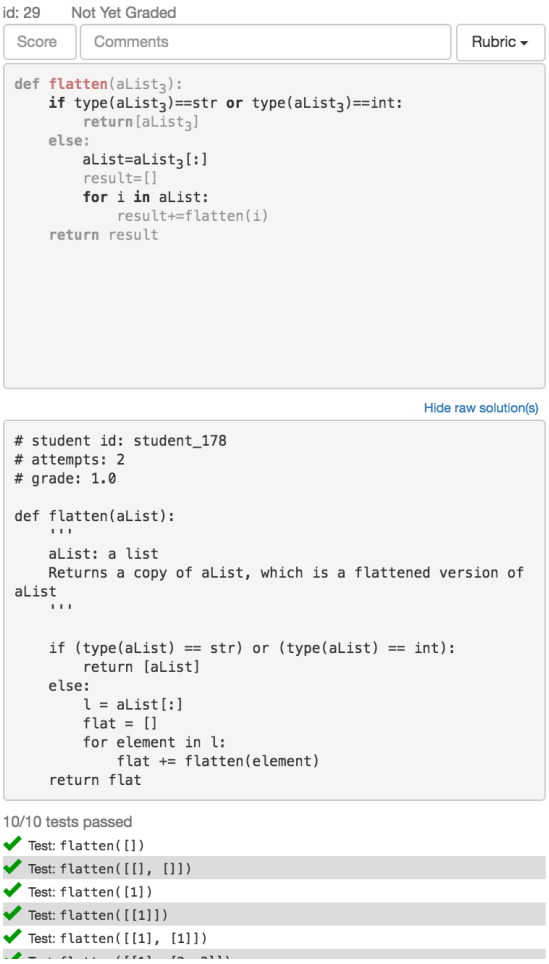


Figure 6-6: A correct solution, its corresponding raw submission, and its performance on test cases, as displayed in GroverCode.

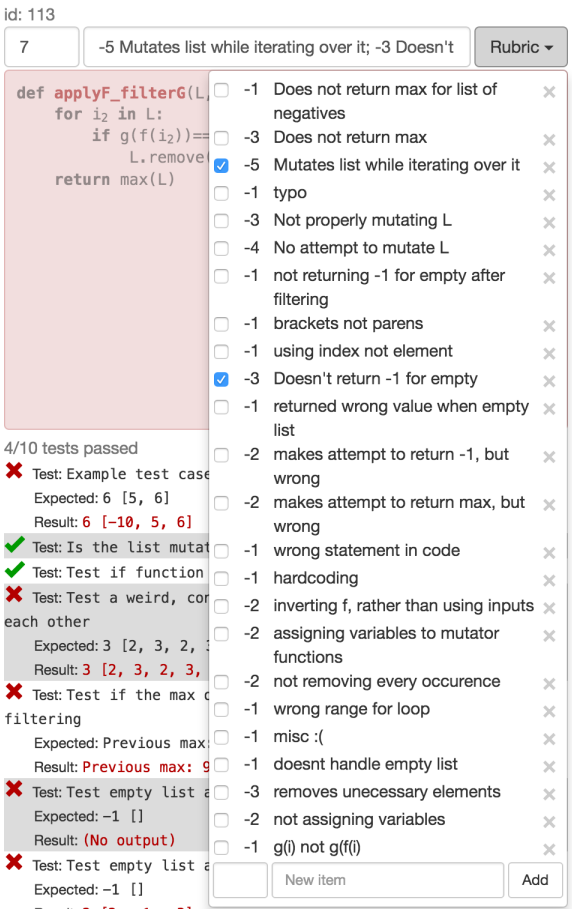


Figure 6-7: A stack with the rubric dropdown menu open. Text for each of the checked items is automatically inserted in the comment box.

Chapter 7

Discussion

The systems in this thesis give teachers more awareness about the content generated by students in large programming classes and enable styles of teaching that are usually only feasible in smaller classes, such as discussions of variation and style that are directly driven by what the students have already written. These systems also to scale up automated compare-contrast, self-explanation, and formative assessment-style exercises whose content is generated by students and curated by teachers. The current state of the art in theories of how humans learn predict that these supported interactions between teachers and students will enhance learning.

add citations fo value of forma- tive assessment to related work

7.1 Design Decisions and Choices

This thesis work began with the vision that a teacher would someday be able to look at a display summarizing hundreds or thousands of student solutions to the same problem and immediately see—and comment on—good and bad design decisions that students made. The number of possible distinct student solutions grows rapidly with the number of design decisions and design choices students can make.

The solution space could be imagined as a large n-dimensional space where each solution has a single coordinate. Each design decision, e.g., whether to solve a problem iteratively or recursively, would be a dimension in the solution space. The choices students make could be thought of as discrete points along that dimension in solution space.

While the number of distinct combinations of design choices students choose can be large, the number of dimensions in this space, i.e., the underlying design decisions, grows much more slowly.

However, the structure of this hypothetical solution space ignores how design choices affect each other. Some design decisions are mutually exclusive, e.g., looping over a particular array with a for or a while, some decisions are correlated with one another, and some decisions are completely independent. A tree-like description of the solution space can capture some of these relationships between decisions. If a node represents a design choice, e.g., to loop over an array, there may be two or more choices, e.g., a `for` or a `while` loop, that can be represented as child nodes. The choice to use a `for` loop poses an additional design design, e.g., whether to use `range` or `xrange`: `for i in [x]range(input)`. The average depth of the tree would correspond to the average number of design decisions the students faced and the average branching factor would correspond to the average number of distinct choices students in the corpus make at each decision point.

The curse of dimensionality predicts that, as we add more and more dimensions to the solution space, the density of solutions will decrease and the liklihood that any two solutions occupy the same location in that space will go down. The regularity of code discussed in the chapter on related work should help ameliorate the curse of dimensionality, but only partly. In other domains, it is often necessary to collect more data as the dimensionality of the space increases. However, given that the solutions are generated by students, the number of correct solutions are more likely to go down rather than up as the complexity of solutions, and the associated dimensionality of the solution space, increases.

