

**Clustering and Visualizing Solution Variation in Massive
Programming Classes**

by

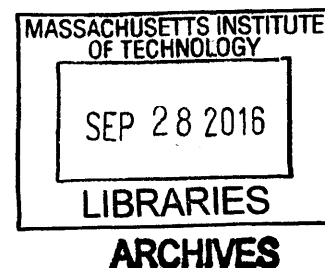
Elena L. Glassman

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

September 2016



© Massachusetts Institute of Technology, 2016. All rights reserved.

Author..... **Signature redacted**

Department of Electrical Engineering
and Computer Science
August 11, 2016

Certified by..... **Signature redacted**

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

Accepted by..... **Signature redacted**

Leslie A. Kolodziejcki
Professor of Electrical Engineering and Computer Science
Chair, 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 11, 2016, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In large programming classes, a single problem may yield thousands of student solutions. Solutions can vary in correctness, approach, and readability. Understanding large-scale variation in solutions is a hard but important problem. For teachers, this variation could be a source of innovative new student solutions and instructive examples. Understanding solution variation could help teachers write better feedback, test cases, and evaluation rubrics. Theories of learning, e.g., analogical learning and variation theory, suggest that students would benefit from understanding the variation in the fellow student solutions as well. Even when there are many solutions to a problem, when a student is struggling in a large class, other students may have struggled along a similar solution path, hit the same bugs, and have hints based on that earned expertise.

This thesis describes systems that exploit large-scale solution variation and have been evaluated using data or live deployments in on-campus or edX courses with thousands of students. OverCode visualizes thousands of programming solutions using static and dynamic analysis to cluster similar solutions. Compared to the status quo, OverCode lets teachers quickly develop a high-level view of student understanding and misconceptions and provide feedback that is relevant to more student solutions. Foobaz helps teachers give feedback at scale on a critical aspect of readability, i.e., variable naming. Foobaz displays the distribution of student-chosen names for each common variable in student solutions and, with a few teacher annotations, it generates personalized quizzes that help students learn from the good and bad naming choices of their peers. Finally, this thesis describes two complementary learnersourcing workflows that help students write hints for each other while reflecting on their own bugs and comparing their own solutions with other student solutions. These systems demonstrate how clustering and visualizing solution variation can help teachers directly respond to trends and outliers within student solutions, as well as help students help each other.

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. I only needed 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" or "nerd," and the production and popularity of movies about programmers may be responsible for driving up enrollment in computer science classes to unprecedented levels. Hundreds or thousands of students enroll in programming courses 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 Andrés Monroy-Hernández, 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, for all the hugs, proofreading, brainstorming, snack deliveries, technical discussions, and moral support.

Contents

- Cover page** **1**

- Abstract** **3**

- Preface** **5**

- Acknowledgments** **7**

- Contents** **9**

- List of Figures** **15**

- List of Tables** **23**

- 1 Introduction** **25**
 - 1.1 Solution Variation 26
 - 1.2 A Challenge and an Opportunity 29
 - 1.3 A Tale of Two Turing Machines 30
 - 1.4 Research Questions 31
 - 1.4.1 OverCode 31
 - 1.4.2 Foobaz 32
 - 1.4.3 Learnersourcing Personalized Hints 35
 - 1.4.4 Systems as Answers 36
 - 1.5 Thesis Statement and Contributions 38
 - 1.6 Thesis Overview 40

2	Related Work	43
2.1	Clustering	43
2.2	Mining Solution Variation	47
2.2.1	Code Outside the Classroom	47
2.2.2	Code Inside the Classroom	54
2.3	Teaching Principles	63
2.4	Learning through Variation in Examples	65
2.4.1	Analogical Learning	66
2.4.2	Variation Theory	67
2.5	Personalized Student Support and Automated Tutoring	70
3	OverCode: Visualizing Variation in Student Solutions	73
3.1	Introduction	73
3.2	OverCode	76
3.2.1	Target Users and Tasks	77
3.2.2	User Interface	78
3.3	Implementation	84
3.3.1	Analysis Pipeline	84
3.3.2	Variable Renaming Details and Limitations	91
3.3.3	Complexity of the Analysis Pipeline	93
3.4	Dataset	95
3.5	OverCode Analysis Pipeline Evaluation	98
3.6	User Studies	103
3.6.1	User Study 1: Writing a Class Forum Post	103
3.6.2	User Study 2: Coverage	110
3.7	Discussion	117
3.8	GroverCode: OverCode for Grading	120
3.9	Conclusion	130
4	Foobaz: Feedback on Variable Names at Scale	131
4.1	User Interface	133

4.1.1	Producing Stacks and Common Variables	134
4.1.2	Rating Variable Names	135
4.1.3	Making Quizzes	136
4.2	Evaluation	140
4.2.1	Datasets	140
4.2.2	Teacher Study	141
4.2.3	Student Study	146
4.3	Limitations	151
4.4	Conclusion	151
5	Learnersourcing Debugging and Design Hints	153
5.1	System Design	155
5.1.1	Design Questions	155
5.1.2	Self-Reflection Workflow	158
5.1.3	Comparison Workflow	159
5.2	User Interfaces	159
5.2.1	Dear Beta	161
5.2.2	Dear Gamma	163
5.3	Evaluation	167
5.3.1	Dear Beta	167
5.3.2	Dear Gamma	167
5.3.3	Limitations	170
5.4	Results	170
5.4.1	Dear Beta Study	170
5.4.2	Dear Gamma Study	171
5.5	Discussion	175
5.5.1	Answers to Research Questions	175
5.5.2	Lessons for Self-Reflection and Comparison Workflows	176
5.5.3	Generalization	177
5.6	Conclusions	177

6	Bayesian Clustering of Student Solutions	179
6.1	OverCode as Clustering Algorithm	180
6.2	The Applicability of Bayesian Clustering Methods	180
6.3	Clustering Solutions with BCM	182
6.3.1	Preliminary Evaluation	182
6.4	Clustering Solution Components with LDA	184
6.4.1	Preliminary Evaluation	187
6.5	Discussion and Future Work	187
6.6	Conclusion	188
7	Discussion	189
7.1	Design Decisions and Choices	189
7.2	Capturing the Underlying Distribution	191
7.3	Helping Teachers Generate Solution-Driven Content	192
7.4	Writing Solutions Well	193
7.5	Clustering and Visualizing Solution Variation	195
7.6	Language Agnosticism	196
7.7	Limitations	197
8	Conclusion	199
8.1	Summary of Contributions	200
8.2	Future Work	201
8.2.1	OverCode	201
8.2.2	GroverCode	204
8.2.3	Foobaz	205
8.2.4	Targeted Learnersourcing	205
	Appendix	207
A	GroverCode Deployment Programming Problems	207
A.1	Midterm Problems	207

A.2 Final Exam Problems	208
Bibliography	213

List of Figures

1-1	A solution synthesized by OverCode that represents 1538 student solutions.	26
1-2	Three different solutions that exponentiate a base to an exponent. They are all marked correct by the autograder because they pass all the autograder test cases.	27
1-3	Two different approaches to solving the problem. The first disregards teacher instructions to not use equivalent library functions and the second includes the keyword <code>continue</code> in a place where it is completely unnecessary, casting doubt on student understanding of the keyword <code>while</code>	28
1-4	Common and uncommon solutions to exponentiating a base to an exponent produced by students in the 6.00x introductory Python programming course offered by edX in the Fall of 2012.	28
1-5	These two solutions only differ by comments, statement order, formatting, and variable names. Note: <code>wynik</code> means ‘result’ in Polish.	29
1-6	The OverCode user interface. The top left panel shows the number of clusters, called <i>stacks</i> , 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.	33

1-7	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.”	34
1-8	A personalized active learning activity as seen by the student. Students are shown their own solution, 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 labels are revealed, along with teacher comments.	35
1-9	In the <i>self-reflection</i> workflow, students generate hints by reflecting on an obstacle they themselves have recently overcome. In the <i>comparison</i> 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.	37
3-1	The OverCode user interface. The top left panel shows the number of clusters, called <i>stacks</i> , 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 in the platonic solutions and their frequencies.	74
3-2	(a) A stack of 1534 similar <code>iterPower</code> 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. . .	80
3-3	Similar lines of code between two stacks are dimmed out such that only differences between the two stacks are apparent.	80
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.	81

3-5	(a) An example rewrite rule to replace all occurrences of statement <code>result = base * result</code> with <code>result *= base</code> . (b) The preview of the changes in the platonic solutions because of the application of the rewrite rule.	82
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.	83
3-7	A student solution before and after reformatting.	85
3-8	Illustration of running a (dummy) <code>iterPower</code> solution on a test case. . . .	85
3-9	A student solution, its program trace when running on the test case <code>iterPower(5.0, 3)</code> , and the sequence of values extracted from the trace for each variable. . . .	86
3-10	A recursive solution to the same programming problem, its program trace when executing on <code>iterPower(5.0, 3)</code> , and the extracted sequences of values for each variable.	87
3-11	<code>i</code> and <code>k</code> take on the same sequence of values, 0 then 1 then 2, in Solution A and B. They are therefore considered the same <i>common variable</i>	88
3-12	Common and uncommon variables found across the solutions in the previous figure.	88
3-13	The unique variable in the previous pipeline step was originally named <code>exp</code> . OverCode appends a double underscore as a modifier to resolve a name collision with the common variable of the same name. This is referred to as a unique/common collision.	89
3-14	Normalized solutions after variable renaming.	89
3-15	Four solutions collapsed into three stacks.	90
3-16	The top row illustrates a name collision of two common variables, both most commonly named <code>i</code> , across two stacks. The second row illustrates how the colliding variable name in the smaller stack of solutions is modified with an appended character to resolve the collision.	92

3-17	A variable name in a solution with multiple instances of a common variable, i.e., a multiple-instances collision, is modified so that its behavior during execution is unchanged.	94
3-18	A student solution with a unique variable, originally named <code>exp</code> by the student. Since it does not share the same behavior with the common variable also named <code>exp</code> , OverCode appends two underscores to the name of the unique variable, <code>exp</code> , to distinguish it.	94
3-19	Number of solutions for the three problems in our 6.00x dataset.	96
3-20	Example solutions for the <code>iterPower</code> problem in our 6.00x dataset.	96
3-21	Example solutions for the <code>hangman</code> problem in our 6.00x dataset.	97
3-22	Example solutions for the <code>compDeriv</code> problem in our 6.00x dataset.	98
3-23	Running time and the number of stacks and common variables generated by the OverCode backend implementation on our dataset problems.	99
3-24	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 axes corresponding to the size of stacks and the number of stacks are in logarithmic scale.	100
3-25	The two largest stacks generated by the OverCode backend algorithm for the (a) <code>iterPower</code> , (b) <code>hangman</code> , and (c) <code>compDeriv</code> problems.	101
3-26	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.	102
3-27	The number of common/common, multiple instances, and unique/common collisions discovered by our algorithm while renaming the variables to common names.	102
3-28	Screenshots of the baseline interface. The appearance changed between the Forum Post study (left) and the Coverage study (right) in order to minimize superficial differences between the baseline and OverCode interfaces. Functionality did not change.	105

3-29	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.	108
3-30	In (a), we plot the mean number of platonic solutions read in OverCode over time versus the number of raw solutions read in the baseline interface over time while performing the 15-minute Coverage Study task. In (b), we replace the mean number of platonic solutions with the mean number of solutions, i.e., the stack size, they represent. These are shown for each of the three problems in the dataset.	112
3-31	Mean feedback coverage, i.e., the percentage of raw solutions covered, per trial during the coverage study for each problem, in the OverCode and baseline interfaces.	113
3-32	Mean rating with standard error for (a) post-condition perception of coverage (excluding one participant) and (b) confidence ratings that identified strategies were frequently used (1-7 scale).	114
3-33	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.	116
3-34	Mean Likert scale ratings (with standard error) for the usefulness of features of OverCode and baseline. <i>Find</i> refers to the find function within the web browser. <i>Find</i> appears twice in this figure because users rated its usefulness for each condition.	117
3-35	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.	122
3-36	Correct and incorrect solutions, as rendered by GroverCode.	123

4-1	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.”	132
4-2	A personalized quiz as seen by the student. The student is 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 labels are revealed, along with their explanatory comments.	137
4-3	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 <code>i</code> , which iterates through indices of a list, the common variable <code>letter</code> , which iterates through the characters in an input string, and the common variable <code>result</code> , which accumulates one of the acceptable return values, ‘ <code>_i_e_</code> ’.	138
4-4	Number of solutions in datasets.	139
4-5	Subjects in the teacher study, on average, labeled a small fraction of the total names, covering all three provided name categories.	139
4-6	Quiz coverage of student solutions across three datasets.	147
4-7	Variables in <code>iterPower</code> solutions labeled by each teacher.	148
5-1	In the <i>self-reflection</i> workflow, students generate hints by reflecting on an obstacle they themselves have recently overcome. In the <i>comparison</i> 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.	160

5-2 *Dear Beta* serves as a central repository of debugging advice for and by students, indexed by autograder errors. In this figure, there are three learn-sourced hints, sorted by upvotes, for a autograder error on test 33 in the ‘lab5/beta’ checkoff file. 162

5-3 After fixing a bug, students can add a hint for others, addressing what mistake prevented their own solution from passing this particular autograder test. 162

5-4 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. 165

5-5 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. 166

5-6 Between the Dear Beta release date (4/2) and the lab due date (4/10), autograder errors were consistently being entered into the system. The addition of hints followed close behind. 171

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. . . 174

6-1	This interface displays iBCM clusterings of OverCode platonic solutions and affords user feedback on cluster quality. 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 identifies as characteristic of the cluster represented by that prototype. When a user hovers the cursor over a keyword or variable name, e.g., <code>len</code> , it is highlighted in a blue rectangle, indicating that it can be clicked.	182
6-2	Example of a recursive student solution.	185
6-3	Example of a student solution using the Python keyword <code>while</code>	185
6-4	Example of a student solution using the Python keyword <code>while</code> , where the student has not modified any input arguments, i.e., better programming style.	186
6-5	The sequences of variable values recorded by OverCode while executing <code>iterPower(5, 3)</code>	186

List of Tables

3.1	Example: All templates and locations in which the abstract variable <code>exp</code> , the second argument to a recursive <code>power</code> 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.	125
3.2	Number of solutions 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 making inappropriate function calls.	128
3.3	The degree of variation in input-output behavior and statistics about stack sizes.	128
3.4	Statistics about variables renaming based on different heuristics in the GroverCode normalization process.	129
5.1	Breakdown of Dear Gamma hints by type. Students in the Dear Gamma lab study initially received 5 pointing hints (<i>p</i>), followed by 5 pure teaching hints (<i>t</i>), and finally 5 pure bottom-out hints (<i>b</i>), delivered whenever the student was stuck and asked for more help.	172
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) . . .	187

Chapter 1

Introduction

Introductory programming classes are big and hard to teach. Programming classes on some college campuses are reaching hundreds or thousands of students [76]. Massive Open Online Courses (MOOCs) on programming have drawn tens or hundreds of thousands of students [88]. Millions of students complete programming problems online through sites like Khan Academy.

Massive classes generate massive datasets of solutions to the same programming problem. The problem could be exponentiating a number, computing a derivative, or transforming a string in a specific way. The solutions are typically a single function that prints or returns an answer. A terse solution might contain a couple of lines. An excessively verbose solution might contain over twenty lines.

This thesis revolves around clustering and visualizing massive datasets of solutions in novel, human-readable ways. For example, rather than representing solutions as points in a projection of a high-dimensional space into two or three dimensions, OverCode's deterministic unsupervised clustering pipeline synthesizes solutions that each represent entire stacks of solutions. For example, the solution shown in Figure 1-1 represents 1538 solutions [41]. OverCode is the first of several systems for clustering and visualizing solutions and solution variation that were developed in this thesis.

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

Figure 1-1: A solution synthesized by OverCode that represents 1538 student solutions.

1.1 Solution Variation

In this thesis, *solution variation* means three things: correctness, approach, and readability. Since correctness is difficult to prove for an arbitrary piece of code, it is approximated in industry and education by correctness with respect to a set of test cases. In education, the machinery that checks for correctness is often called an autograder.

There is a wide range of solutions that pass all the test cases and are labeled correct by the autograder, but they are not all equally good. Figure 1-2 shows three different solutions of widely varying approach and readability that are correct with respect to the autograder test cases.

Solutions can have different approaches. For example, a student might be subversive and disregard a request by the teacher to solve the problem without using an existing equivalent library function. Or the student might include unnecessary lines of code that reveal possible misconceptions about how the language works. Figure 1-3 gives an example of each.

Approaches can be common or uncommon. One can think of the students submitting solutions as a generative function that produces a distribution of solutions we could characterize and take into account while teaching or designing new course material. Figure 1-4 shows the most common and one of the most uncommon solutions produced by students for a problem assigned in 6.00x, an introductory programming course offered on edX in the fall of 2012. Uncommon solutions may be highly innovative or extraordinarily poor.

Readability is the third critical component of solution variation. Poorly written solutions

Iterative Solution

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

Poorly Written Solution

```
def power(base, exp):
    tempBase=base
    result = base

    if type(base)==int:
        while exp==0:
            result = 1
            print(result)
            break
        exp=exp-1
        while exp >0:
            tempCal=abs(tempBase)
            exp=exp-1
            while exp<0:
                break
            for i in range
                (1,tempCal):
                    result=result+base
                    tempCal=tempCal-1
                    tempRes=base
            base=result
        return(result)

    else:
        result = 1
        while exp > 0:
            result = result* base
            exp = exp- 1
        return result
```

Recursive Solution

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

Figure 1-2: Three different solutions that exponentiate a base to an exponent. They are all marked correct by the autograder because they pass all the autograder test cases.

Subversive Solution

```
def power(base, exp):  
    return base**exp
```

Solution with Unnecessary Statement

```
def power(base, exp):  
    result=1  
    while exp>0:  
        result=result*base  
        exp-=1  
        continue #keyword here does not change execution  
    return result
```

Figure 1-3: Two different approaches to solving the problem. The first disregards teacher instructions to not use equivalent library functions and the second includes the keyword `continue` in a place where it is completely unnecessary, casting doubt on student understanding of the keyword `while`.