This difficulties posed by high-dimensional spaces have been addressed, in each problem tackled, by choosing what information to ignore and, in some cases, what information to index by. For example, OverCode ignores white space, comments, variable names, and statement orders. It indexes by canonicalized lines. In order to assign a variable name quiz, Foobaz looks at the behavior of the variables in the student's solution, ignoring everything else about the solution's composition. Dear Beta and GroverCode forget what values cause a solution to fail a test case, preserving only that the test case has failed. Dear Gamma ignores circuit topology and indexes by the number of transistors.

It is important to note that this work is not intended to capture an enumeration of all possible design choices and resulting solutions. It is intended to capture the design choices students are actually making. The relative popularity of these choices is discussed in the section that follows.

7.2 Capturing the Underlying Distribution

There will be some design decisions within each solution that are rare and some that are common, some that exemplify good programming practices and others that do not. These decisions might create inefficient solutions, reveal a student's fundamental misunderstanding, or use a feature of the language in a creative way. The distribution of solutions along these dimensions of variation may reflect student prior knowledge, teacher explanations, and common misconceptions.

During Hannah Wallach's invited talk at the interpretable machine learning workshop at ICML 2016, she made the following observation: computer scientists are often looking for better ways to find needles in haystacks and computationally-minded social scientists are trying to characterize the haystack. One could think of work like Codex [36] and Webzeitgeist [66] as mechanisms for finding needles in haystacks through creative indexing of Ruby code and webpages, respectively. OverCode, Foobaz, Dear Beta, Dear Gamma, and GroverCode are trying to faithfully represent the haystack itself, while also supporting needle-finding.

7.3 Writing Solutions Well

The focus on introductory Python programming courses is often just correctness. The MIT EECS introductory Python programming course whose staff used GroverCode makes it a policy not to penalize students on how their solution is written. This means that they sometimes have to hand out full marks to a solution that makes them groan.

This policy may exist for several reasons. The first is the clarity of the policy: if it is correct, it gets full credit. Second is the clarity of the evaluation: if it passes the suite of

test cases, it is correct. Third is the apparent appropriateness of the objective for novices: it may be too hard for novices to write a solution well in addition to achieving correctness.

However, it can also be very hard to achieve a correct solution if it is not written well. Something as simple as poor variable names, such as giving an array index a name that suggests it holds the value of the array at that index, can cause students to produce incorrect code.

Just as there is no silver bullet for writing prose well, there is no silver bullet for writing solutions well. Solutions, prose, products, buildings etc., are all designed, and each community has its own ways for how to help students make good design decisions. For example, in Steven Pinker's book "The Sense of Style", he suggests and then demonstrates how to pick apart examples of good writing to understand what makes them good. Students of a particular design form may attend design studios, where they discuss each of their work in turn and give and receive constructive criticism from peers. In both these examples of prose and design, the designed objects are examined individually and also as a group, emulating the conditions for learning from variation espoused by the theories of learning reviewed in the related work.

Perhaps because software can be marvelously complex, there is less of a design studio culture for software. Many software companies compose and maintain prescriptive style guides against which new code is carefully compared. These guides are not necessarily built on data about how software engineers actually design their code.

While peer review is not a design studio, they are closer to that mode of education. Peer review practices are now being used in large online courses in order to make up for small teacher-to-student ratios. Some residential software engineering courses, e.g., MIT's 6.005, also set up infrastructure for peer review, even when the number of staff members is sufficient. This forces students to engage with some (usually random) sampling of how other students solved a problem.

However, the work in this thesis takes this idea farther. Rather than hope that the randomly assigned peer review experiences provides students with a sufficient variety of examples, the work in this thesis attempts to pull out the design dimensions as well as concrete examples along those dimensions and, when possible, ask students to engage with them in

a targeted, personalized way.

This may be helpful even at the level of introductory programming. For example, according to Variation Theory, a student will understand the concept of a loop in a more generalized, robust way if they have seen all the different ways in which their peers have written that loop. The teacher can quickly and easily provide an expert's perspective by commenting on the popular and rare choices. Students can be overwhelmed by choices, e.g., "Should I use `range` or `xrange`? Does it matter?" With concrete examples, teachers can help students identify what matters and what does not.

Many students now taking introductory programming courses will take these skills with them to other majors. While computer science majors can acquire the skills of writing code well in more advanced software engineering classes, the lessons in introductory courses on writing code well may be the only ones non-majors ever get.

7.4 Clustering and Visualizing Solution Variation

OverCode is a form of unsupervised clustering. Clustering is a function of similiarity measures and mechanisms for grouping or splitting clusters of data points. There is no true correct answer, but there are distinct failure modes when using it in the context of teachers reviewing solutions. Two are most relevant in this work. First, the representation of the solution and/or the measure of similiarity between solutions can ignore, hide, or otherwise fail to capture what the teacher cares about. Second, when the teacher cannot infer what a cluster "means" based on its members and the clustering algorithm cannot explicitly communicate why the cluster exists, the teacher may not trust the clustering and may not feel comfortable using it for propagating feedback and grades back to students. This is exacerbated when the teacher discovers a member of a cluster that seems not to belong.

The OverCode clustering pipeline attempts to escape the first clustering failure mode: erasing distinctions that the teacher may care about. For reasons discussed in detail in the OverCode chapter, OverCode is designed to reveal to the teacher what their students' solutions actually look like, modulo white space and comments. These solutions are rendered

eck the state-
ent order

for the teacher using the most common variable names and statement orders.To stay true to what students actually wrote, this process preserves syntactic differences. For example, when iteratively exponentiating `base` to an exponent, there are multiple ways to multiply an accummulating variable `result` by `base` and save the product back into `result`, such as `result *= base` and `result = result * base`. If the teacher just gave a lecture on common forms of syntactic sugar, OverCode will be sensitive to whether or not students use it. OverCode would allow teachers to observe whether students absorbed the lesson on syntactic sugar based on the way they write their subsequent solutions. The fact that even small differences in syntax creates separate clusters within the OverCode pipeline nearly ensures that any syntactic choices a teacher is interested in has been preserved and can be filtered for in the interface.