Common Solution

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

Uncommon solution

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

Figure 1-4: Common and uncommon solutions to exponentiating a base to an exponent produced by students in the 6.00x introductory Python programming course offered by edX in the Fall of 2012.


```

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

    returns: int or float, base^exp
    """
    result = 1
    while exp > 0:
        result *= base
        exp -= 1
    return result

def iterPower(base, exp):
    wynik = 1

    while exp > 0:
        exp -= 1 #exp argument is counter

        wynik *= base

    return wynik

```

Figure 1-5: These two solutions only differ by comments, statement order, formatting, and variable names. Note: *wynik* means ‘result’ in Polish.

like the one in Figure 1-2 may be both the symptom and the cause of student confusion. Unreadable code is harder to understand and debug, for both teacher and student. In industry, peer-to-peer code reviews help prevent code with poor readability from entering code bases, where it can be difficult and costly to maintain.

Student design choices affect correctness, approach, and readability. Examples include choosing:

- for or while
- $a *= b$ or $a = a*b$
- recursion or iteration

Solution variation is a result of these choices. Even simple differences, like comments, statement order, formatting and variable names can make solutions look quite different to the unaided eye, as shown in Figure 1-5.

1.2 A Challenge and an Opportunity

When teachers may have hundreds or thousands of raw student solutions to the same problem, it becomes arduous or impractical to review them by hand. And yet, only testing for approximate correctness via test cases has been automated. Identifying student solution

approaches and readability automatically are still open areas of research. Given the volume and variety of student solutions, how do teachers comprehend what their students wrote? How do they give feedback on approach and readability at scale?

What value can *only* be extracted from a massive programming class? This thesis focuses on the opportunity posed by exploiting solution variation within large datasets of student solutions to the same problem. With algorithms, visualizations and interfaces, teachers may learn from their students by exposing student solutions they did not know about before. Teachers could find better examples to pull out for discussion. Teachers could write better feedback, test cases, and evaluation rubrics, given new knowledge of the distribution of student solutions across the dimensions of correctness, approach, and readability. And students could be tapped as experts on the solutions they create and the bugs they fix.

1.3 A Tale of Two Turing Machines

A short story about my time as a teaching assistant illustrates some of the challenges posed by solution variation. Students were programming simulated Turing machines, which compute by reading and writing to a simulated infinitely long tape. I struggled to help a student whose approach was not familiar to me. I could not tell if the approach was fatally flawed or unusual and possibly innovative. I did not want to dissuade him from a novel idea simply because I did not recognize it.

The staff server had thousands of previously submitted correct student solutions, but I could not easily see if one of them successfully employed the same approach as the one envisioned by the struggling student. After experimenting with different representations, I found a visualization of Turing machine behavior on test cases that separated 90% of the correct student solutions into two main approaches [38]. The remaining approximately 10% of solutions included less common strategies. Two solutions passed all test cases but subverted teacher instructions. Many teaching staff members were not aware that there were multiple solutions to the problem and that the current test suite was insufficient to distinguish correct

solutions from incorrect ones. At least one staff member admitted steering students away from solutions they did not recognize, but in retrospect may have indeed been valid solutions.

1.4 Research Questions

When a teacher cannot read all the student solutions to a programming problem because there are too many of them,

- **R1** how can the teacher understand the common and uncommon variation in their student solutions?
- **R2** how can the teacher give feedback on approach and readability at scale?
- **R3** ... in a personalized way?
- **R4** ... and how can students do the same?

In this thesis, several systems and workflows are developed and studied to better answer these questions. The first is OverCode, a system for visualizing variation in student Python solutions. The second is Foobaz, a system for visualizing variable name variation and delivering personalized feedback on naming choices. The third and fourth systems are based on two novel, complementary learnersourcing workflows that help students write hints for each other through self-reflection and comparison with other student solutions.

1.4.1 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 for course materials and forum posts and expose corner cases that spur autograder refinements.

OverCode, shown in Figure 1-6 renders thousands of small student Python solutions to the

same problem and makes them explorable. With OverCode, teachers can better understand the distribution of common and uncommon solutions without reading every student solution. The OverCode analysis pipeline uses both static and dynamic analysis to normalize and cluster similar solutions into stacks represented by a single normalized solution. The OverCode normalization process and user interface work together to support human readability and switching between solutions with minimal cognitive load. In two user studies, OverCode helped teachers more quickly develop a high-level view of students' understanding and misconceptions and compose feedback that is relevant to more student solutions, compared to the status quo.

A key, novel component of the normalization process is identifying *common variables* across multiple student solutions to the same programming problem. Common variables are found in multiple student solutions to the same problem and have identical sequences of distinct values when those solutions are executed on the same test cases. During normalization, every common variable in every solution is renamed to the most popular name students gave it.

The normalized solutions encode both static and dynamic information. More specifically, its syntax carries the static information and its variable names encode dynamic information. A single normalized solution can represent an entire stack of hundreds or thousands of similar student solutions.

1.4.2 Foobaz

Traditional feedback methods, such as hand-grading student solutions for approach and readability, are labor intensive and do not scale. Foobaz transforms the output of the OverCode analysis pipeline into an interface for teachers to compose feedback at scale on a critical aspect of readability, variable naming. As shown in Figure 1-7, the Foobaz teacher interface displays the most common and uncommon names for variables in each stack of solutions and allows teachers to label good examples of good and bad student-chosen

The screenshot displays the OverCode interface for the search term 'iterPower'. It is organized into three columns:

- Left Column:** Shows search statistics: 'iterPower' (selected), 'showing 818 total stacks that 3841 total represent submissions'. Below this, it lists 'Largest stack (matching filters)' with 1534 submissions and 'Remaining stacks (matching filters)' with 406 submissions. A 'done' button is visible next to the submission count.
- Middle Column:** Displays the code for the largest stack (1534 submissions):


```
def iterPower(base, exp):
    result = 1
    while (exp > 0):
        result *= base
        exp -= 1
    return result
```
- Right Column:** Shows a list of lines of code from various submissions, with a filter set to 'lines that appear in at least 50 submissions'. The list includes:
 - 77 base = resultB
 - 2905 def iterPower(base, exp):
 - 701 def iterPower(base, expB):
 - 349 def iterPower(base, expC):
 - 51 def iterPower(base, expD):
 - 51 def iterPower(resultB, expC):
 - 55 elif (expC == 1):
 - 525 else:
 - 2473 exp -= 1
 - 281 exp = exp - 1
 - 135 exp = expB
 - 366 expC -= 1
 - 65 expC = expC - 1
 - 63 for i in range(0, expB):
 - 176 for i in range(expB):
 - 52 iC += 1
 - 64 iC = 0
 - 209 if (exp == 0):
 - 210 if (expB == 0):

Figure 1-6: 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.

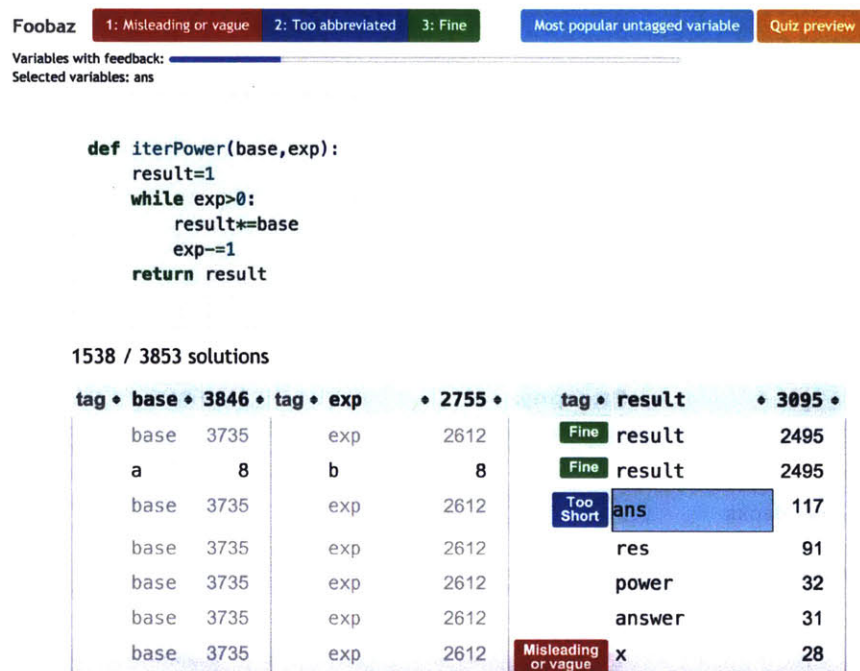


Figure 1-7: The Foobaz teacher interface. The teacher is presented with a scrollable list of normalized solutions, each followed by a table of student-chosen variable names. Some names shown here have been labeled by the teacher as “misleading or vague,” “too short,” or “fine.”

variable names. Students receive these labels in the form of a personalized active learning exercise, where they can learn from the good and bad variable name choices of fellow students.

Once created, these exercises are reusable for as long as the programming problem specification is unchanged. In the first of two user studies, teachers quickly created exercises that could be personalized to the majority of student solutions collected from a Python programming MOOC. In the second of two user studies, fresh students wrote solutions to the same programming problems and received personalized feedback from the teachers in the previous study. A snapshot of this feedback, in the form of an active learning exercise, is shown in Figure 1-8. Foobaz demonstrates that teachers can efficiently give feedback at scale on variable naming, a critical aspect of readability.

You recently wrote the following solution, and we've replaced one variable's name with A:

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

Rate the quality of the following names for the bold symbol A:

Name	Misleading or vague	Too abbrev	Fine
a	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
number	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
out	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
result	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Figure 1-8: A personalized active learning activity as seen by the student. Students are shown their own solution, 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 labels are revealed, along with teacher comments.

1.4.3 Learnersourcing Personalized Hints

Personalization, in the form of one-on-one tutoring, has been a gold standard in educational psychology for decades [10]. It can be hard to get personalized help in large classes, especially when there are many varied solutions and bugs. Students who struggle, then succeed, become experts on writing particular solutions and fixing particular bugs. This thesis describes two workflows built on that insight, shown in Figure 1-9.

Unlike prior incarnations of assigning tasks to and collecting data from students, also known as learnersourcing [60], these workflows collect and distribute hints written only by students who earned the expertise necessary to write them. Both workflows give students an opportunity to reflect on their own technical successes and mistakes, which is helpful for learning [27] and currently lacking in the engineering education status quo [120]. One of the two workflows also systematically exposes students to some of the variation present in other student solutions, as recommended by theories from educational psychology, specifically variation theory [77] and analogical learning theory [67, 74].

These workflows were applied to programming problems in an undergraduate digital circuit programming class with hundreds of students. Field deployments and an in-lab study show that students can create helpful hints for their peers that augment or even replace the personalized support of teachers.

1.4.4 Systems as Answers

While they are not the only answers or complete answers to the research questions that motivated this thesis, these systems shed new light on how to support teachers and students in massive programming classes.

(R1) Understanding Variation at Scale

The OverCode and Foobaz systems both give teachers a better understanding of student solutions and how they vary in approach and readability. OverCode allows teachers to more quickly develop a high-level view of student understanding and misconceptions. Foobaz displays the variety of common and uncommon variable names in student solutions to a single programming problem, so teachers can better understand student naming practices. The clustering and visualization techniques created for both these systems provide some new answers to the research question **R1** for introductory Python problems.

(R2) Feedback on Approach and Readability at Scale

OverCode is also an answer to the research question **R2**. With OverCode, teachers produced feedback on solution approaches that was relevant to more student solutions, compared to feedback informed by status quo tools.

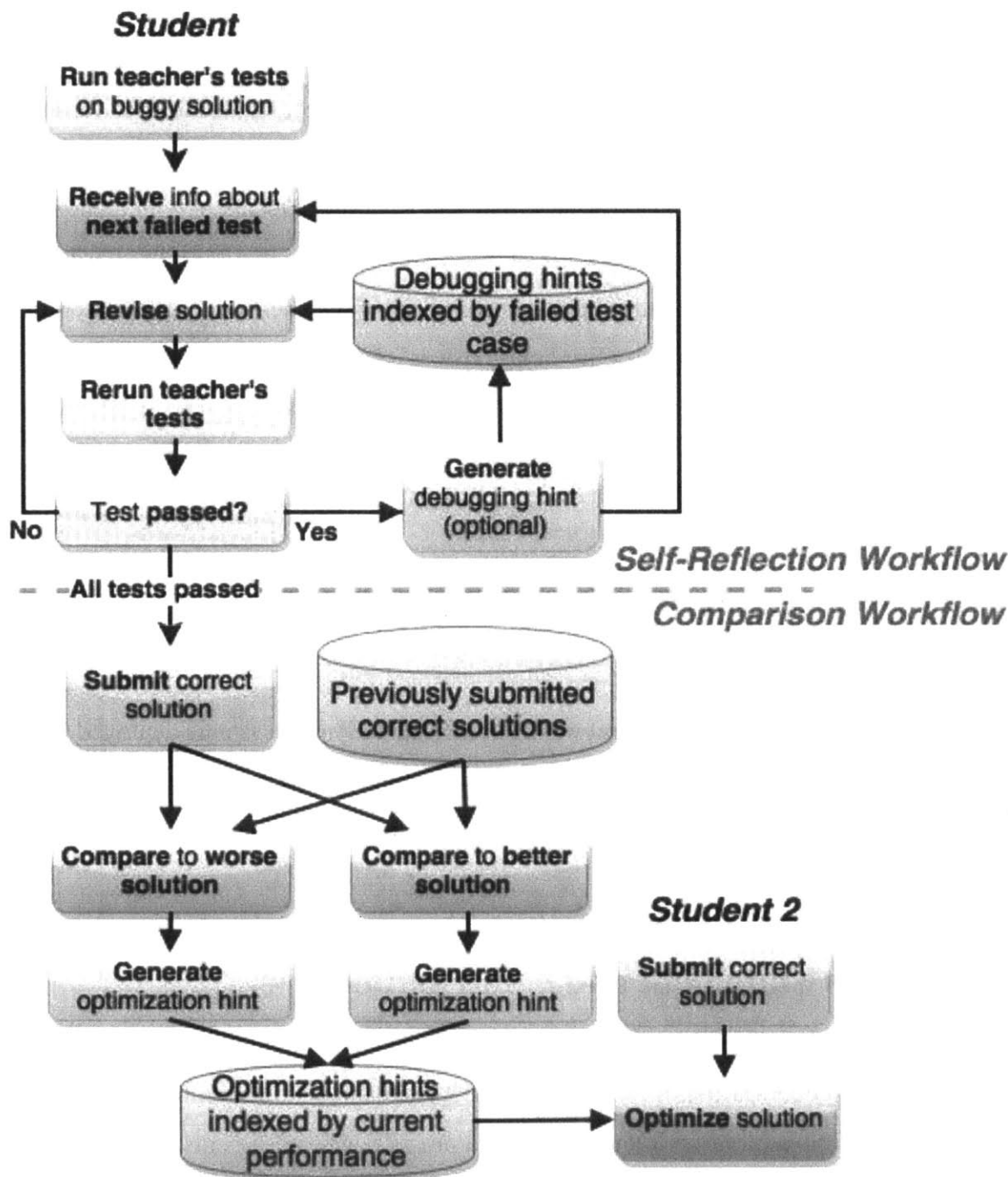


Figure 1-9: In the *self-reflection* workflow, students generate hints by reflecting on an obstacle they themselves have recently overcome. In the *comparison* workflow, students compare their own solutions to those of other students, generating a hint as a byproduct of explaining how one might get from one solution to the other.

(R3) Personalized Feedback on Approach and Readability at Scale

To create Foobaz, the challenge of delivering personalized feedback on approach and readability at scale was narrowed down to just one critical aspect of readability: variable names. In a one-on-one scenario, a teacher helping a student might notice that the student is choosing poor variable names. The teacher might start a conversation about both their good and bad variable naming choices. In a massive classroom where that kind of chat is not possible, Foobaz delivers personalized active learning exercises intended to spark the same thought processes in the student. Foobaz helps answer **R3** by demonstrating how teachers can compose automatically personalized feedback on an aspect of readability to students at scale.

(R4) Personalized Feedback at Scale

The two novel workflows that collect and deliver personalized student-written hints address this final research question in a general way. Their instantiations as systems deployed in an undergraduate engineering course demonstrated that, given appropriate design choices and prompts, students can give personalized feedback in large-scale courses.

1.5 Thesis Statement and Contributions

The systems described in this thesis show various mechanisms for handling and taking advantage of solution variation in massive programming courses. Students produce many variations of solutions to a problem, running into common and uncommon bugs along the way. Students can be pure producers whose solutions are analyzed and displayed to teachers. Alternatively, students can be prompted to generate analysis of their own and others' solutions, for the benefit of 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:

- An algorithm that uses the behavior of variables to help cluster Python solutions and generate the platonic solution for each cluster. Platonic solutions are readable and encode both static and dynamic information, i.e., the syntax carries the static information and the variable name encodes dynamic information.
- A novel visualization that highlights similarity and variation among thousands of Python solutions while displaying platonic solutions for each variant.
- Two user studies that show this visualization is useful for giving teachers a bird's-eye view of thousands of students' Python solutions.
- A grading interface that shows similarity and variation among Python solutions, with faceted browsing so teachers can filter solutions by error signature, i.e., the test cases they pass and fail.
- Two field deployments of the grading interface within introductory Python programming 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 teacher and student 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.

- 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.
- Deployments of both workflows in a 200-student digital circuit programming class, and an in-depth lab study with 9 participants.

1.6 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 various ways that solution variation in large-scale programming courses can be clustered and visualized. These chapters include evaluations on archived data or in the field.

- OverCode (Chapter 3) visualizes thousands of programming solutions using static and dynamic analysis to cluster similar solutions. This chapter also describes GroverCode, an extension of OverCode optimized for grading correct and incorrect student solutions.
- Foobaz (Chapter 4) clusters variables in student solutions 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 learnersourcing workflows that collect and group hints for solutions indexed by autograder test failures or performance characteristics like size or speed. They help students reflect on their debugging or optimization process and

write hints that can help other students with a similar problem.