The second clustering failure mode–producing clusters that the teacher does not trust–is avoided in several complementary ways. First, their is a clear interpretation of what an OverCode cluster can and cannot include, based on the canonicalized solution that represents it. Specifically, all solutions in the cluster have the same set of lines after the behavior-based canonicalization of their variable names, regularization of white space, and removal of comments. Second, the differences between clusters is made clear by highlighting which lines make the non-reference clusters different from the reference cluster. Third, OverCode's filtering features and rewrite rules help teachers change their view of solutions into one that preserves the differences they are explicitly interested in and ignores those they are not interested in. Note that filtering by semantic choices may only be possible when the work on Bayesian modeling of these solutions becomes more mature.

ld "Users do
ot notice re-
amed variables
nless the names
e inappropriate
do not look
ke they are hu-
an generated.
ariable names
corpus-wide

In general, the interfaces in this thesis cluster complex objects and visualize those clusters so that there is little guesswork about what is included and excluded in a group and what the boundary between groups is. There are no outliers that are grouped with something else without explanation. Rather than losing faith in the clustering process, outliers can be used as the teacher sees fit to spur improvements either to the software infrastructure of the course, e.g., the input-output testing harness, or the examples students are asked to engage with.

7.5 Language Agnosticism

It is not surprising that the evolving values of variables would carry significant semantic meaning in code written by students at the introductory level in languages like Python and Matlab, especially if the style of programming is procedural. This thesis confirms that within the context of introductory procedural Python programs. According to Taherkhani et al. [116], variables carry useful semantic meaning in object-oriented and functional programming styles as well. Taherkhani et al. [], Gulwani et al. [44], and [105] have all authored variable-behavior-based semantic analyses of Java, C++, and Pascal programming languages, respectively. One could think of it as behavior-based semantic variable duck typing.

OverCode and Foobaz could also be applied to other 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. For non-variable-centric languages like Haskell, other dynamic characteristics of execution would likely need to be tracked.

7.6 Limitations

The thesis presents a series of case studies about how to present the variety of programming solutions in a human-interpretable way and make use of it in pedagogically valuable, scalable ways. The methods described only work in a particular domain of solutions: those that are executable and solve the same programming exercise. This excludes natural language, for example. These case studies embody the design principles espoused, but there is no validated unifying recipe by which a corpus of student solutions can be processed and used. Each corpus and set of teachers' values were considered together in order to engineer a representation of solutions–both in the pipeline and in the user interface–that would empower teachers and benefit students. There are many specific technical limitations of the approaches described in the previous chapters. In the next chapter, the section on future work describes these limitations and suggests next steps.

I am not aware of any complex domain where this kind of feature engineering and design has been automated. General guidelines and trial-and-error are accepted practice. Before the resurgence of deep neural nets training on massive data sets, one could reasonable argue that carefully human-designed features are were responsible for a great deal of the performance of machine learning systems. Otherwise, systems fall into the failure mode of garbage in, garbage out.

7.7 Design Recommendations

While this thesis does not offer a unifying recipe, these are some design recommendations based on experience accumulated since the start of this thesis:

1. When possible to do with high confidence, propagate human-assigned labels to similar data points.

2. Avoid outliers within clusters at all costs; they cause doubt and confusion.

Chapter 8

Conclusion

This thesis demonstrates several ways to capture distributions of student solutions and make this information useful to teachers and students in large programming classes. Capturing commonalities can save teachers time and exhaustion by summarizing many solutions with a single representative. It can also direct teachers to where the majority of the class is within the space of solutions or errors, so they can respond to student needs based on data, not hunches. Capturing variation allows teachers to see how students deviate from the common choices in ways that are superior and deserving celebration or inferior and requiring corrective feedback. These deviations can be used to inspire or directly seed exercises based on modern theories of how students learn generalizeable knowledge from concrete examples. It also saves teachers the time and effort of generating pedagogically valuable examples themselves. Systems in this thesis also demonstrate how the inherent variation in student solutions can be automatically presented back to students, as an active reflection exercise that can benefit both current and future students.

The technical challenges tackled in order to capture and display distributions of solutions include designing solution representations, measures of similarity between solutions, mechanisms for grouping or filtering them, and human-interpretable displays. Challenges in workflow design centered around simple, minimally-intrusive ways to perform targeted learnersourcing in an on-going programming class.

8.1 Summary of Contributions

The original OverCode system included a novel visualization that shows similarity and variation among thousands of Python solutions, backed by an algorithm that uses the behavior of variables to help cluster Python solutions and generate human-interpretable canonicalized code for each cluster of solutions. The Foobaz system demonstrated how, even when an aspect of variation is hidden in one view of a distribution of solutions, it can be exposed, in context, in a separate view. Additional follow-on work expanded OverCode's canonicalization process to incorrect solutions and explored additional human-interpretable mechanisms for identifying population-level design alternatives.

Foobaz contributed a workflow for generating personalized active learning exercises, emulating how a teacher might socratically discuss good and bad choices with a student while they review the student's solution together. The Foobaz system demonstrates one concrete way in which a conversation about design decisions, i.e., variable naming, that might only happen in one-on-one interactions with an instructor, can be scaled up to an arbitrarily large class, personalized to each student, be based on the distribution of naming choices found in solutions composed by the teacher's current class or previous classes, and deployed to future classes who assign the same programming exercise.

Finally, the self-reflection and comparison workflows demonstrate the potential of targeted learnersourcing. The self-reflection workflow allows students to generate hints for each other by reflecting on an obstacle they themselves have recently overcome while debugging their own solution. The comparison workflow prompts students to generate design hints for each other by comparing their own solutions to alternative designs submitted by other students. Studies of the Dear Beta and Dear Gamma systems show that personalized hints collected through these workflows can be viably learnersourced, and that these hints serve as helpful guides to fellow students encountering the same obstacle or attempting to reach the same goal.

8.2 Future Work

"One never notices what has been done; one can only see what remains to be done..." - Marie Curie (1984)

There are many directions in which this work can be grown. Some of the research that has been published in parallel with this work can be directly incorporated into future systems. At the same time, many of the existing features can be improved without incorporating any new techniques, based on already collected feedback during user studies, deployments, and other design critiques.

8.2.1 OverCode

OverCode was designed for thousands of correct solutions to introductory Python programming problems. Many residential and online exercises now afford automatic, immediate feedback about the correctness of a solution. Under these conditions, nearly all the final solutions are correct, but it is clear from reading through them that some students have better command of programming than others, just based on compositional quality. As a result, OverCode was designed to reveal to the teacher what their students' solutions actually look like, modulo white space and comments. These solutions are rendered for the teacher using most common variable names and statement orders.

check the statement order

However, as programming exercises escalate from the simple to the complex, OverCode's utility breaks down. It can still create canonicalized cluster-representing solutions and display them in a way that helps teachers spot contrasts, but the clusters themselves become smaller and smaller. As programming exercises require more complex solutions, there is also an increasing need to go beyond representing what students actually wrote and toward human-interpretable representations of larger clusters. There are two promising complementary avenues for implementing this next step.

The first avenue is more bottom-up clustering. Just as the variation in variable names was hidden by recognizing that variables across programs were semantically equivalent and could be replaced with their most popular student-given name, it may be possible to do the same with respect to semantically-equivalent subexpressions. There two promising

ways to determine the semantic equivalence of subexpressions or lines of code in large corpuses of student solutions: (1) semantic equivalence based on the rules of the programming language (2) probablistic semantic equivalence based on the dynamic behavior during execution on test cases, as defined in Nguyen et al. [89]. OverCode could feature an additional button which collapses these detected equivalences, showing the most popular choice in the solution representing the new, larger clusters. An interface, perhaps like the teacher interface in Foobaz, can allow teachers to view and compose feedback that dimension of variation, if they wish.

The second promising avenue for creating larger clusters is continuing to model the corpus of solutions to the same programming exercise as samples from an underlying distribution of solutions shaped by the teacher's explanations, student's prior knowledge, common novice misconceptions, and the constraints of the language itself. Depending on the model chosen, larger clusters could be groups of whole solutions or groups of sub-components the co-occur within many solutions. Iterating on solution representation, model choice, interface, and interaction will hopefully allow teachers to understand their students' design choices, give style feedback, and assign grades with confidence.

Even simple additions to the OverCode interface could approximate what more sophisticated statistical methods would do. For example, LDA may pull out distributions of commonly co-occuring variable behaviors (as described in the chapter on OverCode extensions), but the current OverCode interface could be modified to suggest and filter by common sets of strictly co-occurring variable behaviors. This approximation is less robust in the face of noise, but it is at least easier for the user to interpret using the language of filtering.

When visualizing the resulting clusters, it is important to keep in mind the ease with which users can interpret the results. Simple modifications to the OverCode interface can go a long way, such as highlighting the sub-expressions or canonicalized variable names within lines of code, rather than the entire lines, that separate one cluster from another. Another modification would increase the interpretability of the canonicalized variables. Since the variable behavior is used to canonicalize variable names, the values of variables and sub-expressions during execution on a test case could be displayed just beneath or

alongside the code, as demonstrated in an experimental Python notebook [**?**], in addition to the way it is currently shown, as a legend mapping each variable name to its behavior beneath a tab that many user study subjects did not consult.

When the program tracing, renaming, or reformatting scripts generate an error while processing a solution, the solution is removed from the pipeline. This percentage of correct edX solutions that did not make it through the pipeline was 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. As solutions get more complex, it will also be necessary to expand the OverCode pipeline to more gracefully handle user-created objects and helper functions. Complementary efforts, like [22], also cannot yet handle more than a single function. This remains a difficult but important hurdle to expanding beyond introductory Python programming courses. As part of this expansion, it may be helpful to adopt the `observe` construct introduced by Gulwani et al. [44] and support teacher annotation of solutions with the observe construct within the OverCode user interface. By soliciting human annotation of important variable values, two variables would not need to behave identically in order to be considered by OverCode as semantically the same.

OverCode could also be integrated with the autograder that tests student solutions 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, teachers could give 'power feedback' by writing line- or stack-specific feedback to be sent to students alongside the input-output test results. The OverCode interface may also provide benefit to students, in addition to teaching staff, after students have solved the problem on their own. Cody is a standing example of the value of this kind of design/style feedback. However, the current interface may need to be adjusted for use by non-expert students, instead of teachers.

Simple interface enhancements would give teachers, and potentially students, more freedom to explore. For example, many user study subjects requested the ability to promote any cluster to be the reference cluster. A student might want to see their solution as the reference, at least as a starting point. Similar to GroverCode's sorting mechanism, the

OverCode interface could support sorting clusters by their similarity to the reference as well as by cluster size.

ld text about
oolean variables
ot being re-
mable, require
ore informa-
on, perhaps
en more than
roverCode col-
ctions

8.2.2 GroverCode

GroverCode's canonicalization of incorrect solutions is based on heuristics about co-occurrences of syntax and variable behavior in both correct and incorrect solutions. This was an attempt to see how far the OverCode features could go toward canonicalizing incorrect solutions in addition to correct solutions, without using any additional technology. However, there is pre-existing technology, i.e., AutoGrader [109], that could be added to the pipeline.

The AutoGrader understands a language for expressing corrections in introductory Python programs. The authors created a list of such possible corrections, and if applying a small subset of those corrections changes a solution's input-output behavior to match a reference solution, the Autograder can automatically correct the solution. It does not take into account which sets of corrections are more common, unlike the design philosophy the

eck this

defines OverCode.However, the corrections that are applied are guaranteed to convert the incorrect solutions into correct ones. The correct solutions can move through the OverCode pipeline to be canonicalized and clustered without heuristics.

If the AutoGrader is added to the pipeline, it will likely move some but not all solutions from the incorrect to the correct category. The corrections that have been automatically applied could be displayed alongside the corrected code in the GroverCode interface, for transparency. The GroverCode process for analyzing and displaying solutions that are still incorrect after being analyzed by the AutoGrader would be unchanged. If the language for expressing corrections is simple and expressive enough that it can be used by graders to write new rules while they are grading, the combined AutoGrader-OverCode pipeline could be rerun to correct any other yet ungraded solutions that are not already automatically corrected.

There are also a variety of interface updates that have also been requested by the teachers who used GroverCode during live deployments, such as the ability to add new tests to

the test suite. These requests have been catalogued and added the the project as feature requests.

8.2.3 Foobaz

The approach taken in designing Foobaz may be generalizable to other aspects of programming style. Just as OverCode establishes the equivalency of variables based on their behavior during test cases, one could establish the behavior equivalency of larger or more abstract entities, such as student-written lines of code, sets of lines of code, or entire functions. We consider this a class of problems that Foobaz, and similar systems built after it, can tackle. Future iterations of Foobaz-inspired interfaces could also include additional constraints and affordances to encourage teachers to leave more explanations for their assessments, accompanied by better support for reusing common comments. Teachers could also be given the option to augment or overwrite existing labels, e.g., "too short," to match their own preferences.