- Chapter 6 describes Bayesian clustering and mixture modeling algorithms applied to the OverCode pipeline output for extracting additional insight into patterns within student solutions.

Chapter 7 discusses some of the insights that came out of building and testing the systems in this thesis. Chapter 8 outlines avenues of future work on the systems and ideas in this thesis, in combination with the complementary work of others in this space.

Chapter 2

Related Work

Systems that help students in massive programming courses may build on work from program analysis, program synthesis, crowd workflows, user interface design, machine learning, intelligent tutoring systems, natural language processing, data mining, and learning science. This chapter summarizes relevant technical and psychological work that supports the pedagogical value of the systems developed in this thesis and gives context to the technical contributions.

2.1 Clustering

Grouping similar items, i.e., *clustering*, is a fundamental human activity for organizing, making sense of, and drawing conclusions from data. Across many scientific fields, clustering goes by different names but serves a common purpose: helping humans explore, interpret, and summarize data [52].

Clustering is *unsupervised machine learning*. There are no labels, no ground truth. For example, is there one true clustering of all the articles the New York Times ever published?

Humans have goals, and how useful clusters are to them is all that matters. A human can

look at collections of items and group them in various ways. For example, a collection of dogs could be grouped according to their size or the color of their fur. Both clusterings are equally valid, but if a dog walker is deciding which dogs to walk together, clustering by size is more useful.

In order for computers to cluster items, the system designer needs to choose a representation for the items, possibly select or extract features from that representation, choose a clustering method, and assess the validity of its results. The chosen *representation* effects every subsequent step of the clustering process [52]. The representation might be an image or a set of attributes, also called *features*, like weight, height, and fur color. Since the clustering is performed for a human with a goal, the representation ideally captures aspects of the items that are relevant to that goal.

Further customizing the item representation for the human goal, also known as *feature extraction*, is optional but often very helpful. Feature extraction is computing new features about each item from the original representation. These new features may better capture the aspects of items that are relevant to the human goal. For example, if the goal is to partition dogs by fur color and dogs are represented by images, one might extract the most common color in every dog image.

A second optional step is *feature selection*, the process of determining what features can be ignored because they are irrelevant to the human goal. For example, if the goal is to partition dogs by fur color and dogs are represented by collections of features, then only the fur color feature may be selected, ignoring height and weight.

Hopefully, after choosing a representation and possibly performing feature selection and extraction, some items are closer to each other than they are to others, with respect to some aspect of the items the human cares about. If a computer is to verify this, one needs to define exactly what the *closer* means. The computer needs a human to define a *distance function* that quantifies how close or far items are from each other.

The next step is choosing a *clustering method*. Clustering is finding clusters of items that

are closer to each other than they are to items in other groups. There is no one best method for all clustering problems, but some methods produce clusters that support the human's problem-specific goal better than others. Many methods require the human to provide additional parameters, such as how similar two items need to be to be grouped together or how many clusters to look for.

Some methods have an *objective function* that quantifies some desirable characteristic of clusters to maximize or some undesirable characteristic of clusters to minimize [52]. One objective function sums up how different each item is from its cluster center. Presumably, the smaller the differences, the better the clustering. Specific clustering algorithms may define different centers for each cluster, tally up the differences in different ways, and follow different processes to minimize the objective function.

Some methods are more easily characterized by their process than their objective function. For example, hierarchical clustering methods are characterized as either:

- *agglomerative* or bottom-up, by merging individual items into small clusters, then merging small clusters into larger clusters, etc., or
- *divisive* or top-down, by splitting the entire collection of items into clusters, then splitting those clusters into further clusters, until all clusters only contain one item [134].

The objective functions used in these methods help determine which clusters to merge or split.

Some statistical clustering methods, such as mixture models, assume that the items are samples from underlying, unseen distributions of items. By making some assumptions about what those distributions might look like, an algorithm can map items to clusters to maximize a statistical measure of fit between the observed data and the unobserved distributions.

The types of clusters produced by clustering algorithms can be partitioned in several ways [131], such as:

- *Hard* versus *soft*: If each item is mapped to a single cluster, the method is hard. If

- items can partially belong in different degrees to multiple clusters, the method is soft.
- *Partition vs. overlap vs. hierarchy*: If each item belongs to one and only one cluster, the method is producing strict partitions. If each item belongs to one or more clusters, than the method is producing overlapping clusters. If each item belongs to one cluster and clusters are strictly contained by other clusters, the method is hierarchical.

Some methods do not produce clusters of items, but rather clusters of components within items. For example, Latent Dirichlet Allocation (LDA) [9] does not cluster entire documents. It produces a hard, strict partition of words into different clusters, typically called *topics*. At the level of documents, however, this can look like a soft clustering: an individual document might be 50% hospital-related words, 30% government-related words, and 20% negotiation-related words.

Clusters are hard to objectively evaluate. It is expensive and time consuming to bring in humans with goals to evaluate how useful the clusters are to them. *Cluster validation* refers to the metrics used to approximate the quality of clusters with little or no human input [134]. These metrics can be the basis of choosing one method over another, one parameter value over another, or one clustering over another quickly but approximately. Note that a clustering may have good quality scores with respect to those cluster validation metrics, but if it is difficult for the human to understand what each cluster contains, they may have trouble using it to achieve their goals. Therefore, there are a growing number of methods that are designed with *interpretability* in mind.

Another way to produce more helpful clusterings is to involve the human in an *interactive* process. The clustering algorithm produces clusters and gives the human one or more ways to give feedback. Feedback could be in the form of indicating what is good or bad about this clustering. Feedback could be the human reaching in and directly moving an item from one cluster to another. The method reruns, based on this new information, and produces a new clustering for the human to evaluate and give feedback on.

There are thousands of published papers and working systems that use combinations of these properties and strategies. When comparing two systems that cluster, it is helpful to

break the comparison down by representation, features, distance function, objective function, and algorithm they use as well as the type of clusters they produce. In subsequent sections, this language will be used to help describe the many different code analysis and clustering processes found in related work.

2.2 Mining Solution Variation

One important aspect of both engineering and design is that there are multiple ways to solve the same problem, each with their own trade-offs. The consequences of these design choices can be significant. Suboptimal solutions, including poorly designed code, can have high personal, safety, and economic costs.

It may be possible to learn good and bad practices from large collections of previous solutions by looking at common and uncommon choices and making judgements about their successes and trade-offs. In order to understand the space of current solutions, there are many questions one could ask. In the face of a common design choice, what do designers most commonly pick? Has this changed over time? What are popular design alternatives? What are examples of design failures that should be learned from and never repeated? What are examples of design innovations that are clearly head-and-shoulders above the rest? The answers to these questions could fuel education about designing solutions that complies with variation theory recommendations. OverCode, Foobaz, and the comparison learnersourcing workflow try to answer these questions for student solutions written in code.

2.2.1 Code Outside the Classroom

Web pages

Ritchie et al. [97] describe a user interface for finding design examples from a curated database of web pages. Their 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. Their system indexes web page 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. [65] defined *design mining* as “using knowledge discovery techniques to understand design demographics, automate design curation, and support data-driven design tools.” Their work goes beyond searching and filtering a gallery of hundreds of curated web pages. 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 vector of descriptive features is computed for each DOM node in each page.

Webzeitgeist enables users to ask and answer questions like the following, with respect to a large web page repository. What are all the distinct cursors? What are the most popular colors for text? How many DOM tree nodes does a typical page have? How deep is a typical DOM tree? What is the distribution of aspect ratios for images? What are the spatial distributions for common HTML tags? 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 web pages 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.

Android Apps

Shirazi et al. [104] and Alharbi and Yeh [1] describe automated processes for taking apart and analyzing Android app code as well as empirical analyses of corpora of Android apps available from 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 captured 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 corpora, 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?

Open-Source Code Repositories

Ideally, code is not just correct, it is simple, readable, and ready for the inevitable need for future changes [34, 92]. 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. The OverCode and Foobaz pipelines are optimized for user

interfaces that help users answer design mining questions about their student solutions and put the code front and center.

Code, like natural language, exhibits regularity. Hindle et al. [48] argue that the regularity of code may be even more true for code than for natural language. Allamanis and Sutton [2] observe that some syntactic fragments serve a single semantic purpose and recur frequently across software projects. Fast et al. [35] 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 [83]. I will borrow an example from Allamanis and Sutton: [2]

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

Fast et al. [35] 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.

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 n-gram language models built by analyzing corpora 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 an 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 [2] 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 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. [35] 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. [3] observed that common behavior is often correct. Based on that observation, they use probabilistic learning from program execution traces to infer the program’s 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 usage. 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-medoids 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 [53]. 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 is automatically determined to be an implementation rule that all names in the corpus should follow. When run on a large corpus of Java projects, this analysis pipeline found a variety of naming bugs.

Fast et al.'s Codex [35] 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. [35] and Høst and Østvold's [49, 50] work in providing feedback on variable names because all solutions are known to address the same programming problem.

2.2.2 Code Inside the Classroom

The common purpose of the code in a corpus of student solutions to the same programming problem almost certainly contributes to the regularity already found in code from large corpora of open source projects. However, student-written code may not exhibit the same regularity as code written by software developers contributing to an open source project. In other words, the regularity of student solutions to the same programming problem that comes from sharing a common purpose may be counter-balanced by the variety of student coding styles.

It is easier to run student solutions than code in open source projects. Teacher-designed tests already exist, some available to and some hidden from students while they developed their solutions, and some generated on the fly through fuzz testing [113]. This means that dynamic analysis can complement the static analysis of solutions.

Also unlike corpora of open source code, the solutions in large corpora of student code often require feedback, so that the author can learn. Automated feedback on solutions to programming problems 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. Some teachers hand-review hundreds of exam solutions because the autograder output is not sufficient to assign grades.

When a course has 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, clone detection, and mixture modeling methods.

Regularizing Student Solutions

Solutions to the same problem can have different syntax but common semantics. Semantics-preserving transformations can be used to identify and regularize semantically equivalent code. This can include standard compiler optimizations, such as dead code removal, constant folding, copy propagation, and the inlining of helper functions [100]. 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 normalization or standardization, has been used for a variety of applications, including detecting clones [5, 55], diagnosis of bugs in student-written programs [135], and self-improving intelligent tutoring systems [100].

Semantics-preserving transformations will not change the *schema(s)* within a solution. A schema 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.

Nguyen et al. [88] mines probabilistic semantically equivalent AST subtrees from a corpus of solutions to the same problem. The equivalences are probabilistic because the subtrees are only verified to be semantically equivalent with respect to the problem and the specific test cases provided. OverCode uses a similar technique for identifying common variables across solutions.

In OverCode and Foobaz, variables are carefully tracked and strategically renamed, rather than replaced by generic placeholders. This normalization step 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.

Features and Distance Functions for Student Solutions

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. [31] catalog additional distance measures that are potentially helpful for clustering interactive programs.

Just as the authors of *Webzeitgeist* defined sets of features to be computed for each node in a web page DOM [65], there are many sets of features used to estimate program similarity in the literature. Aggrawal et al. [114], Elenbogen and Seliya [33], Roger [102], 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 static or dynamic features. Some include both.

Static features include counts of various keywords and tokens in the solution, 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* [23] used the ABC score, a weighted count of assignments, branches, and conditional statements in a block of code.

Dynamic features include data collected from running a solution on each test case, e.g., the solution output and whether or not it is correct with respect to the teacher 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 problems collected from

the original Machine Learning MOOC offered by Stanford, 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, the authors acknowledge that many different mistakes can produce the same output vector.

Variable behavior is a 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* [32]. 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 known in the literature as role of variables theory.

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.

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 normalized in a novel way by those identified variables and (3) the solution outputs in response to each test case. Gulwani et al. [45] uses the same variable value tracing method to cluster solutions, augmenting it with information about control flow structure to overcome syntactic differences between solutions.

Statistical natural language processing techniques can also be applied to code, preferably after preprocessing. For example, Biegel et al. [6] 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. In other words, it is the Jaccard coefficient of the two solutions' w -shinglings. The resemblance distance is defined as $1 - r$, which 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 deviations from corpus trends [36].

CCFinder [55] and MOSS [106] 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.

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 [101, 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 cost function.

While a naive TED algorithm scales very poorly with tree size, an optimized TED algorithm [107] makes this computation more feasible. The optimized algorithm 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 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. [102]. Yin et al. [136] 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.

Clustering Student Solutions

Clustering student solutions can be difficult. Teachers may not agree on which solutions are closest to each other [36]. Rogers et al. [102] found that official graders for a large programming course agreed on solution clusters only 47.5% of the time even when there were only 3 possible clusters to choose from. In spite of the lack of agreement across some expert code evaluators, several research efforts have focused on automatically clustering solutions.

Luxton-Reilly et al. [75] suggest 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. [75] develop a hierarchical clustering taxonomy for types of solution variations, from high- to low-level: structural, syntactic, or presentation-related. The structural similarity between solutions 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. Variation in presentation, 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.

Other systems rely on clustering algorithms applied to solutions whose pairwise distances are determined by TED scores. AutoStyle [23] uses the OPTICS clustering algorithm to cluster solutions based on normalized TED scores. Huang et al. [51] and Rogers et al. [102] 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. [43] use the Relational Neural Gas technique (RNG) to cluster graded solutions and find solutions, called prototypes, that can represent entire clusters. Feedback on new solutions is provided by highlighting the differences between the new solution 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 solutions are clustered, and these clusters help identify problems in new solutions. In contrast, OverCode, Foobaz, and GroverCode process ungraded solutions and help staff compose feedback for the whole class or assign grades or feedback to a whole body of solutions at one time. OverCode and GroverCode do highlight differences between solutions to help pinpoint problems, although it is staff, rather than students, who view these differences.

Inconsistent holistic grades could be explained by teachers relying on different internalized rubrics. 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 components that teachers are

interested in, similar to what was done for web pages in the Webzeitgeist dataset by Lim et al. [68]. (2) Model solutions as mixtures of components using mixture modeling. They have already been applied to source code and student solutions to open-response mathematical questions [7, 71]. (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. [88] did for student solutions and Fast et al. [35] did for general open source code.

Visualizing and Interacting with Student Solutions

There are several existing visualizations or interfaces that help teachers and students understand how solutions vary within a large corpus of solutions to a common problem. A common design choice is to map each solution to a point in some feature space. Huang et al. [51], Rogers et al. [102], AutoStyle [23], and Cody [38] all use this strategy.

Ned Gulley designed solutions maps for Cody, a Matlab programming game¹, to help users pick and compare pairs of solutions from hundreds of solutions to the same programming problem. The solution map plots each solution as a point against two axes: time of solution 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 [38]. 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. [102] 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. [102] built a grading interface on top of this clustering process. Graders graded one solution at a

¹mathworks.com/matlabcentral/cody

time, grouped by cluster.

AutoStyle [23] 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 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.

Tools that are not built for representing an entire corpus, 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.

The OverCode interface puts the code front and center, synthesizing platonic solutions that represent entire stacks of solutions, 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. OverCode was also inspired by information visualization projects like WordSeer [86, 87] and CrowdScape [103]. WordSeer helps literary analysts navigate and explore texts, using query words and phrases [85]. CrowdScape gives users an overview of crowd-workers’ performance on tasks.

More generally, several 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 [4] 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].

2.3 Teaching Principles

This section describes ideas and techniques from educational psychology and the learning sciences that influenced this thesis work. The systems in this thesis are designed to support teachers and students in massive classrooms. For example, one way to support teachers is to give them a better idea of the solution space, so they can better help a student one-on-one. Another way to support teachers is to deploy personalized prompts that mimic what the teacher might have said to the student if they could interact one-on-one.

One-on-one or small group 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. This became known as Bloom’s two sigma problem.

Foobaz and the learnersourcing workflows explicitly prompt students to reflect and generate explanations on their own. These explanations are helpful to others, but they are also helpful to the student who generated them. These explanations, sometimes called self-explanations, foster the integration of new knowledge. Effective tutors encourage self-explanation by prompting students with questions like *Why?* and *How?* [21]. Students of tutors who

fostered self-explanations had learning gains similar to those whose tutors provided their own explanations and feedback[22].

Self-explanations are a form of reflection, which is a critical method for triggering the transformation from conflict and doubt into clarity and coherence [27]. Turns et al. [120] argue that the absence of reflection in traditional engineering education is a significant shortcoming. This thesis contributes systems designed, in part, to address that shortcoming.

OverCode and Foobaz are designed to help teachers give faster and more personalized feedback to massive numbers of introductory programming students. Rapid personalized feedback supports learning in foundational engineering classrooms [20].

Rapid feedback is also a critical part of deliberate practice, a targeted form of concentrated practice that helps build a specific skill. Deliberate practice is goal-directed, effortful, 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 [42]. 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. 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 [26].

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. Following this pattern of instruction, Foobaz introduces a new way for teachers to provide systematic personalized feedback on good and bad variable names. The student can use this feedback to improve their own solutions and build up their own mental model about variable names are good and bad.

The comparison learnersourcing workflow was designed with zones of proximal development (ZPD) and scaffolding in mind. The concept of the ZPD was first introduced in the mid-1920's by the Soviet psychologist 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.

The comparison workflow dictates showing students better and worse solutions, relative to the solution they generated on their own. Some solutions are optimized in ways that may make them difficult to understand for a student who struggled just to make a working, non-optimized solution. Teachers who use the comparison workflow can decide whether students are prompted to reflect on slightly better and worse solution or radically better and worse solutions. Ideally, these better and worse solutions are not so different, they are outside a student's theoretical ZPD.

Wood et al. [132] 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 [128].

Formative feedback [108] can be helpful as part of the scaffolding. It should non-evaluative, supportive, timely, and specific. It usually arrives as a hint, an explanation, or a verification based on the student answer. Likewise, the comparison workflow, identifies where a student solution is on the spectrum of optimality and prompts the student to reflect on a better solution, such as the next most optimal one.

2.4 Learning through Variation in Examples

The work in this thesis is deeply influenced by multiple theories about the role of variation in learning. Designing sets of examples that illuminate an object of learning can have significant effects on how students understand, generalize, and transfer their learning to new contexts. This section summarizes these theories and how they have been applied in studies of human

learning.

Concrete examples of an object of learning—like how to apply an appropriate statistical test in a statistics word problem—vary in ways that may be superficial or fundamental. In the language of educational psychologists, these are often called surface and structural features [94]. A simple compare and contrast exercise when solving equations [98] or examining case studies in negotiation [74] 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 [90, 122]. Several educational models, e.g., variation theory and the 4C/ID Model [121], build on the value of variability by suggesting specific ways for how it should be deployed in the classroom. The three 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.

2.4.1 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 [67, 74]. 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, 74].

Kolodner [62] suggests creating software tools that align examples to facilitate analogical learning. This thesis is, in part, a response to this suggestion. The comparison learnersourcing workflow pairs solutions the student wrote themselves with solutions that are novel to

them. They are prompted to compare the solutions and write a hint for future students.

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 [29]. 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 [66]. 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 [73]. 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.

2.4.2 Variation Theory

Variation theory (VT) [77] views learning and understanding as discernment, specifically of the various features of objects and concepts that define what they are. An aphorism captures the ideas at the heart of variation theory 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 is built on the understanding that learning is not possible unless the learner can discern what the object of learning is [77]. Discernment is not possible without experiencing variation in the object of learning and the world in which it is situated [79]. Dimensions of variation are described by features [70]². Some feature values are irrelevant, while critical feature values collectively define the object of learning.

Through the lens of machine learning, VT asserts that human learning can suffer from overfitting for some of the same reasons that machine learning algorithms do. 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.

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. [77] identify four patterns of variation: *contrast*, *separation*, *generalization*,

²In variation theory literature, the nomenclature is similar but distinct from that of machine learning: features are referred to as aspects and feature values are referred to as features.

and *fusion*. The contrast pattern of variation includes examples that differ by feature values that determine whether each example represents the object of learning. 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. The generalization pattern highlights, through contrast, what about an object of learning remains constant while certain features vary. Using this pattern of variation, a child learning the concept of three might be introduced to different groups of three, e.g., 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 [89], vocabulary learning [30], the color of light [69], mathematics [82], chemical engineering [14], Laplace transforms [16], supply and demand [78] and computing education [115]. It has been the subject of a government-funded three-year longitudinal study in Hong Kong, with promising results [72].

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.

2.5 Personalized Student Support and Automated Tutoring

Human tutoring can be very effective [10] but expensive to scale up. Effective human tutors often have characteristics described by Lepper and Wolverson'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 [133].

Several types of solutions have been deployed to help students get the personalized attention they need, without relying solely on human tutors. These solutions span the spectrum from recruiting more teaching assistants from the ranks of previous students [91] to automating hints using program synthesis or intelligent tutoring systems.

Singh et al. [110] uses 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 [64], or cues from the student's problem-solving actions [28]. Despite the advantage of automated support, intelligent tutoring systems often require domain experts to design and build them, making them expensive to develop [43]. 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 the point of view of a novice [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 [99] 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 transformations to simplify, anonymize, and order syntax. The solution space can then be used to locate the potential learning progression for a student solution 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.

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 [130], and Klemmer et al. demonstrate peer assessments' scalability to large online design-oriented classes [61]. Peer instruction [81] and peer assessment [119] 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 [63].

Recent work on learnersourcing proposes that learners can collectively generate educational content for future learners while engaging in a meaningful learning experience themselves [60, 84, 129]. For example, Crowdy enables people to annotate how-to videos while simultaneously learning from the video [129]. 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.

Other important forms of personalized support for learning include peer groups, home environment, learning communities, and identity formation [17, 126], but they are outside the scope of this thesis.

Chapter 3

OverCode: Visualizing Variation in Student Solutions

This chapter presents the first example of clustering, visualizing, and giving feedback on an aspect of student solutions. It is adapted and updated from a paper in the Transactions on Computer-Human Interaction (TOCHI) in 2015 [41]. It also includes extensions of that original publication developed in collaboration with Stacey Terman. The text about those extensions is, in part, adapted from her Master's of Engineering thesis [118]. When discussing OverCode, pronouns “we”, “us”, and “our” refer to the coauthors of the TOCHI paper. When discussing modifications and extensions of OverCode, “we”, “us”, and “our” includes Stacey Terman.

3.1 Introduction

Intelligent tutoring systems (ITS), Massive Open Online Courses (MOOCs), and websites like Khan Academy and Codecademy are now used to teach programming courses on a massive scale. In these courses, a single programming problem may produce thousands of solutions from learners, which presents both an opportunity and a challenge. For teachers,

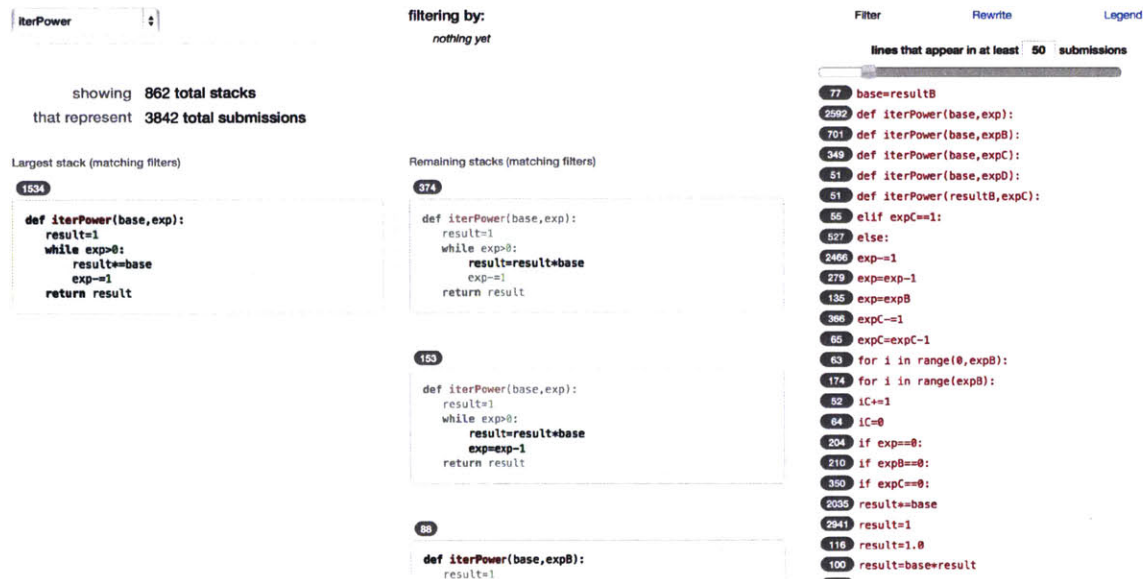


Figure 3-1: The OverCode user interface. The top left panel shows the number of clusters, called *stacks*, and the total number of solutions visualized. The next panel down in the first column shows the largest stack, while the second column shows the remaining stacks. The third column shows the lines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

the wide variation among these solutions can be a source of pedagogically valuable examples [80], and understanding this variation is important for providing appropriate, tailored feedback to students [4, 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 problems 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, 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 [37, 51, 88] use a combination of clustering to group solutions that are semantically similar, and graph visualization to show the variation between these clusters. These clustering algorithms perform pairwise comparisons that are quadratic

in both the number of solutions and in the size of each solution, which scales poorly to thousands of solutions. Graph visualization also struggles with how to label the graph node for a cluster, because it has been formed by a complex combination of code features. Without meaningful labels for clusters in the graph, the rich information in student solutions is lost and the ability of the teacher 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 human-interpretable solution clustering technique. The clustering process is linear in time with respect to both the number of solutions and the size of each solution. The platonic solution that represents each cluster is readable, executable, and describes every solution in that cluster in a deterministic way that accounts for both syntax and behavior on test cases. The platonic solutions are shown in a visualization that puts code front-and-center (Figure 3-1). In the OverCode interface, teachers read through platonic solutions that each represent an entire cluster of solutions. The differences between clusters are highlighted to help teachers discover and understand the variations among 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.

This chapter concludes by presenting 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 developed a better high-level view of the students' understandings and misconceptions. While participants did not 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. These forum posts were relevant to more student solutions when using OverCode as compared to the baseline.

3.2 OverCode

OverCode is an information visualization application for teachers to explore student solutions. OverCode uses the metaphor of solutions stacked on top of each other to denote a cluster of similar solutions. The top solution in the stack is the platonic solution that represents all the other solutions in the stack. The OverCode interface allows the teacher to scroll through, filter, and merge stacks of solutions.

The platonic solutions that represent stacks are normalized in a novel way. Specifically, they have standardized formatting, no comments, and variable names that reflect the most popular naming choices across all solutions. They can be filtered by the normalized lines of code they contain and, through rewrite rules, modified and merged. They are the result of the normalization process applied to all student solutions in the OverCode analysis pipeline.

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 3.3, the backend program analysis pipeline is described in detail.

3.2.1 Target Users and Tasks

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 instructors who compose 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 Student 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 platonic 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 [80], students can learn through concrete examples of these multiple solutions, which vary along different conceptual dimensions.

3.2.2 User Interface

The OverCode user interface is the product of an iterative design process with multiple stages, including paper prototypes and low-fidelity prototypes rendered in a web browser. User interface design experts and programming teachers critiqued initial prototypes. While exploring the low-fidelity prototypes, the 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 of solutions. Side

by side with the largest stack, the remaining 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 platonic 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 3.3. For example, a stack for the `iterPower` problem is shown in Figure 3-2(a). The size of each stack, i.e., the number of solutions in a stack, is shown in a pillbox at the top-left corner of the stack. Stacks are listed in the scrollable second panel from largest to smallest. The solution on the top of the stack is a platonic solution that describes all the solutions in the stack. See Section 3.3 for details on the normalizing process that produces platonic solutions.

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 teachers 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. Double clicking on any stack reveals the raw solutions that belong to that stack. This last feature was added after user testing.

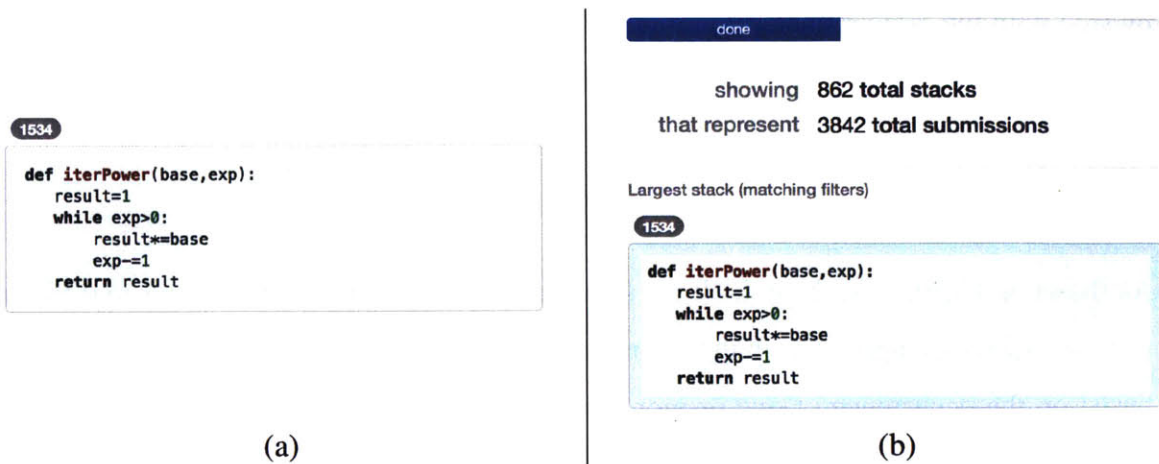


Figure 3-2: (a) A stack of 1534 similar `iterPower` solutions. (b) After clicking a stack, the border color of the stack changes and the done progress bar denotes the corresponding fraction of solutions that have been checked.

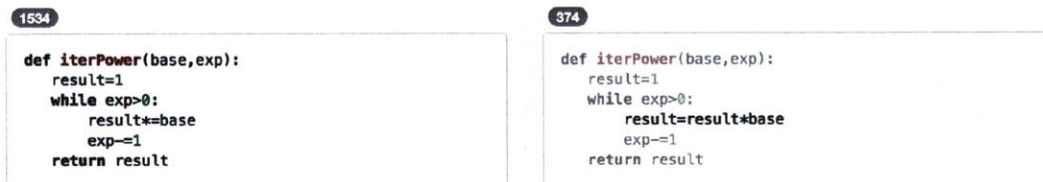


Figure 3-3: Similar lines of code between two stacks are dimmed out such that only differences between the two stacks are apparent.

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 platonic solutions of the two largest stacks. In earlier iterations of the user interface, lines in stacks not shared with the largest stack were highlighted in yellow, but this produced a lot of visual noise. By dimming the lines in stacks that *are* shared with the largest stack, 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 platonic 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`.

lines that appear in at least submissions

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

filtering by:
 for i in range(expB):
 result=1

Remaining stacks (matching filters)

49

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

5

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

(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.

Rewrite Rules

There are often small differences between the platonic solutions that can lead to a large number of stacks for a teacher to review. OverCode provides *rewrite rules* by which teachers 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`, teachers wanted to ignore that difference in order to more easily find other differences.

A rewrite rule is described with a left hand side and a right hand side as shown in Figure 3-5(a). The semantics of a rewrite rule is to replace all occurrences of the left-hand side expression in the platonic solutions with the corresponding right-hand side. As the rewrite rules are entered, OverCode presents a preview of the changes in the platonic solutions as shown in Figure 3-5(b). After the application of the rewrite rules, OverCode collapses stacks that now have the same platonic 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.

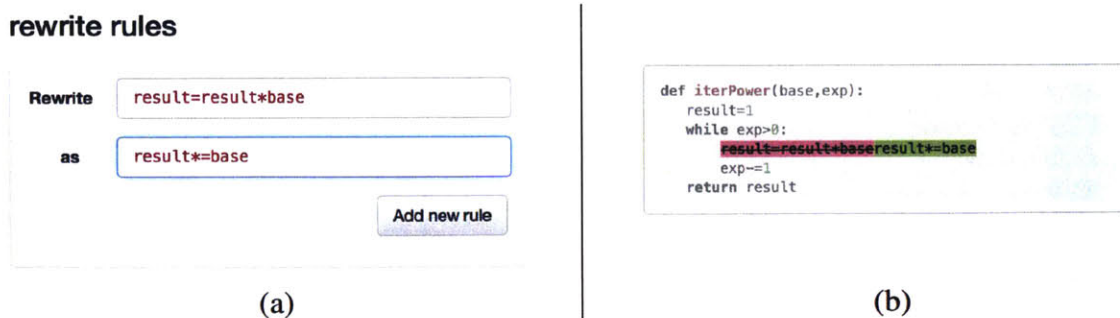


Figure 3-5: (a) An example rewrite rule to replace all occurrences of statement `result = base * result` with `result *= base`. (b) The preview of the changes in the platonic solutions because of the application of the rewrite rule.

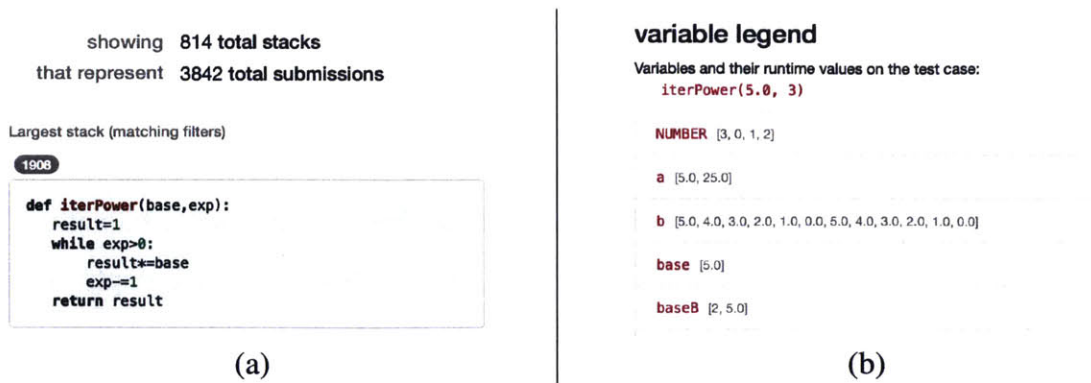


Figure 3-6: (a) The merging of stacks after application of the rewrite rule shown in Figure 3-5. (b) The variable legend shows the sequence of dynamic values that all program variables in normalized solutions take over the course of execution on a given test case.

Variable Legends

OverCode also shows the sequence of values that variables in the platonic solutions take on, over the course of their execution on a test case. As described in Section 3.3, 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 intended to help teachers (1) understand the behavior of each platonic solution without having to mentally execute it on a test case and (2) 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 reinstated in future versions.

3.3 Implementation

The OverCode user interface depends on an analysis pipeline that normalizes solutions in a manner designed for human readability. 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 $f(a, \dots)$, and a set of test cases. 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.

3.3.1 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 package¹ 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. Figure 3-7 illustrates the effect of this reformatting.

2. Execute solutions. Each solution is on the same test case(s). During each step of the execution for each test case, the names and values of local and global variables, and also return values from functions, are recorded as a program trace adapted from the execution logger written by Philip Guo [46]. There is one program trace per solution per test case. Included for the purpose of illustrating this pipeline, the examples in the figures that follow are derived from executing definitions of `iterPower` on a base of 5.0 and an `exp` of 3,

¹Created by Charles Curtis Rhode. <https://pypi.python.org/pypi/PythonTidy/>

Raw Solution A

```
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
```

Reformatted Solution A

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

Figure 3-7: A student solution before and after reformatting.**Student Solution with Test Case**

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

Figure 3-8: Illustration of running a (dummy) `iterPower` solution on a test case.

as illustrated in Figure 3-8.

3. Extract variable sequences. For each variable in a program trace, the pipeline extracts the sequence of distinct values it takes on. Figure 3-9 shows the program trace for a solution run on a test case, as well as the extracted sequences of distinct values for each variable. 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 base^{exp} iteratively, 60 of the 3842 `iterPower` solutions in the dataset were in fact recursive. One of these recursive examples is shown in Figure 3-10, along with the variable sequences extracted.