8.2.4 Targeted Learnersourcing

The primarily difficulty of iterating on the two workflows, Dear Beta and Dear Gamma, is that they are most successful when they are directly integrated into teacher's own systems and endorsed or promoted by staff. In future classes with supportive teachers, a 'prompting' module can be added to the Dear Beta workflow which identifyies when a student is in a good place to supply a hint. This would be an improvement over the current stand-alone website that students are explicitly directed to when they need help, but not when they can give help. Future work on Dear Gamma should include improving the metric for optimality and the rendering of solutions such that all the differences between more and less optimal solutions are visible to reflecting, hint-writing students. One potential future incarnation of the Dear Beta and Dear Gamma workflow is a crowd-sourced intelligent tutor that helps students get to a correct solution and then optimize that solution, completely driven by student-written hints.

Bibliography

[1] URL http://d.ucsd.edu/srk/papers/2014/TalkaboutLAS2014.pdf. 68

[2] K. Alharbi and T. Yeh. Collect, decompile, extract, stats, and diff: Mining design pattern changes in android apps. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '15, pages 515–524, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3652-9. doi: 10.1145/2785830.2785892. URL http://doi.acm.org/10.1145/2785830.2785892. 49

[3] M. Allamanis and C. Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 472–483. ACM, 2014. 51, 52

[4] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002. 53

[5] S. Basu, C. Jacobs, and L. Vanderwende. Powergrading: a clustering approach to amplify human effort for short answer grading. *TACL*, 1:391–402, 2013. 66, 69

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–377, Washington, DC, USA, 1998. IEEE Computer Society. 57

[7] B. Biegel, Q. D. Soetens, W. Hornig, S. Diehl, and S. Demeyer. Comparison of similarity metrics for refactoring detection. In *Proceedings of the 8th Working Con-*

ference on Mining Software Repositories, MSR '11, pages 53–62, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985452. URL http://doi.acm.org/10.1145/1985441.1985452. 60

[8] D. Binkley, D. Heinz, D. Lawrie, and J. Overfelt. Understanding lda in source code analysis. In *Proceedings of the 22Nd International Conference on Program Comprehension*, pages 26–36. ACM, 2014. 63

[9] S. A. Birch and P. Bloom. The curse of knowledge in reasoning about false beliefs. *Psychological Science*, 18(5):382–386, 2007. 42

[10] B. S. Bloom. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher*, pages pp. 4–16, 1984. 23, 41

[11] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Papers from the sixth international world wide web conference syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29(8):1157 – 1166, 1997. ISSN 0169-7552. doi: http://dx.doi.org/10.1016/S0169-7552(97)00031-7. URL http://www.sciencedirect.com/science/article/pii/S0169755297000317. 60

[12] M. Brooks, S. Basu, C. Jacobs, and L. Vanderwende. Divide and correct: using clusters to grade short answers at scale. In *Learning at Scale*, pages 89–98, 2014. 66, 99

[13] R. P. Buse and W. Weimer. Synthesizing api usage examples. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 782–792. IEEE, 2012. 53

[14] T. J. Bussey, M. Orgill, and K. J. Crippen. Variation theory: A theory of learning and a useful theoretical framework for chemical education research. *Chem. Educ. Res. Pract.*, 14:9–22, 2013. doi: 10.1039/C2RP20145C. URL http://dx.doi.org/10.1039/C2RP20145C. 47

[15] C. Camerer, G. Loewenstein, and M. Weber. The curse of knowledge in economic settings: An experimental analysis. *Journal of Political Economy*, 97(5):pp. 1232–1254, 1989. ISSN 00223808. URL http://www.jstor.org/stable/1831894.

[16] A.-K. Carstensen and J. Bernhard. Laplace transforms: Too difficult to teach, learn, and apply, or just a matter of how to do it? 2004. 47

[17] J. Case. Education theories on learning: an informal guide for the engineering education scholar. 2008. 41

[18] R. Catrambone and K. J. Holyoak. Overcoming contextual limitations on problem-solving transfer. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 15(6):1147, 1989. 45

[19] J. C. Chen, D. C. Whittinghill, and J. A. Kadlowec. Using rapid feedback to enhance student learning and satisfaction. In *Proceedings. Frontiers in Education. 36th Annual Conference*, pages 13–18, Oct 2006. doi: 10.1109/FIE.2006.322306. 43

[20] M. T. Chi, N. De Leeuw, M.-H. Chiu, and C. Lavancher. Eliciting self-explanations improves understanding. *Cognitive Science*, 18(3):pp. 439–477, 1994. ISSN 1551-6709. doi: 10.1207/s15516709cog1803_3. URL http://dx.doi.org/10.1207/s15516709cog1803_3. 42, 143

[21] M. T. Chi, S. A. Siler, H. Jeong, T. Yamauchi, and R. G. Hausmann. Learning from human tutoring. *Cognitive Science*, 25(4):471–533, 2001. 42

[22] R. R. Choudhury, H. Yin, J. Moghadam, and A. Fox. Autostyle: Toward coding style feedback at scale. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion*, pages 21–24. ACM, 2016. 58, 62, 65, 197

[23] D. Coetzee, A. Fox, M. A. Hearst, and B. Hartmann. Should your mooc forum use a reputation system? In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '14, pages 1176–1187, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2540-0. 143

[24] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. 155

[25] L. Deslauriers, E. Schelew, and C. Wieman. Improved learning in a large-enrollment physics class. *Science*, 332(6031):862–864, 2011. ISSN 0036-8075. doi: 10.1126/science.1201783. URL http://science.sciencemag.org/content/332/6031/862. 43

[26] J. Dewey. How we think: A restatement of the relation of reflective thinking to the educational process. *Lexington, MA: Heath*, 1933. 42

[27] D. Diziol, E. Walker, N. Rummel, and K. R. Koedinger. Using intelligent tutor technology to implement adaptive support for student collaboration. *Educational Psychology Review*, 22(1):89–102, 2010.