4. Identify common variables. The pipeline analyzes all program traces, identifying which variable value sequences are identical. Variables that have identical sequences of values across two or more program traces are referred to as *common variables*. Variables which occur in only one program trace are called *unique variables*. This is illustrated in Figures 3-11 and 3-12.

Solution A 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 Solution A

```

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 Solution A

```

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

```

Figure 3-9: A student solution, its program trace when running on the test case `iterPower(5.0, 3)`, and the sequence of values extracted from the trace for each variable.

Recursive Example

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

Program Trace

```
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

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

Figure 3-10: A recursive solution to the same programming problem, its program trace when executing on `iterPower(5.0, 3)`, and the extracted sequences of values for each variable.

Solution B 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 Solution B

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

Solution C 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 Solution C

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

Figure 3-11: *i* and *k* take on the same sequence of values, 0 then 1 then 2, in Solution A and B. They are therefore considered the same *common variable*.

Common variables in solutions A, B, and C:

- 5.0:
 - base (Solutions A, B, C)
- 3:
 - exp (Solutions A, B)
- 1, 5.0, 25.0, 125.0:
 - result (Solutions A, C)
 - r (Student B)
- 0, 1, 2:
 - i (Solution A)
 - k (Student B)

Unique variables in solutions A, B, and C:

- 3, 2, 1, 0:
 - exp (Student C)

Figure 3-12: Common and uncommon variables found across the solutions in the previous figure.

Common Variables, Named

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

Unique Variables, Named

- exp__: 3, 2, 1, 0

Figure 3-13: The unique variable in the previous pipeline step was originally named `exp`. OverCode appends a double underscore as a modifier to resolve a name collision with the common variable of the same name. This is referred to as a unique/common collision.

Normal Solution A

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

Normal Solution B

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

Normal Solution C

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

Figure 3-14: Normalized solutions after variable renaming.

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

There are exceptions made to avoid name collisions. For example, the unique variable in the previous pipeline step was originally named `exp`. As shown in Figure 3-13, OverCode appends a double underscore as a modifier to resolve a name collision with the common variable of the same name. This is referred to as a unique/common collision.

After common and unique variables in the solutions are renamed, the solutions are now called *normal*, as shown in Figure 3-14.

6. Make stacks. The pipeline iterates through the normal solutions, making stacks of

Stack 1 Normal Solution A

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

Stack 2 Normal Solution B

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

Stack 3 Normal Solution C

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

Stack 3 Normal Solution D

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

Figure 3-15: Four solutions collapsed into three stacks.

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 platonic 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 in Figure 3-15, the normal C and D solutions have the 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 teacher. The two solutions are put in the same stack, with one solution arbitrarily chosen as the visible normalized code. However, since Solution A and B use different functions, i.e., `xrange` vs. `range`, and different operators, i.e., `*` vs. `=`, `*`, the pipeline puts them in separate stacks. Some programming problems have no deadline, and new solutions are submitted intermittently. The entire pipeline does not need to be rerun to normalize a new solution and restack all the solutions in the dataset. The pipeline has been split into two phases: solution preprocessing and batch processing. During preprocessing, each solution formatting is

standardized and all comments are removed. It is then executed on one or more test cases, which makes the resulting normalization 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 per 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 normalizes variable names. This must occur as a batch operation because variable renaming depends on the behavior and name of every variable in every solution.

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.

3.3.2 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 the name of a unique variable collides with the name of a common variable.

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

Solution A (Represents 500 Solutions)

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

Normal Solution A (After Renaming)

(unchanged)

Solution E (Represents 250 Solutions)

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

Normal Solution E (After Renaming)

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

Figure 3-16: The top row illustrates a name collision of two common variables, both most commonly named `i`, across two stacks. The second row illustrates how the colliding variable name in the smaller stack of solutions is modified with an appended character to resolve the collision.

be named `iB`.

For example, a subset of `iterPower` solutions created a variable that iterated through the values generated by `range(exp)`. Solution A is an example. A smaller subset created a variable that iterated through the values generated by `range(1, exp+1)`, as seen in Solution E. 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. This is illustrated in Figure 3-16.

Multiple-instances collision. The pipeline identifies variables by their sequence of values. This representation does not preserve information about the timing of when one value transitions to the next, relative to the values of other variables. Without those relative transition times, it is not possible to differentiate between multiple instances of a common variable within a single solution or identically behaving, semantically distinct Boolean flags

in different solutions. This is not a fundamental limitation of this method, because this information is in the program trace. Future versions of OverCode could extract it as well.

Rather than injecting a name collision into an otherwise correct solution, the pipeline preserves the variable name chosen by the student for all the instances of that common variable in that solution. If the student-chosen name collides with any common variable name in any program trace and does not share its sequence of values, the pipeline appends a double underscore to the student-chosen name, so that both the interface and the human reader do not conflate them.

In Figure 3-17, 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`.

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

In Figure 3-18, 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.

3.3.3 Complexity of the Analysis Pipeline

Clustering methods that use pairwise AST edit distance metrics have quadratic complexity both in the number of solutions and the size of the ASTs [51]. The OverCode analysis pipeline has linear complexity in the number of solutions and in the size of the ASTs.

Solution 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:

```
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
```