[28] S. D'Mello, B. Lehman, R. Pekrun, and A. Graesser. Confusion can be beneficial for learning. *Learning and Instruction*, 29(0):pp. 153–170, 2014. ISSN 0959-4752. doi: http://dx.doi.org/10.1016/j.learninstruc.2012.05.003. URL http://www.sciencedirect.com/science/article/pii/S0959475212000357. 45

[29] F. Doshi-Velez. personal communication.

[30] F. Doshi-Velez et al. The indian buffet process: Scalable inference and extensions. *Master's thesis, The University of Cambridge*, 2009. 64

[31] A. Driver, K. Elliott, and A. Wilson. Variation theory based approaches to teaching subject-specific vocabulary within differing practical subjects. *International Journal for Lesson and Learning Studies*, 4(1):72–90, 2015. doi: 10.1108/IJLLS-10-2014-0038. 47

[32] A. Drummond, Y. Lu, S. Chaudhuri, C. M. Jermaine, J. Warren, and S. Rixner. Learning to grade student programs in a massive open online course. In *ICDM*, pages 785–790, 2014. 58

[33] K. Ehrlich and E. Soloway. An empirical investigation of the tacit plan knowledge in programming. In *Human factors in computer systems*, pages 113–133. Norwood, NJ: Ablex Publishing Co, 1984. 59

[34] B. S. Elenbogen and N. Seliya. Detecting outsourced student programming assignments. *J. Comput. Sci. Coll.*, 23(3):50–57, Jan. 2008. ISSN 1937-4771. URL http://dl.acm.org/citation.cfm?id=1295109.1295123. 58

[35] R. M. et al. 6.005 software construction. 50

[36] E. Fast, D. Steffee, L. Wang, J. R. Brandt, and M. S. Bernstein. Emergent, crowd-scale programming practice in the ide. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 2491–2500. ACM, 2014. 51, 52, 54, 64, 187

[37] A. Fox, D. A. Patterson, and S. Joseph. *Engineering software as a service: An Agile approach using Cloud Computing*. Strawberry Canyon LLC, 2013. 48

[38] M. Gaudencio, A. Dantas, and D. D. Guerrero. Can computers compare student code solutions as well as teachers? In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 21–26, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2605-6. doi: 10.1145/2538862.2538973. URL http://doi.acm.org/10.1145/2538862.2538973. 60, 61

[39] M. Gaudencio, A. Dantas, and D. D. Guerrero. Can computers compare student code solutions as well as teachers? In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE '14, pages 21–26, New York, NY, USA, 2014. ACM. 69

[40] E. L. Glassman, N. Gulley, and R. C. Miller. Toward facilitating assistance to students attempting engineering design problems. In *Proceedings of the Tenth Annual International Conference on International Computing Education Research*, ICER '13, New York, NY, USA, 2013. ACM. 65

[41] E. L. Glassman, R. Singh, and R. C. Miller. Feature engineering for clustering student solutions. In *Proceedings of the first ACM conference on Learning@ scale conference*, pages 171–172. ACM, 2014. 62

[42] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):7, 2015. 129

[43] F. Gobet. *Deliberate Practice and Its Role in Expertise Development*, pages 917–919. Springer US, Boston, MA, 2012. ISBN 978-1-4419-1428-6. doi: 10.1007/978-1-4419-1428-6_104. URL http://dx.doi.org/10.1007/978-1-4419-1428-6_104. 43

[44] S. Gulwani, I. Radiček, and F. Zuleger. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 41–51, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5. doi: 10.1145/2635868.2635912. URL http://doi.acm.org/10.1145/2635868.2635912. 59, 191, 197

[45] S. Gulwani, I. Radiček, and F. Zuleger. Automated clustering and program repair for introductory programming assignments. *arXiv preprint arXiv:1603.03165*, 2016. 59

[46] P. J. Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1868-6. 29, 81, 87

[47] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages pp. 1019–1028. ACM, 2010.

[48] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337322. 51, 52

[49] E. W. Høst and B. M. Østvold. The java programmerâĂŹs phrase book. In *Software Language Engineering*, pages 322–341. Springer, 2008. 54

[50] E. W. Høst and B. M. Østvold. Debugging method names. In *ECOOP 2009–Object-Oriented Programming*, pages 294–317. Springer, 2009. 54

[51] J. Huang, C. Piech, A. Nguyen, and L. J. Guibas. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED Workshops*, 2013. 58, 61, 62, 65, 69, 91

[52] D. M. Jones. Operand names influence operator precedence decisions (part. 2008. 54

[53] D. M. Jones. Operand names influence operator precedence decisions, February 2008.

[54] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani. Semi-supervised verified feedback generation. *arXiv preprint arXiv:1603.04584*, 2016. 58

[55] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002. ISSN 0098-5589. doi: 10.1109/TSE.2002.1019480. 57, 60

[56] L. Kavanagh and L. O'Moore. Reflecting on reflection-10 years, engineering, and uq. In *Proceedings of the 19th Annual Conference of the Australasian Association for Engineering Education: To Industry and Beyond*. Institution of Engineers, Australia, 2008. 45

[57] J. Kibler. Cognitive disequilibrium. In S. Goldstein and J. Naglieri, editors, *Encyclopedia of Child Behavior and Development*, pages pp. 380–380. Springer US, 2011. ISBN 978-0-387-77579-1. doi: 10.1007/978-0-387-79061-9_598. URL http://dx.doi.org/10.1007/978-0-387-79061-9_598. 45

[58] B. Kim. Interactive and interpretable machine learning models for human machine collaboration, 2015. PhD thesis. 63, 164

[59] B. Kim, C. Rudin, and J. A. Shah. The bayesian case model: A generative approach for case-based reasoning and prototype classification. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 1952–1960. Curran Associates, Inc., 2014. 63

[60] B. Kim, J. A. Shah, and F. Doshi-Velez. Mind the gap: A generative approach to interpretable feature selection and extraction. In *Advances in Neural Information Processing Systems*, pages 2251–2259, 2015. 63

[61] J. Kim, R. C. Miller, and K. Z. Gajos. Learnersourcing subgoal labeling to support learning from how-to videos. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pages 685–690. ACM, 2013. 24, 68, 137

[62] S. Klemmer. Scaling studio critique: success, bruises, and future directions. MIT CSAIL Seminar, 2012. 68

[63] D. Knuth. Literate programming, June 2000. Retrieved April 8, 2015 from http://www.literateprogramming.com/.