Solution 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)
```

Figure 3-17: A variable name in a solution with multiple instances of a common variable, i.e., a multiple-instances collision, is modified so that its behavior during execution is unchanged.

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

Figure 3-18: A student solution with a unique variable, originally named exp by the student. Since it does not share the same behavior with the common variable also named exp, OverCode appends two underscores to the name of the unique variable, exp, to distinguish it.

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 testing set equality, which can be performed in linear time by hashing the sets.

3.4 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 programming problems were chosen. This dataset consists of student solutions submitted within two weeks of the release date of each problem. There are thousands of solutions to these problems in the dataset. All solutions marked as correct by an autograder were passed on to OverCode for analysis. The number of solutions analyzed for each problem is shown in Figure 3-19.

- **iterPower** The `iterPower` problem asks students to write a function to compute the exponential `baseexp` iteratively using successive multiplications. This was an in-lecture exercise for the lecture on teaching iteration. See Figure 3-20 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

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

Figure 3-19: Number of solutions for the three problems in our 6.00x dataset.

IterPower Examples

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

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

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

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

Figure 3-20: Example solutions for the iterPower problem in our 6.00x dataset.

week of problem set problems. See Figure 3-21 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 problems. See Figure 3-22 for examples.

The three programming problems represent the typical problems students solve in the early weeks of an introductory programming course. The three problems 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 problems span the second and third weeks of the course in which they were assigned.

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 3-21: 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) < 2:
        return [0.0]
    poly.pop(0)
    for power, value in
        enumerate(poly):
        poly[power] =
            value * (power
                + 1)
    return poly

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):
    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 3-22: Example solutions for the `compDeriv` problem in our 6.00x dataset.

3.5 OverCode Analysis Pipeline Evaluation

We now present the evaluation of the OverCode analysis pipeline implementation on our Python dataset. We first present the running time of our algorithm and show that it can generate stacks within a 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.

Running Time The complexity of the pipeline that generates stacks of solutions grows linearly in the number of solutions as described in Section 3.3.3. Figure 3-23 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 normalize thousands of student solutions and generate stacks within a few minutes for each problem.

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

Figure 3-23: 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 3-24. 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` consist of 97 and 22 solutions, respectively. The two largest stacks for each problem are shown in Figure 3-25.

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 more 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 include line removal rules, so that teachers can remove inconsequential extra lines and cause singleton(s) to merge with other stacks.

The tail of singleton solutions is long, and cannot be read in its entirety by teachers. Even so, the user studies indicate that teachers still extracted significant value from OverCode presentation of solutions. It may be that the largest stacks are the primary sources of

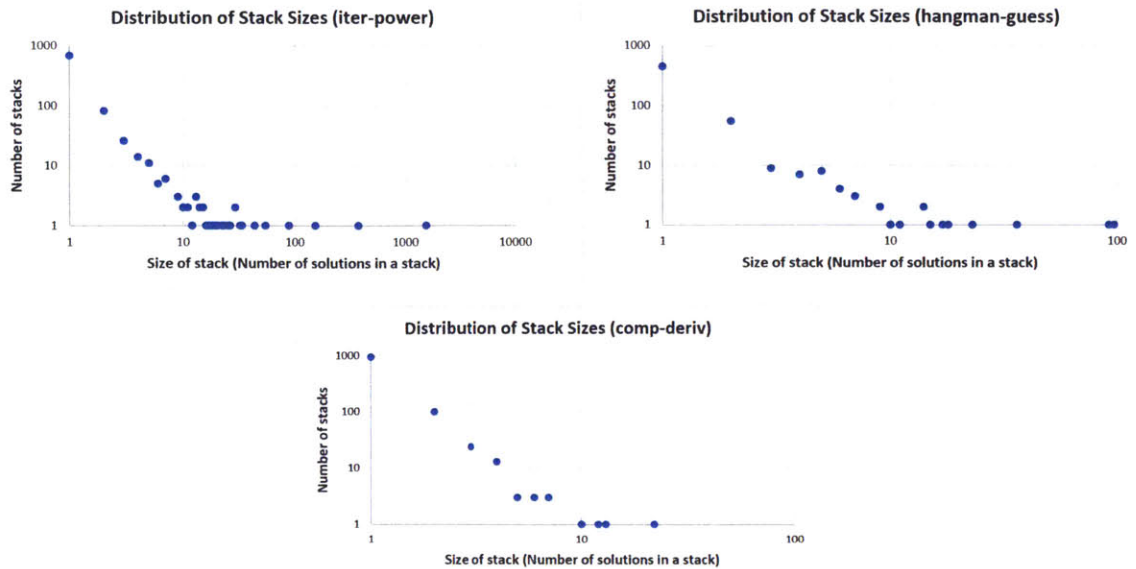


Figure 3-24: The distribution of sizes of the initial stacks generated by our algorithm for each problem, showing a long tail distribution with a few large stacks and a lot of small stacks. Note that the two axes corresponding to the size of stacks and the number of stacks are in logarithmic scale.

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 3-23 and some examples of these common variable names are shown in Figure 3-26. Figure 3-26 also presents the number of times such a variable occurs across the solutions of a given problem, the corresponding variable sequence value on a given test input, and a subset of the original variable names used in the student solutions.

Collisions in Variable Renaming The number of Common/Common, Multiple Instances, and Unique/Common collisions discovered and resolved while performing variable renaming



Figure 3-25: The two largest stacks generated by the OverCode backend algorithm for the (a) iterPower, (b) hangman, and (c) compDeriv problems.

is shown in Figure 3-27. A large majority of the collisions were Common/Common Collisions. For example, Figure 3-26 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	Occurrence 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	['_','_','_','_','_','_','_','_','_','_','_','_','_','_','_']	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 3-26: 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 3-27: The number of common/common, multiple instances, and unique/common collisions discovered by our algorithm while renaming the variables to common names.

3.6 User Studies

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 student solutions. Two user studies 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 solutions 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).

Together, these studies test four hypotheses:

- **H1 Interface Satisfaction** Subjects will find OverCode easier to use, more helpful and less overwhelming for browsing thousands of solutions, compared to the baseline.
- **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 student 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 User Study 1: Writing a Class Forum Post

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 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. This study tests the hypothesis about interface satisfaction (**H1**).

OverCode and Baseline Interfaces

The experiment relies on two interfaces, referred to here as OverCode and the baseline. The OverCode interface and backend are described in detail in Section 3.2. The baseline interface is a single webpage with all student solutions concatenated in a random order into a flat list, as shown in Figure 3-28. This design emulates existing methods of reviewing solutions, and aims to draw out differences between browsing stacked and unstacked solutions. This is analogous to the flat interface chosen as a baseline in another study on interfaces for grading clusters of student work published by Basu et al. [12]. Basu et al. assume that existing options for reviewing solutions are limited to going through solutions one-by-one. This assumption is supported by our pilot studies, interviews with teaching staff, and our own grading experiences. 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 solution at a time. If the solutions can be downloaded locally, some teachers may use a search tool like `grep`. Our baseline allows for search too, through the in-browser `Find` command.

Each solutions is rendered with the same syntax highlighting, background color, and borders in the baseline and OverCode interfaces. However, the solutions shown in the baseline are raw, and the solutions shown in OverCode are normalized. In both interfaces, subjects can use standard web-browser features, such as the within-page `Find` action.

Participants

Participants joined the study by responding to advertisements sent out to an MIT CSAIL-wide email list as well as past and present staff members of MIT introductory programming courses. These individuals were qualified to participate if they have at least one of the following requirements: (1) taught a computer science course (2) graded Python code before, or (3) significant Python programming experience, making them potential future teaching staff. Participants received \$20 for an hour of their time in the lab.

Subjects filled out forms about themselves during recruitment and again at the beginning

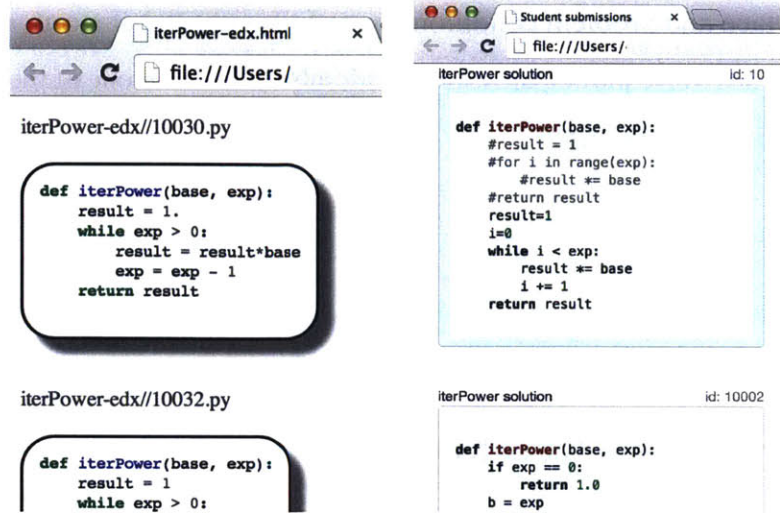


Figure 3-28: Screenshots of the baseline interface. The appearance changed between the Forum Post study (left) and the Coverage study (right) in order to minimize superficial differences between the baseline and OverCode interfaces. Functionality did not change.

of their one-hour in-lab 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 the other half were undergraduates.

Apparatus

Subjects 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.

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 3.4) 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.

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 reviewed an example² of a class forum post that used examples taken from student solutions to explain different strategies for solving a Python problem. For reference, subjects had print-outs of the prompts for each of the three problems in our dataset.

Training The subjects already had extensive experience using web browsers, so 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, familiarized themselves with the interface, and asked 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

²Our 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>)

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 wrote a class forum post on the good and bad ways students solved the problem. The 15-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 test cases. Some solutions may deviate from the problem prompt, and therefore be considered incorrect by teachers, e.g., recursive solutions to `iterPower` when its prompt explicitly calls for an iterative solution. As a secondary task, subjects wrote 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.

Results

Figure 3-29 shows that, compared to the control interface, subjects find OverCode less overwhelming, easier to use, and more helpful for getting a sense of students' understanding. 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 ($W = 66, Z = 3.04, p < 0.001, r = 0.88$). These differences are statistically significant, as computed by the Wilcoxon Signed Rank test with pairing users' ratings of each interface.

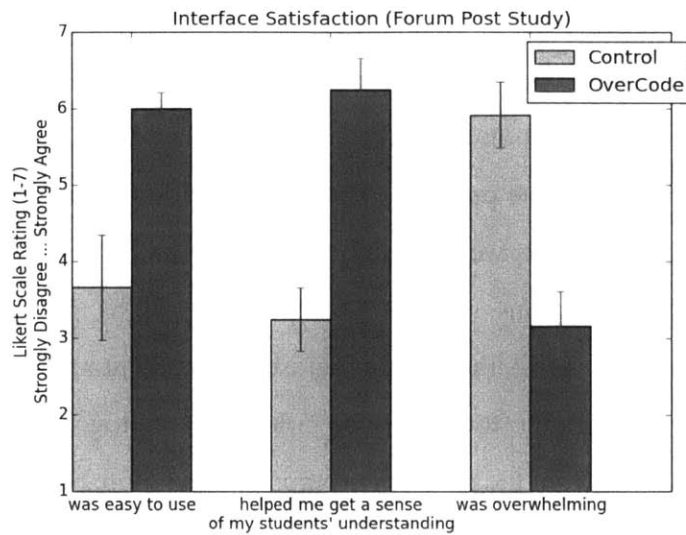


Figure 3-29: H1: Interface satisfaction Mean Likert scale ratings (with standard error) for OverCode and baseline interfaces, after subjects used both to perform the forum post writing task.

This supports **H1**, the hypothesis on interface satisfaction.

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!*
- *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 teacher to change the main stack that is being compared against the others.
- Suggest possible rewrite rules, based on what the teacher has already written, and will not affect the answers on the test case.
- Create a filter that shows all stacks that do *not* 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.”* Participant feedback included comments on the relative goodness of different correct solutions in the dataset. This was a more holistic look at student 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 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.

3.6.2 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 task in the second study was more constrained than the first. Instead of writing a freeform post, subjects identified the five most frequent strategies used by students and rated their confidence that these strategies occurred frequently in the student solutions. These changes to the task enabled us to measure coverage in terms of solutions read, the relevance of written feedback and the perceived coverage of the subject. This study tests the remaining hypotheses on read coverage and speed (**H2**), feedback coverage (**H3**) and perceived coverage (**H4**).

The OverCode and baseline interfaces changed slightly prior to running the second study. Figure 3-28 shows the baseline changes that reduce the differences between the rendering of solutions in the baseline and OverCode interfaces. The OverCode interface was changed to show identifiers next to every stack and solutions so that subjects could provide it when asked.

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). Subjects included 5 graduate students, 6 undergraduates, and 1 independent computer software professional.

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 watched a training video and given time to practice using OverCode features prior to their trial in the OverCode condition.

Task The coverage study task consisted of a more constrained feedback task. Given 15 minutes with either the OverCode or baseline interface, subjects filled out a table, identifying the five most frequent strategies used by students to solve the problem. For each strategy they identified, they filled 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 (one 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 marked solutions or stacks as *read* by clicking on them after they had processed them, even if they did not choose them as representative strategies. We combined this data with logs of subject actions in the interface to quantify read coverage.

Surveys Although we measured interface satisfaction for a realistic task in the forum post study, we also measured interface satisfaction through surveys for this more constrained, coverage-focused task. Subjects filled out a post-interface condition survey in which they rated agreement (on a 7-point Likert scale) with positive and negative adjectives about their experience using the interface, and reflected on task difficulty. At the end of the study, subjects rated 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.

Results

H2: Read coverage and speed Figure 3-30 shows that subjects reading platonic `iterPower`, `hangman`, and `compDeriv` solutions in the OverCode interface covered the equivalent of 64%, 25%, and 14% of all student solutions to those problems (Mann–Whitney $U = 16$, $n_1 = n_2 = 4$, $p < 0.05$). Subjects read through more raw solutions than platonic solutions to the simplest problems, i.e., `iterPower` and `hangman`, but when problem difficulty increased, i.e., `compDeriv`, subjects read through the platonic solutions in the OverCode interface faster than raw solutions. This supports the H2 hypothesis.

H3: Feedback coverage Each subject reported on the five 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

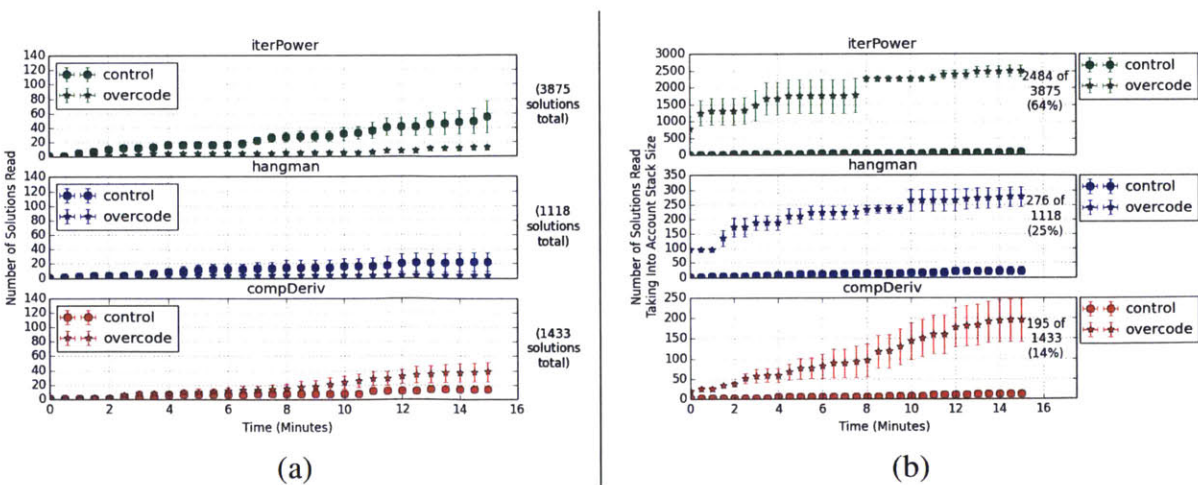


Figure 3-30: In (a), we plot the mean number of platonic solutions read in OverCode over time versus the number of raw solutions read in the baseline interface over time while performing the 15-minute Coverage Study task. In (b), we replace the mean number of platonic solutions with the mean number of solutions, i.e., the stack size, they represent. These are shown for each of the three problems in the dataset.

example using the following process:

- Reduce the quoted code down to only the lines referred to in the annotation. Often, the annotation would focus on a specific feature of the quoted code, which sometimes included additional lines unrelated to the written 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 subject wrote down a stack ID. In the baseline condition, the subject wrote down the solution ID. During post-study analysis, the authors determined which stack the solution is in, as determined by the OverCode analysis 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.

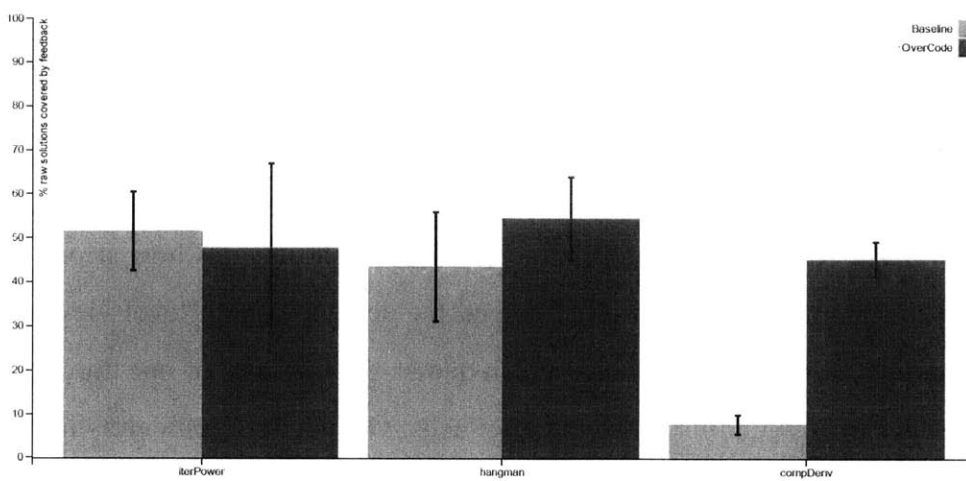


Figure 3-31: Mean feedback coverage, i.e., the percentage of raw solutions covered, per trial during the coverage study for each problem, in the OverCode and baseline interfaces.

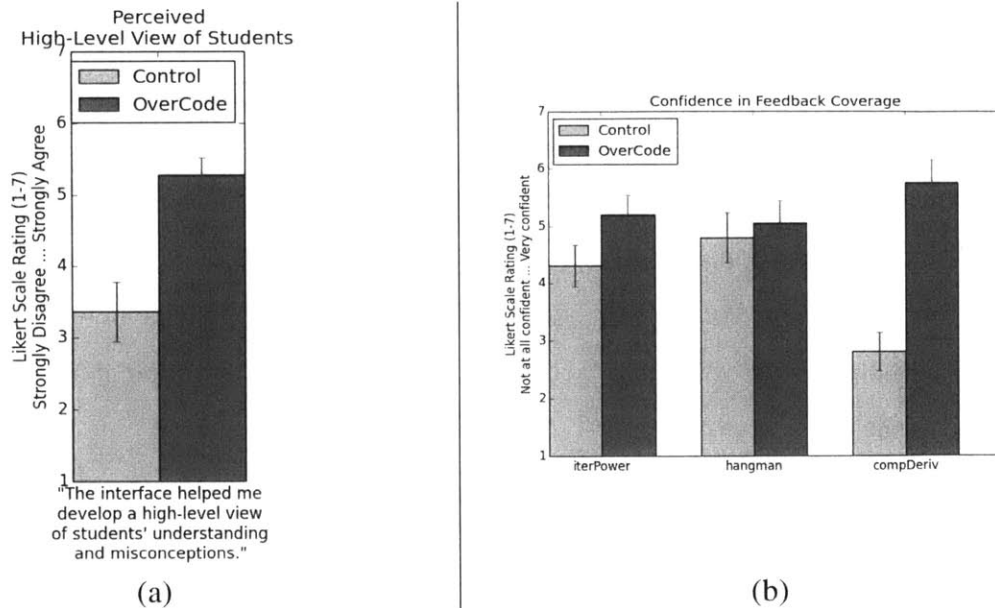


Figure 3-32: Mean rating with standard error for (a) post-condition perception of coverage (excluding one participant) and (b) confidence ratings that identified strategies were frequently used (1-7 scale).

Figure 3-31 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` achieved significantly more coverage of student solutions than when using the baseline interface (Mann–Whitney $U = 0$, $n_1 = n_2 = 4$, $p < 0.05$).

H4: Perceived coverage After using each interface, we asked participants how strongly they agreed with the statement ‘This interface helped me develop a high-level view of students’ understanding and misconceptions,’ which quotes the first part of our third hypothesis. Participants agreed with this statement after using `OverCode` significantly more than when using the baseline interface ($W = 63$, $Z = 2.70$, $p < 0.01$, $r = 0.81$). Statistical significance was computed using the Wilcoxon Signed Rank test, pairing users’ ratings of each interface. The analysis was done for only 11 participants because the data of one

participant was lost. The mean rating with standard error for the responses is shown in Figure 3-32(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 3-32(b). We found that for `compDeriv`, subjects using OverCode felt significantly more confident that their annotations were relevant to many students, compared to the baseline (Mann–Whitney $U = 260.5, n_1 = 18, n_2 = 16, p < 0.0001$).

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 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 for only 11 participants because the data of one participant was lost. The mean ratings (with standard error) for the responses are shown in Figure 3-33.

After the completion of the Coverage Study, participants rated 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 for the responses are shown in Figure 3-33, as well as the standard error.

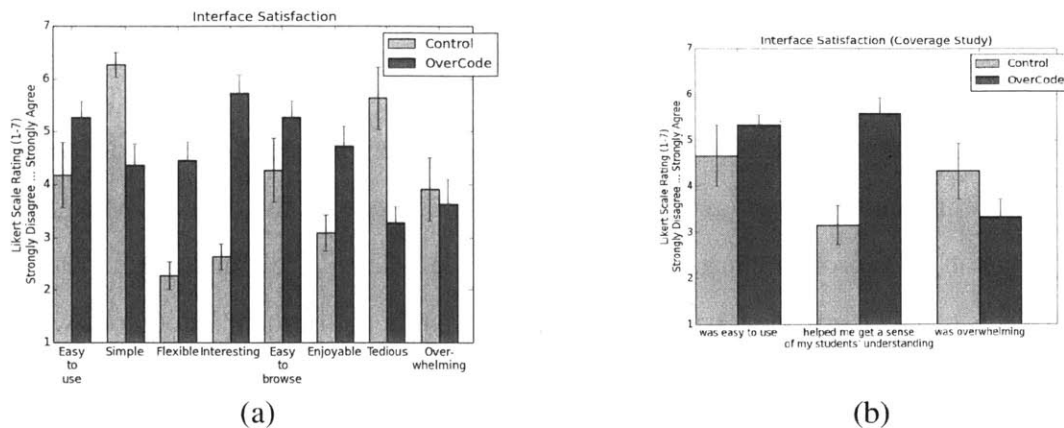


Figure 3-33: H1: Interface satisfaction Mean Likert scale ratings (with standard error) for OverCode and baseline, (a) immediately after using the interface for the Coverage Study task, and (b) after using both interfaces.

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. Statements about these baseline features were mixed in with statements about OverCode features. The OverCode feature of stacking equivalent solutions was found more helpful than the baseline features of in-browser find ($W = 41, Z = 2.07, p < 0.05, r = 0.60$) and viewing raw student solutions, comments included ($W = 45, Z = 2.87, p < 0.005, r = 0.83$). Subjects found both the OverCode feature of variable renaming and previewing rewrite rules significantly more helpful than seeing raw student 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 3-34.

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`

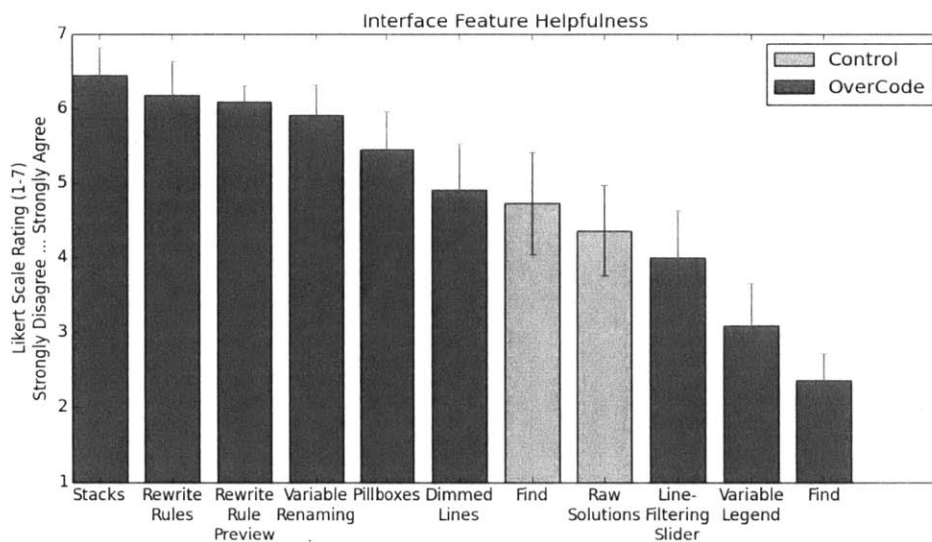


Figure 3-34: Mean Likert scale ratings (with standard error) for the usefulness of features of OverCode and baseline. *Find* refers to the find function within the web browser. *Find* appears twice in this figure because users rated its usefulness for each condition.

or an extremely long expression. Subjects rarely used the filter panel frequency slider (Figure 3-4a) and the variable legend (Figure 3-6b).

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)`. These variations may be considered noteworthy or trivial by different teachers.

3.7 Discussion

A three-part evaluation of the OverCode backend and user interface demonstrates its usability and usefulness in a realistic teaching scenario. Given that the datasets are drawn from the first three weeks of an introductory Python programming course, the 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 student misconceptions or knowledge gaps. Future work will address more complex algorithms and data types.

Read coverage

Subjects covered more student solutions when reading solutions in the OverCode interface (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. The OverCode backend produces platonic solutions that represent many raw solutions, reducing the cognitive load of mentally processing all the raw solutions, including their variation in formatting and variable naming.

Figure 3-30(a) shows that in some cases, subjects 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 ability of subjects to move quickly through the set of solutions, reading as many as 90 solutions in 15 minutes.

Figure 3-30(b) shows the effective number of raw solutions read, when accounting for the number of solutions represented by each platonic solution read 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 includes 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.

Feedback coverage

We also found that subject-written 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.

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 felt 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.

Clarity in Variable Renaming

The modifiers appended to common variables to resolve `Common/Common` collisions caused some confusion. For example, `iB` is a legitimate, but odd, variable name. To indicate that the modifier is not something the student wrote themselves, modifiers are now 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.

3.8 GroverCode: OverCode for Grading

While understanding the contents of thousands of correct student solutions can be helpful in both residential and online contexts, another application of the OverCode 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 test suites can 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 course at MIT, this can be one of the most time-consuming and exhausting parts of teaching the course. It can take a full workday for the entire staff of eight to ten teachers to sit in a room and review several hundred student solutions by hand in order to assign a single numerical grade to each solution.

Stacey Terman, in partial fulfilment of her Master's of Engineering, worked with me to extend OverCode to a new domain, i.e., incorrect solutions, and a new task, i.e., grading hundreds of incorrect solutions by hand [118]. This new version of OverCode for grading is referred to as GroverCode. GroverCode was iteratively designed and evaluated as a grading tool through two live deployments during the Spring 2016 6.0001 staff exam grading sessions. The following tables, figures, and code samples generated from those deployments

are adapted with permission from that thesis.

GroverCode contains two main technical contributions: 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.

User Interface

The GroverCode interface, shown in Figure 3-35, has several additions and distinctions from the OverCode interface. As shown in Figure 3-36, both correct solutions and incorrect solutions are displayed, differentiable by their varied background colors and red X next to every failed test. To better explain each test failure, the actual and expected outputs are displayed beneath each solution. The staff iteratively develop a rubric as they grade. Figure 3-36b is a snapshot of one of these rubrics.

The panel of progress indicators and filters, shown on the left side of Figure 3-35, helps teachers understand the distribution of solution behavior on test cases, track their grading progress, and grade sets of solutions that fail the same test cases one at a time. The rows of aligned sequences of green checks and red dashes represent error vectors, i.e., 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.

The screenshot displays the GroverCode interface for a Python exam problem. On the left, a 'Tests Passed' table shows 10 tests with varying success rates. The main area shows two student solutions for a 'flatten' function. The first solution (id: 106) is highlighted in pink and has a score of 2/10. The second solution (id: 9) is highlighted in white and has a score of 10/10. Below each solution, a list of test cases is shown with their expected and actual results.

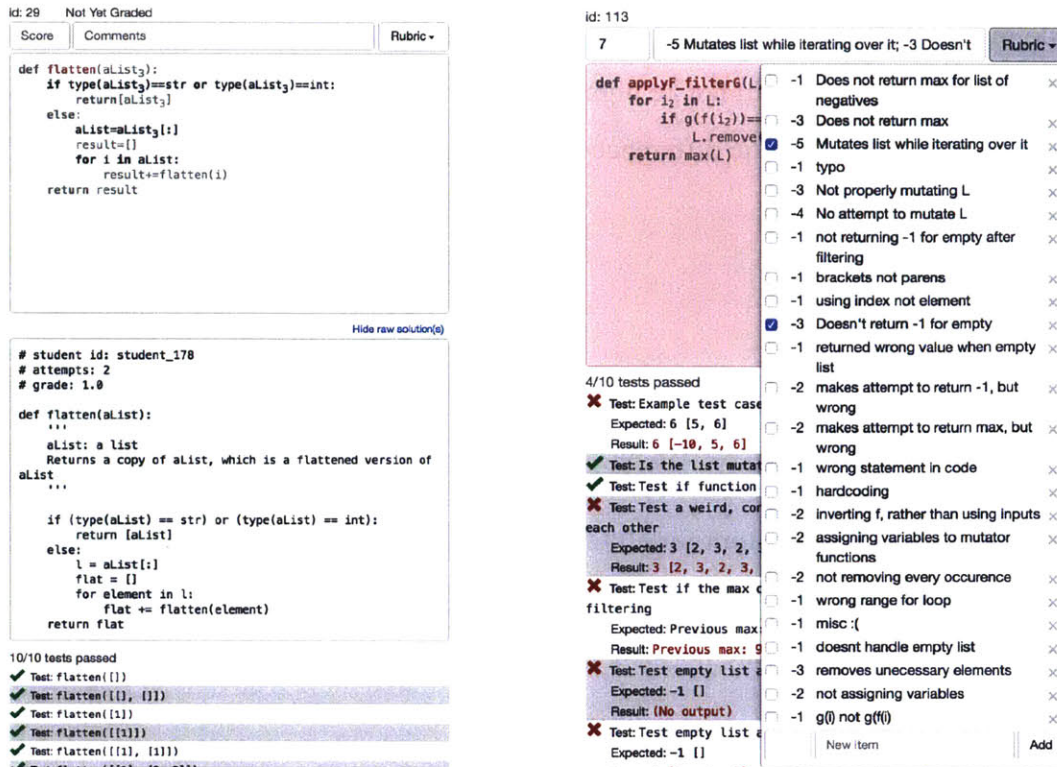
Toggle Between Grading Interfaces	Hide graded solution(s)	id: 106 Not Yet Graded	id: 9 Not Yet Graded																																													
<table border="1"> <thead> <tr> <th>Tests Passed</th> <th># and % Graded</th> </tr> </thead> <tbody> <tr><td>10 ✓</td><td>0 / 77</td></tr> <tr><td>9 ✓</td><td>0 / 2</td></tr> <tr><td>9 ✓</td><td>0 / 1</td></tr> <tr><td>8 ✓</td><td>0 / 2</td></tr> <tr><td>8 ✓</td><td>0 / 1</td></tr> <tr><td>8 ✓</td><td>0 / 1</td></tr> <tr><td>8 ✓</td><td>0 / 1</td></tr> <tr><td>7 ✓</td><td>0 / 17</td></tr> <tr><td>6 ✓</td><td>0 / 1</td></tr> <tr><td>6 ✓</td><td>0 / 1</td></tr> <tr><td>5 ✓</td><td>0 / 1</td></tr> <tr><td>4 ✓</td><td>0 / 7</td></tr> <tr><td>4 ✓</td><td>0 / 1</td></tr> <tr><td>3 ✓</td><td>0 / 9</td></tr> <tr><td>3 ✓</td><td>0 / 1</td></tr> <tr><td>2 ✓</td><td>0 / 21</td></tr> <tr><td>2 ✓</td><td>0 / 7</td></tr> <tr><td>2 ✓</td><td>0 / 2</td></tr> <tr><td>1 ✓</td><td>0 / 4</td></tr> <tr><td>1 ✓</td><td>0 / 3</td></tr> <tr><td>0 ✓</td><td>0 / 8</td></tr> <tr><td colspan="2">0 / 153</td></tr> </tbody> </table>	Tests Passed	# and % Graded	10 ✓	0 / 77	9 ✓	0 / 2	9 ✓	0 / 1	8 ✓	0 / 2	8 ✓	0 / 1	8 ✓	0 / 1	8 ✓	0 / 1	7 ✓	0 / 17	6 ✓	0 / 1	6 ✓	0 / 1	5 ✓	0 / 1	4 ✓	0 / 7	4 ✓	0 / 1	3 ✓	0 / 9	3 ✓	0 / 1	2 ✓	0 / 21	2 ✓	0 / 7	2 ✓	0 / 2	1 ✓	0 / 4	1 ✓	0 / 3	0 ✓	0 / 8	0 / 153		<pre>def flatten(aList3): result=[] if type(aList3)!=list: return[aList3] else: for i in aList3: result+=flatten(i) return result</pre>	<pre>def flatten(aList3): if type(aList3)!=list: return[aList3] else: result=[] for i in aList3: result+=flatten(i) return result</pre>
Tests Passed	# and % Graded																																															
10 ✓	0 / 77																																															
9 ✓	0 / 2																																															
9 ✓	0 / 1																																															
8 ✓	0 / 2																																															
8 ✓	0 / 1																																															
8 ✓	0 / 1																																															
8 ✓	0 / 1																																															
7 ✓	0 / 17																																															
6 ✓	0 / 1																																															
6 ✓	0 / 1																																															
5 ✓	0 / 1																																															
4 ✓	0 / 7																																															
4 ✓	0 / 1																																															
3 ✓	0 / 9																																															
3 ✓	0 / 1																																															
2 ✓	0 / 21																																															
2 ✓	0 / 7																																															
2 ✓	0 / 2																																															
1 ✓	0 / 4																																															
1 ✓	0 / 3																																															
0 ✓	0 / 8																																															
0 / 153																																																
	<p>2/10 tests passed</p> <ul style="list-style-type: none"> ✗ Test: flatten([]) Expected: [] Result: None ✗ Test: flatten([[[]], [[]]) Expected: [] Result: (No output) ✓ Test: flatten([1]) ✓ Test: flatten([[1]]) ✗ Test: flatten([[1], [1]]) Expected: [1, 1] Result: [1] ✗ Test: flatten([[1], [2, 3]]) Expected: [1, 2, 3] Result: [1] 	<p>10/10 tests passed</p> <ul style="list-style-type: none"> ✓ Test: flatten([]) ✓ Test: flatten([[[]], [[]]) ✓ Test: flatten([1]) ✓ Test: flatten([[1]]) ✓ Test: flatten([[1], [1]]) ✓ Test: flatten([[1], [2, 3]]) ✓ Test: flatten([[3], [2, 1, 0], [4, 5, 6]]) ✓ Test: flatten([[[]], [[5]]) ✓ Test: flatten([[1, [2, 3]], [4, 5, 6]]) ✓ Test: flatten([[1, [2, 3]], [[4, 5, 6], [2, 1], [1, [0]]) 																																														

Figure 3-35: 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.

Like OverCode, GroverCode groups correct solutions into stacks. Grades are propagated to all the solutions in a stack. OverCode does not group incorrect solutions into stacks because the stacking process for incorrect solutions is too new and experimental for field deployment in an actual grading session. GroverCode still normalized all solutions, both correct and incorrect.

The grading performed by the 6.0001 staff was intellectually demanding. For example, many staff members attempted to correct incorrect solutions by hand to better understand how far off the student was from a correct answer. When raw solutions are shown in random order, one solution may have a different approach, behavior, and appearance from the next. The transition from grading one solution to the next can mean starting from scratch. In an attempt to minimize the cognitive load of switching from one solution to the next, the GroverCode user interface includes the following features:

- Horizontally aligned solutions for side-by-side comparison
- Easy filtering of solutions by input-output behavior, i.e., their error vectors



(a) A correct solution, its corresponding raw solution, and its performance on test cases, as displayed in GroverCode. (b) An incorrect solution with the rubric dropdown menu open. Text for each of the checked items is automatically inserted in the comment box.

Figure 3-36: Correct and incorrect solutions, as rendered by GroverCode.

- Normalized variable names
- Solutions ordered to minimize the distance between adjacent solutions with respect to a custom similarity metric defined in Stacy Terman’s Master’s of Engineering thesis [118]
- Highlighted lines of code in each solution which differ from the previous solution in the horizontal list

In OverCode, a single-line difference between two solutions that affects variable behavior causes the variable renaming process to propagate naming differences to every line that mentions the affected variables. This non-locality of difference makes it harder to spot where the original syntactic difference between two platonic solutions is. As a result, the rendering of stacks in the GroverCode user interface is slightly different than in OverCode. Even if the normalized 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 original syntactic difference will be highlighted.

Implementation

The GroverCode implementation is a modification of the OverCode pipeline. 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. GroverCode applies this rule to incorrect solutions as well, but only for variables whose behavior matches a common variable in the correct solutions. This step may be error prone. Incorrect solutions are known to be wrong with respect to input-output behavior, the behavior of the variables within them is suspect too, even if it happens to match a common variable in a correct solution.

In incorrect solutions, syntax, e.g., the operations applied to a variable, may be more helpful for normalizing variable names than behavior. For example, in 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 identifying common variables will be thrown off.

The OverCode pipeline is modified for GroverCode to capture this syntactic information. 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:

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

Table 3.1: 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.

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 variable behaviors and their normalized 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]`

Item 1 in this list, the template, captures the syntax of the line, ignoring the variables mentioned in it. Items 1 and 2 in this list together capture the information necessary to reconstruct the original normalized lines in OverCode. 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.

The GroverCode normalization process uses this syntactic information, in addition to behavior, to identify and rename common variables. The template of a line, e.g., `for __ in __:` is the syntax of the line with variable names removed. 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 3.1. 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 normalized, the counts of template-locations associated with each common variable in the correct solutions are used, as described in detail in Terman [118], 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.

Field Deployment

Approximately two hundred students enrolled in 6.0001 during the spring 2016 semester, and nine instructors used GroverCode to grade nearly all student solutions. 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. The midterm problem prompts were:

- `power` (q4): Write a recursive function to calculate the exponential base to the power `exp`.
- `give_and_take` (q5): 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`.
- `closest_power` (q6): 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.

The final problem prompts were:

- `deep_reverse` (q4): Write a function that takes a list of lists of integers `L`, and reverses `L` and each element of `L` in place.
- `applyF_filterG` (q5): 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.
- `MITCampus` (q6): 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`.
- `longest_run` (q7): 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.

For each problem processed during the field deployments, an example of a staff-written correct solution is included in the Appendix.

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. Stacey Terman, the Master's of Engineering student who implemented most of GroverCode, was one of these TAs. GroverCode logged TA grading events, such as adding or applying rubric items and point values to solutions. TAs hand-graded solutions that did not successfully pass through the GroverCode pipeline. The observer (the author of this thesis) took extensive notes during each day-long grading session to capture spontaneous feature requests as well as bugs and complaints.

Pipeline Evaluation

Table 3.2 shows the number of solutions submitted for each problem and the number successfully processed by the GroverCode pipeline. Table 3.3 captures the scale of the variation as well as some clustering statistics. Table 3.4 summarizes the counts of the various mechanisms by which variables in incorrect solutions were normalized.

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

Table 3.2: Number of solutions 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 making inappropriate function calls.

	Quiz			Final			
	q4	q5	q6	q4	q5	q6	q7
Correct solutions	182	160	94	96	49	16	12
	94%	82%	49%	55%	28%	9%	7%
Incorrect solutions	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 with > 1 solution	13	18	1	8	2	0	0
Solutions in stacks	151	94	2	57	5	0	0

Table 3.3: The degree of variation in input-output behavior and statistics about stack sizes.

Discussion

GroverCode was particularly appreciated on simpler problems where GroverCode clustered correct solutions together to be graded by a single action. Under these conditions, the clustering capability GroverCode inherits from OverCode amplified teacher effort.

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

Table 3.4: Statistics about variables renaming based on different heuristics in the GroverCode normalization process.

Variable renaming was, for simpler programming problems, an invisible and possibly slightly confusing helping hand. One grader remarked, out loud: “Why is everyone naming their iterator variable ‘i’?” at which point he had to be reminded of the variable renaming process.

When applied to more complex solutions, the normalization process was at times more harmful than helpful for teacher comprehension. As solutions became more varied and structurally complex, graders started toggling off normalization because the renaming of variables, removal of comments, and formatting standardization was removing clues they needed in order to understand student intent.

The feature for grouping solutions based on their behavior on test cases was appreciated regardless of solution complexity. 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.

3.9 Conclusion

OverCode is a novel system for visualizing thousands of Python programming solutions that helps teachers explore the variations among them. Unlike previous approaches, OverCode uses a lightweight static and dynamic analysis to generate platonic solutions that represent stacks of similar solutions in an interactive user interface. It allows teachers to filter stacks by normalized lines of code and to further merge different stacks by composing rewrite rules. As observed in two user studies with a total of 24 current and potential teaching assistants, OverCode allowed teachers to more quickly develop a high-level view of students' understanding and misconceptions, and provide feedback that is relevant to more student solutions.

GroverCode, an extension of OverCode which handles incorrect solutions, is a new tool for triaging and hand-grading solutions. It was successfully deployed in the field as the tool for grading the midterm and final exams of MIT's 6.0001 course, "Introduction to Computer Science and Programming in Python." Based on teachers' appreciation of the interface over the existing alternatives, GroverCode will continue to be deployed in the field to ease the subjective psychological burden of grading hundreds of Python programs.

Chapter 4

Foobaz: Feedback on Variable Names at Scale

This chapter presents the second example of clustering, visualizing, and giving feedback on an aspect of student solutions, the variety of student-chosen variable names. It is adapted from a paper presented at the ACM Symposium on User Interface Software and Technology (UIST) in 2015 [39].

When providing feedback on the substance and style of student solutions, the status quo is grading by hand. Unfortunately, hand-grading is labor intensive, potentially inconsistent across graders, and does not scale to the sizes associated with Massive Open Online Courses (MOOCs). Residential courses are also becoming massive. For example, an introduction to programming at UC Berkeley, CS61A, has had more than a thousand students enrolled per semester [76]. Some Computer Science teachers, such as John Guttag and Ana Bell at MIT, are simultaneously teaching hundreds of students in residential programming courses and thousands of online students in MOOC versions of their courses.

Variable naming is a specific, important aspect of writing readable, maintainable code, and many teachers want to give feedback on it. There is no ground truth for which variable names are good or bad, but a teacher can still make informed judgments. The quality of

Foobaz 1: Misleading or vague 2: Too abbreviated 3: Fine Most popular untagged variable Quiz preview

Variables with feedback:
 Selected variables: ans