[64] J. L. Kolodner. Educational implications of analogy: A view from case-based reasoning. *American psychologist*, 52(1):57, 1997. 45

[65] R. Kumar, C. P. Rosé, Y.-C. Wang, M. Joshi, and A. Robinson. Tutorial dialogue as adaptive collaborative learning support. *Frontiers in Artificial Intelligence and Applications*, 158:383, 2007.

[66] R. Kumar, A. Satyanarayan, C. Torres, M. Lim, S. Ahmad, S. R. Klemmer, and J. O. Talton. Webzeitgeist: Design mining the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3083–3092. ACM, 2013. 48, 58, 187

[67] K. J. Kurtz, C.-H. Miao, and D. Gentner. Learning by analogical bootstrapping. *The Journal of the Learning Sciences*, 10(4):417–446, 2001. 45

[68] K. J. Kurtz, C.-H. Miao, and D. Gentner. Learning by analogical bootstrapping. *The Journal of the Learning Sciences*, 10(4):417–446, 2001. 45

[69] M. Lim, R. Kumar, A. Satyanarayan, C. Torres, J. Talton, and S. Klemmer. Learning structural semantics for the web. Technical report, Tech. rep. CSTR 2012-03. Stanford University, 2012. 63

[70] L. M. Ling, P. Chik, and M. F. Pang. Patterns of variation in teaching the colour of light to primary 3 students. *Instructional Science*, 34(1):1–19, 2006. ISSN 1573-1952. doi: 10.1007/s11251-005-3348-y. URL http://dx.doi.org/10.1007/s11251-005-3348-y. 47

[71] M. Ling Lo. *Variation theory and the improvement of teaching and learning*. Göteborg: Acta Universitatis Gothoburgensis, 2012. 46

[72] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi. Mining concepts from code with probabilistic topic models. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 461–464, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321709. URL http://doi.acm.org/10.1145/1321631.1321709. 63

[73] M. L. Lo, W. Y. Pong, and P. P. M. Chik. *For each and everyone: Catering for individual differences through learning studies*, volume 1. Hong Kong University Press, 2005. 47

[74] J. Loewenstein, L. Thompson, and D. Gentner. Analogical learning in negotiation teams: Comparing cases promotes learning and transfer. *Academy of Management Learning & Education*, 2(2):119–127, 2003. 45

[75] J. Loewenstein, L. Thompson, and D. Gentner. Analogical learning in negotiation teams: Comparing cases promotes learning and transfer. *Academy of Management Learning & Education*, 2(2):119–127, 2003. 44, 45

[76] A. Luxton-Reilly, P. Denny, D. Kirk, E. Tempero, and S.-Y. Yu. On the differences between correct student solutions. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*, ITiCSE '13, pages 177–182, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2078-8. 61

[77] A. Mabanta, C. Hunt, S. Najmabadi, and K. Ridenoure. How big is uc berkeleyâĂŹs biggest class?, 2013. http://www.dailycal.org/2013/09/03/how-big-is-uc-berkeleys-biggest-class/.

[78] F. Marton and S. A. Booth. *Learning and awareness*. Psychology Press, 1997. 46

[79] F. Marton and M. F. Pang. On some necessary conditions of learning. *The Journal of the Learning sciences*, 15(2):193–220, 2006. 47

[80] F. Marton, A. B. Tsui, P. P. Chik, P. Y. Ko, and M. L. Lo. *Classroom discourse and the space of learning*. Routledge, 2004. 46

[81] F. Marton, A. Tsui, P. Chik, P. Ko, and M. Lo. *Classroom Discourse and the Space of Learning*. Taylor & Francis, 2013. ISBN 9781135642334. 69, 73

[82] E. Mazur. *Peer Instruction: A User's Manual*. Series in Educational Innovation. Prentice Hall, 1997. URL http://mazur-www.harvard.edu/publications.php?function=display&rowid=0. 68

[83] M. Mhlolo. The merits of teaching mathematics with variation. *Pythagoras*, 34(2), 2013. ISSN 2223-7895. URL http://pythagoras.org.za/index.php/pythagoras/article/view/233. 47

[84] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[85] P. Mitros. Learnersourcing of complex assessments. In *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, pages 317–320, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3411-2. 68, 137

[86] A. Muralidharan and M. Hearst. Wordseer: Exploring language use in literary text. *Fifth Workshop on Human-Computer Interaction and Information Retrieval*, 2011.

[87] A. Muralidharan and M. A. Hearst. Supporting exploratory text analysis in literature study. *Literary and linguistic computing*, 28(2):283–295, 2013.

[88] A. S. Muralidharan, M. A. Hearst, and C. Fan. Wordseer: a knowledge synthesis environment for textual data. In *CIKM*, pages 2533–2536, 2013.

[89] A. Nguyen, C. Piech, J. Huang, and L. J. Guibas. Codewebs: scalable homework search for massive open online programming courses. In *WWW*, pages 491–502, 2014. 57, 64, 69, 196

[90] G. Noble, P. Tabar, and P. Scott. Contents image of publication peer reviewed citation only bookmark and share more information about this publication'if anyone called me a wog, they wouldn't be speaking to me alone'. 1998. 47

[91] F. G. Paas and J. J. van Merrienboer. Variability of worked examples and transfer of geometrical problem-solving skills : a cognitive-load approach. *Journal of Educational Psychology*, 86(1):122–133, 1994. URL `http://doc.utwente.nl/26427/`. 44

[92] K. Papadopoulos, L. Sritanyaratana, and S. R. Klemmer. Community tas scale high-touch learning, provide student-staff brokering, and build esprit de corps. In *Proceedings of the First ACM Conference on Learning @ Scale*, L@S '14, pages 163–164, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2669-8.