```

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

1538 / 3853 solutions

tag	base	3846	tag	exp	2755	tag	result	3095
	base	3735	exp	2612		Fine	result	2495
	a	8	b	8		Fine	result	2495
	base	3735	exp	2612		Too Short	ans	117
	base	3735	exp	2612			res	91
	base	3735	exp	2612			power	32
	base	3735	exp	2612			answer	31
	base	3735	exp	2612		Misleading or vague	x	28

Figure 4-1: 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.”

a name is best 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.

This chapter introduces Foobaz, a system 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 student solutions. The *role* of a variable is a function of the sequence of values that the variable takes on 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 with judgments like “too short” or “misleading.” Figure 4-1 shows the teacher interface.

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 label. Rather than just receive a label on one of their own names, the quizzes also allow students to see examples of good and bad alternatives. This moves the feedback closer to the ideals described by variation theory.

Quizzes are personalized by Foobaz so that students consider good and bad names for variables with the same role in their own solutions. Personalized quizzes render the original student solution with a specific variable replaced by an arbitrary symbol. The quiz presents the student with several variable names as candidate replacements for the symbol. The name that the student originally used may be one of the choices. The student selects appropriate labels for the variable names before checking their labels against teacher labels.

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 problems and received personalized quizzes generated with Foobaz by teachers in the first study.

4.1 User Interface

Thousands of students submit correct solutions to the same programming problem in MOOCs and massive residential courses. Many of these solutions include variables that take on the same sequence of values when run on the same test cases. Table 3-26 in Chapter 3 shows these common variables, the names students most commonly gave them, and some of the less common names for the same variables.

The quality of a name is most easily judged when the teacher understands (1) the algorithmic

role of the variable that bears the name and (2) the relationship between the name and other names in the surrounding code. While variable roles can be repetitive across many solutions, their names can be unpredictable and vary greatly in quality. For example, `result`, `val1`, and `s` might all refer to the same running sum in different student solutions. Instead of giving feedback on variable names by browsing student solutions one by one, teachers could provide feedback on the basis of variable roles.

In Foobaz, teachers can browse *stacks* of student solutions represented by platonic solutions. A stack is a set of solutions whose code is identical after standardizing formatting, normalizing 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.

Foobaz uses teacher labels and comments to provide students with tailored feedback in the form of personalized quizzes on variable names. A teacher only needs to label a few student-chosen names for the same variable role. Foobaz will then generate a quiz and every student whose solution contains that variable role can receive a personalized form of that quiz.

Foobaz is distinct from powergrading systems. Instead of grading as many names as possible, teachers work with Foobaz to strategically label a small subset of good and bad examples. If powergrading were possible, students might get feedback on their own variable name choices, but with Foobaz, they can learn from the good and bad choices of others.

4.1.1 Producing Stacks and Common Variables

The OverCode analysis pipeline executes each student solution on a test case and tracks the sequences of values that variables take on. Common variables are identified as those that take on the same sequence of values in multiple solutions. After executing every solution, the OverCode pipeline groups solutions into stacks of similar solutions. Each stack is represented by a single platonic solution. In the Foobaz teacher view, the teacher browses

the platonic solutions and annotates the quality of variable names for common variables.

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. 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 4-1 shows the teacher view for this task.

The teacher is presented with a scrollable list of stacks. Each stack is represented as a platonic solution followed by a table of alternative variable names. Since some of the tables are taller than the screen, the platonic solution 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 the most popular name given to each common variable serve as column headers. Each row of the table represents a unique set of variable names used in a solution. For example, `secretWord`, `lettersGuessed`, `guessedWord`, and `char` are all the names that were used in one particular raw solution. Foobaz shows *sets* of variable name choices, rather than independent columns of variable names, because early pilot testing showed 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 platonic solution are highlighted, so they can develop an understanding of the variable 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 teacher 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 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. During pilot studies, teachers were motivated by the progress bar to maximize their coverage. Teachers can maximize progress bar growth by clicking on a toolbar button that selects the next most popular but not yet labeled name for a common variable and scrolls it into view.