[93] T. Peters. The zen of python. In *Pro Python*, pages 301–302. Springer, 2010. 50

[94] J. L. Quilici and R. E. Mayer. Role of examples in how students learn to categorize statistics word problems. *Journal of Educational Psychology*, 88(1):144, 1996. 44

[95] M. J. Rees. Automatic assessment aids for pascal programs. *SIGPLAN Not.*, 17(10): 33–42, Oct. 1982. ISSN 0362-1340. doi: 10.1145/948086.948088. URL http://doi.acm.org/10.1145/948086.948088. 58

[96] D. Ritchie, A. A. Kejriwal, and S. R. Klemmer. d. tour: Style-based exploration of design example galleries. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 165–174. ACM, 2011. 48

[97] B. Rittle-Johnson and J. R. Star. Does comparing solution methods facilitate conceptual and procedural knowledge? an experimental study on learning to solve equations. *Journal of Educational Psychology*, 99(3):561, 2007. 44

[98] K. Rivers and K. R. Koedinger. Automatic generation of programming feedback; a data-driven approach. In *AIED Workshops*, 2013.

[99] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, pages 1–28, 2015. 57

[100] S. S. Robinson and M. L. Soffa. An instructional aid for student programs. *SIGCSE Bull.*, 12(1):118–129, Feb. 1980. ISSN 0097-8418. doi: 10.1145/953032.804623. URL http://doi.acm.org/10.1145/953032.804623. 60

[101] S. Rogers, D. Garcia, J. F. Canny, S. Tang, and D. Kang. *ACES: Automatic evaluation of coding style*. PhD thesis, MasterâĂŹs thesis, EECS Department, University of California, Berkeley, 2014. 58, 61, 62, 65

[102] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009. 56

[103] J. M. Rzeszotarski and A. Kittur. Crowdscape: interactively visualizing user behavior and output. In *UIST*, pages 55–62, 2012.

[104] A. Sahami Shirazi, N. Henze, A. Schmidt, R. Goldberg, B. Schmidt, and H. Schmauder. Insights into layout patterns of mobile user interfaces by an automatic analysis of android apps. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 275–284, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2138-9. doi: 10.1145/2494603.2480308. URL http://doi.acm.org/10.1145/2494603.2480308. 49

[105] J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *null*, page 37. IEEE, 2002. 59, 191

[106] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85. ACM, 2003. 60

[107] D. Shasha, J.-L. Wang, K. Zhang, and F. Y. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man and Cybernetics*, 24(4):668–678, 1994. 61

[108] V. J. Shute. Focus on formative feedback. *Review of educational research*, 78(1): 153–189, 2008. 43

[109] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462195. URL http://doi.acm.org/10.1145/2491956.2462195. 198

[110] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

[111] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, Sept. 1984. ISSN 0098-5589. 57

[112] R. A. Sottilare, A. Graesser, X. Hu, and B. Goldberg. *Design Recommendations for Intelligent Tutoring Systems: Volume 2-Instructional Management*, volume 2. US Army Research Laboratory, 2014. 150

[113] S. Sridhara, B. Hou, J. Lu, and J. DeNero. Fuzz testing projects in massive courses. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*, pages 361–367. ACM, 2016. 55

[114] S. Srikant and V. Aggarwal. A system to grade computer programming skills using machine learning. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1887–1896. ACM, 2014. 58, 60

[115] J. Suhonen, J. Davies, E. Thompson, et al. Applications of variation theory in computing education. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*, pages 217–220. Australian Computer Society, Inc., 2007. 47

[116] A. Taherkhani, A. Korhonen, and L. Malmi. Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms. *The Computer Journal*, page bxq049, 2010. 58, 59, 191

[117] K. Topping. Peer assessment between students in colleges and universities. *Review of Educational Research*, 68(3):pp. 249–276, 1998. 68

[118] J. Turns, B. Sattler, K. Yasuhara, J. Borgford-Parnell, and C. Atman. Integrating reflection into engineering education. In *Proceedings of the ASEE Annual Conference and Exposition*. ACM, 2014. 42

[119] J. J. Van Merriënboer, R. E. Clark, and M. B. De Croock. Blueprints for complex learning: The 4c/id-model. *Educational technology research and development*, 50 (2):39–61, 2002. 44

[120] J. J. G. van Merriënboer. *Variability of Practice*, pages 3389–3390. Springer US, Boston, MA, 2012. ISBN 978-1-4419-1428-6. doi: 10.1007/978-1-4419-1428-6_415. URL http://dx.doi.org/10.1007/978-1-4419-1428-6_415. 44

[121] K. Vanlehn, C. Lynch, K. Schulze, J. A. Shapiro, R. Shelby, L. Taylor, D. Treacy, A. Weinstein, and M. Wintersgill. The andes physics tutoring system: Lessons learned. *International Journal of Artificial Intelligence in Education*, 15(3):147–204, 2005. 141, 151

[122] A. J. Viera and J. M. Garrett. Understanding interobserver agreement: the kappa statistic. *Fam Med*, 37(5):360–363, 2005. 155

[123] L. Vygotsky. Zone of proximal development. *Mind in society: The development of higher psychological processes*, 5291, 1987. 142

[124] H. J. Walberg. Improving the productivity of america's schools. *Educational leadership*, 41(8):19–27, 1984. 41

[125] R. Wass and C. Golding. Sharpening a tool for teaching: the zone of proximal development. *Teaching in Higher Education*, 19(6):671–684, 2014. doi: 10.1080/13562517.2014.901958. 43

[126] S. Weir, J. Kim, K. Z. Gajos, and R. C. Miller. Learnersourcing subgoal labels for how-to videos. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '15, pages 405–416, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2922-4. 68, 137, 139, 140

[127] D. Weld, E. Adar, L. Chilton, R. Hoffmann, E. Horvitz, M. Koch, J. Landay, C. Lin, and Mausam. Personalized online education – a crowdsourcing challenge. In *Proceedings of the 4th Human Computation Workshop (HCOMP '12) at AAAI*, 2012. 68

[128] D. Wood, J. S. Bruner, and G. Ross. The role of tutoring in problem solving*. *Journal of child psychology and psychiatry*, 17(2):89–100, 1976. 43

[129] W. B. Wood and K. D. Tanner. The role of the lecturer as tutor: doing what effective tutors do in a large lecture class. *CBE-Life Sciences Education*, 11(1):3–9, 2012. 42

[130] S. Xu and Y. S. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Trans. Softw. Eng.*, 29(4):360–384, 2003. 57

[131] H. Yin, J. Moghadam, and A. Fox. Clustering student programming assignments to multiply instructor leverage. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*, pages 367–372. ACM, 2015. 61