Several Foobaz features support labeling efficiency. Teachers can navigate through the variable names in the table using arrow keys and press hotkeys to label them. For example, the teachers could press 1 to rate a variable name as “misleading or vague” and 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

You recently wrote the following solution, and we've replaced one variable's name with A:

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

Rate the quality of the following names for the bold symbol A:

Name	Misleading or vague	Too abbrev	Fine
a	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
number	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
out	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
result	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

[Compare to teacher](#)

Figure 4-2: A personalized quiz as seen by the student. The student is 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 labels are revealed, along with their explanatory comments.

variable name replaced by an arbitrary symbol everywhere it occurs. In a personalized quiz, this solution is the solution the student wrote themselves, as shown in Figure 4-2. The quiz presents the student with several variable names as candidate replacements for the symbol. One choice may be the original name the student used. The student labels these names before checking their labels against those of the teacher. If a student 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 4-3) 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

Feedback Quiz Preview

Stacks with quizzes:

Jump to Next Program Without a Quiz
Finalize Quizzes

i: 0 -> 1 -> 2 -> 3 -> 4

Merge quiz See a stack that receives this quiz Send quiz to students?
students covered: 172 / 1069

Include in quiz?	Local Name	Misleading or Vague	Too Short	Fine	Comments
<input checked="" type="checkbox"/>	i	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="text"/>
<input checked="" type="checkbox"/>	x	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="text"/>
	<input type="text"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="text"/>

letter: t -> i -> g -> e -> r

Merge quiz See a stack that receives this quiz Send quiz to students?
students covered: 771 / 1069

Include in quiz?	Local Name	Misleading or Vague	Too Short	Fine	Comments
<input checked="" type="checkbox"/>	c	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="text"/>
<input checked="" type="checkbox"/>	char	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="text"/>
<input checked="" type="checkbox"/>	letter	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="text"/>
	<input type="text"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="text"/>

result: -> _ -> _i -> _i_ -> _i_e -> _i_e_

Merge quiz See a stack that receives this quiz Send quiz to students?
students covered: 350 / 1069

Include in quiz?	Local Name	Misleading or Vague	Too Short	Fine	Comments
<input checked="" type="checkbox"/>	guessedWord	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="text"/>
<input checked="" type="checkbox"/>	result	<input type="radio"/>	<input type="radio"/>	<input checked="" type="radio"/>	<input type="text"/>
<input checked="" type="checkbox"/>	s	<input type="radio"/>	<input checked="" type="radio"/>	<input type="radio"/>	<input type="text"/>
<input checked="" type="checkbox"/>	word	<input checked="" type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="text"/>
	<input type="text"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="text"/>

Figure 4-3: 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`, which iterates through indices of a list, the common variable `letter`, which iterates through the characters in an input string, and the common variable `result`, which accumulates one of the acceptable return values, `'_i_e_'`.

Problem Description	Source	Solutions
iterPower	6.00x (edX)	3875
hangman	6.00x (edX)	1118
computeDerivative	6.00x (edX)	1433
dotProduct	6.0001 (residential)	229

Figure 4-4: Number of solutions in datasets.

Problem	Misleading or Vague	Too short	Good	Total names
iterPower	3	3	15	929
hangman	7	4	10	763
compDeriv	6	5	10	670
dotProduct	11	3	17	180

Figure 4-5: Subjects in the teacher study, on average, labeled a small fraction of the total names, covering all three provided name categories.

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 goal of the teacher 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

The Foobaz teacher and student interfaces were evaluated in two consecutive user studies, one for each population. In order to evaluate the scalability of Foobaz, 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

Foobaz was evaluated on sets of correct solutions to four different programming problems, 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 4-4). The four programming problems, referred to here as `iterPower`, `hangman`, `dotProduct`, and `computeDerivative`, are representative of typical problems 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.

Students independently generated many names for variables that play the same role, and some names are far more popular than others. 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 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.

Common variables sampled by hand from each dataset were each given between 50 and 179 unique names by students. This confirms the impracticality of teachers passing a judgement on every name. In the 3875 `iterPower` solutions, students gave 179 unique

names to a variable representing the base being exponentiated and 64 unique names for a variable representing the exponent. In the 1393 `computeDerivative` solutions, students gave 176 unique names for the variable containing the result and 39 unique names for the most commonly used iterator variable. In the 1118 `hangman` solutions, students gave 50 unique names to the variable that iteratively takes on the characters of the secret word input argument and 99 unique names to the variable containing the string most commonly returned by solutions as the answer.

4.2.2 Teacher Study

The teacher study was designed to assess whether teachers can create variable name quizzes with the Foobaz teacher interface. If teachers can create quizzes, the study can measure how well those quizzes covered the student solutions in the dataset. These questions are answered by analyzing activity logs of each subject while using Foobaz, as well as their responses to survey questions about their experiences.

The hour-long session contains a warm-up exercise, a Foobaz demonstration, and a session where subjects use Foobaz to create as many variable name quizzes for as many students as possible. During the initial briefing, the teacher is informed that they will be looking at solutions that have already passed an autograder and instructed to focus only on variable names, ignoring other aspects of code style, structure, and correctness. The teacher then does the warm-up exercise: composing variable name quizzes inspired by the variation in variable names they see in a baseline interface for viewing student solutions to problem 1. They then watch a demonstration-based tutorial of Foobaz on student solutions to problem 2 as a training step. Finally they create variable name quizzes using the Foobaz interface for student solutions to problem 3 as the experimental condition. Over the course of the session, each teacher sees student solutions to three of the four possible problems. Counterbalancing ensures that the same problems are not always shown in the same interface.

The warm-up exercise introduces the teacher to the concept of labeling student-chosen

variable names with quality judgements and composing a multiple-choice quiz about variable name choices for students. To complete the exercise, the teacher reviews a page in a browser with all student solutions concatenated in a random order into a flat list of boxed, syntax-highlighted code. This is the same baseline interface as was used in the OverCode Coverage Study and it emulates existing methods of reviewing student solutions.

Using this baseline interface, the teacher rates as many good and bad variable names as possible with an eye toward maximizing coverage of names. Next, the experimenter shows the teacher an example of a variable name quiz and asks them to compose their own by listing variable names and labeling them as good or bad with whatever short descriptors and explanatory comments they wished. The teacher gets five minutes for each part, labeling and quiz-making. Finally, the teacher fills out a survey about the experience. Only eight of the ten teacher survey results are reported here, due to data loss.

The survey questions include both free response and 7-point Likert scale survey questions (1 means strong disagreement and 7 means strong agreement). The free response questions are:

1. “What did you like about this process of providing feedback on variable names?”
2. “What did you NOT like about this process of providing feedback on variable names?”
3. “What did you wish you could have or achieve during this process of providing feedback on variable names?”
4. “What, if anything, did you find in the data that surprised you?”

The 7-point Likert scale survey questions are:

1. “This interface helped me provide feedback to many students.”
2. “The interface helped me develop a high-level view of students’ variable naming skills.”
3. “I saw a large percentage of these students’ variable names.”
4. “I was able to give specific, personalized feedback to many students.”
5. “The system amplified my effort.”

The teacher learns about the Foobaz interface by watching a tutorial video. This training process takes between 10 and 15 minutes depending on the dataset shown in the video. The teacher is encouraged to hold their questions to the end, and answer them by interacting with the interface.

The teacher performs the same labeling and quiz-making tasks on a third dataset of solutions in the Foobaz teacher interface. The teacher spends 5 minutes on each task. During the study, some teachers decided to spend more time on each task. The experimenter allowed it. To conclude the experience, the teacher fills out the same survey about their experience again, followed by a final survey with 7-point Likert scale questions about each Foobaz feature and Foobaz overall.

Apparatus

All sessions were run on a laptop with a 15.4-inch 2880x1800 pixel Retina screen. All participant interactions with the system were logged with timestamps.

Participants

Ten participants (six female) with ages between 20 and 29 ($\mu = 23.1$, $\sigma = 2.7$) were recruited 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` 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.

Teachers responded more positively to the Likert scale questions about interface helpfulness after using Foobaz than after the baseline. Specifically, 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 scale). 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 ($\sigma = 0.5$).

Efficiency of Foobaz The efficiency of using Foobaz to create feedback came up repeatedly in teacher 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 was significantly reduced by implementing the Foobaz teacher interface in a templating language instead than d3.js, precalculating variable name counts, and removing repeated calculations.

Quiz Composition In both interfaces, teachers appreciated the potential pedagogical value of making quizzes: “I ... 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.

In spite of the unknown underlying distribution of good and bad variable names, the teachers find and label variables across all available categories in order to make quizzes. Figure 4-5 shows that teachers labeled only a small fraction of the total number of unique names and still found multiple examples for each of the categories of quality, which are misleading/vague, too short, and fine/good.

Coverage Most solutions in the edX datasets received at least one or two quizzes, as shown

in Figure 4-6. Figure 4-7 illustrates that, within minutes of their first interaction with the system, a teacher can label a significant portion of student-chosen variable names in a dataset. Their progress trails off as they encounter the long tail of names for particular roles and transition to creating better quizzes.

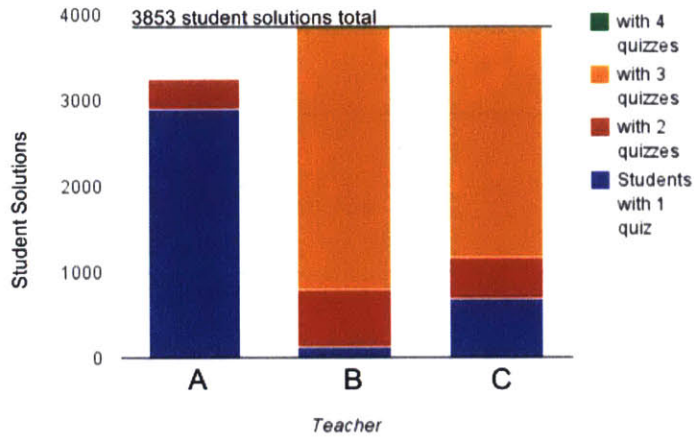
Due to technical difficulties, only one of the ten teachers used Foobaz to create quizzes for the 6.0001 dataset, collected from the residential class of only several hundred students. The teacher achieved a similar percentage of coverage in a similar amount of time, showing that the Foobaz workflow and output may be relatively invariant to the size of the dataset. Specifically, the quizzes the teacher created covered 87% of student solutions with at least one quiz.

Combined with Figure 4-5, Figure 4-6 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 4-7 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 examples of good and bad variable names students would not otherwise get to see and learn from.

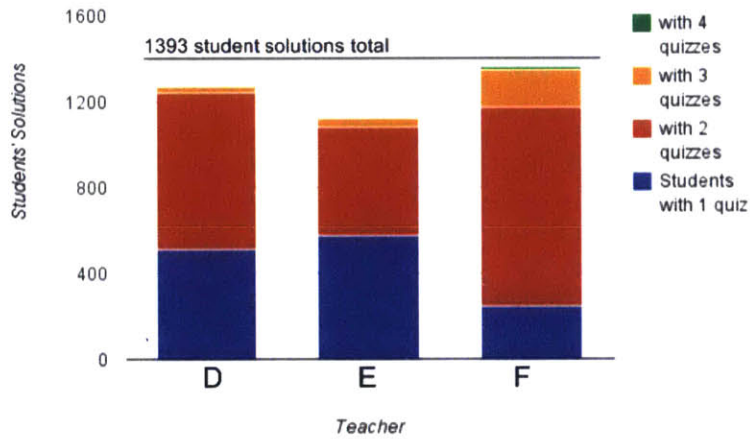
4.2.3 Student Study

A second study on the student side of the workflow was run in order to (1) find out if quizzes written by teachers are relevant to new students and (2) better understand student reactions to this novel form of feedback. In order to do this, recruitment targeted beginner programming students. Students were invited into the lab for a one-hour session to receive personalized quizzes generated by the teachers in the previous study. Two experimenters, including the author of this thesis, separately ran students through the study in parallel, using two different machines.

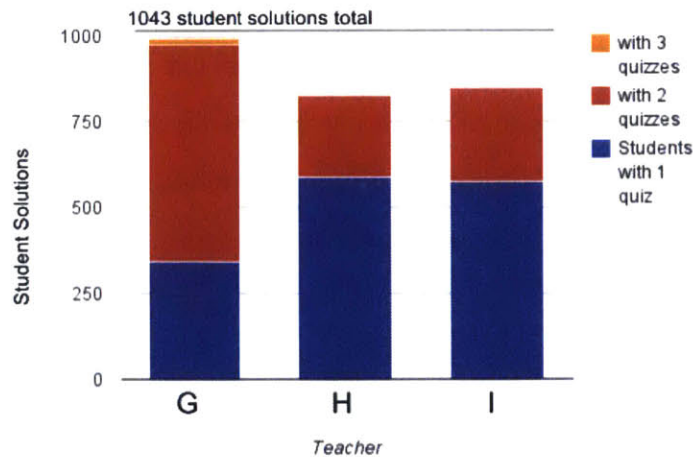
Before the start of the study, quizzes composed by teachers using Foobaz in the first study



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



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



(c) Quiz coverage for student solutions to the `hangman` exercise

Figure 4-6: Quiz coverage of student solutions across three datasets.

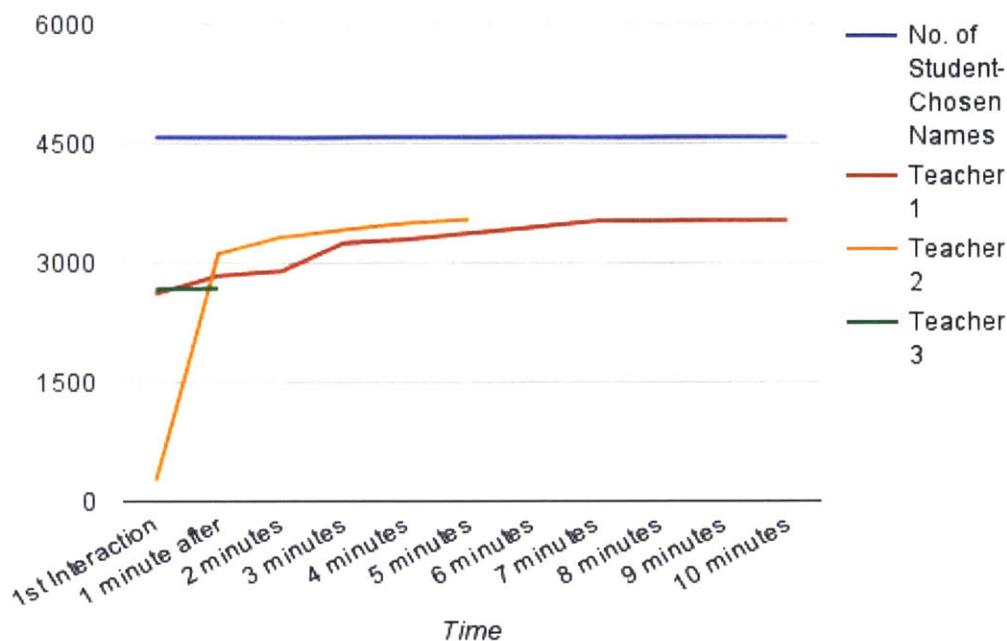


Figure 4-7: Variables in `iterPower` solutions labeled by each teacher.

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, quizzes written by teachers were randomly subsampled to include a maximum of five alternative names for students to consider.

Since personalized quiz generation was not implemented at the time of this study, the experimenters performed a wizard of Oz simulation. At the start of the session, the student wrote a solution to one of the four programming problems. The experimenter mentally executed the student solution and compared the behavior of its variables to the variable behavior covered by the quizzes by hand. If one or more quizzes matched, one quiz was randomly selected. The experimenter then made a copy of the student solution and replaced every instance of the variable to be quizzed on by an arbitrary symbol, a bold letter **A**. The experimenter appended the quiz to the modified copy of the student solution and delivered it to the student as a personalized quiz. If one or more quizzes did not match, then the student received a generic quiz, about a variable name in a solution other than their own. If this experiment was rerun, the wizard of Oz technique would be unnecessary. Based on the

success of the study, personalized quiz generation was implemented.

After the student completed their personalized quizzes, they took a survey about their experience. Students who completed the programming problem and quizzes with significant time left in their one-hour session repeated this process for a second programming problem.

Apparatus

All sessions were run on laptops with 15.4-inch or 13.3-inch screens. All participant interactions with the system were logged using the edX platform infrastructure.

Participants

Six participants (four female) who were either undergraduate or graduate students, were recruited 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 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. 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.

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.” The average level of student 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 perceived the naming preferences of the teachers through the quizzes and disagreed with them. In this situation, when the teacher left no explanation, students tried to imagine what the teacher was thinking, expressed displeasure at the lack of explanation, or decided that they still disagreed with the teacher. 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 multiple choice options down to five 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

Foobaz may require significant additional work to handle more complex programming problems. OverCode and Foobaz both use the same underlying technique for recognizing common variables across solutions. Programming problems that require more complex solutions may have a great diversity of solutions and a greater diversity of variable behaviors in those solutions. This may reduce the number of student solutions that share common variables. While the Foobaz teacher interface supports merging similar common variables together for the purpose of making quizzes about a more general variable role, it may become too tedious for solutions to more complex programming problems. Fully or semi-supervised methods of merging common variables may need to be developed, which would also help OverCode scale to more complex programming problems.

4.4 Conclusion

Foobaz demonstrates how a teacher can efficiently give feedback to students in massive programming classes on a subjective aspect of solution variation, variable naming. These variable name quizzes can be authored in less than 20 minutes and they are reusable as long as the programming problem does not change.

Foobaz also gives the teacher a high-level view of the distribution of student variable naming choices. In addition to composing feedback, the teacher may be able to revise their teaching materials about variable naming to better reflect common student mistakes.

Chapter 5

Learnersourcing Debugging and Design

Hints

This chapter presents two workflows that prompt students to engage with the diversity of solutions written by fellow students, reflect on their own debugging successes, and write hints that may help current and future students. It is adapted from a paper presented at the ACM conference on Computer-Supported Cooperative Work and Social Computing (CSCW) in 2016 [40].

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 [10]. 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.

The two workflows in this chapter address this problem with a form of *learnersourcing*. Prior

work on learnersourcing demonstrates that students can collectively generate educational content for future students, such as video outlines and exam questions, while engaging in a meaningful learning experience themselves [60, 84, 129].

This work builds upon learnersourcing by exploring how it can be applied to the generation of personalized hints during more complex problem solving. While prior work determined which learnersourcing task to present to the student based on what information the *system* needed [129], many educational topics like digital circuit design require some domain expertise that students are not equally well versed in. This raises the question of learnersourcing task routing. More specifically, which learners should be assigned to generate which hints? 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 presents two workflows for learnersourcing hints that assign hint-generating tasks to students based on what problems the students have 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, students 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 chapter describes their deployment in a college-level computer architecture course. 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.

The workflows are applied to two kinds of hints, *debugging hints* and *optimization hints*. A

debugging hint is advice written by a student to help a future student change their solution so it generates the expected output for a particular input. An optimization hint is advice written by a student 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. The system for learnersourcing debugging hints is called Dear Beta and the system for learnersourcing optimization hints is called Dear Gamma.

5.1 System Design

There are a number of necessary decisions to make when designing interventions to collect and deliver hints. This chapter provides some answers to key design questions in the context of two systems deployed within a computer architecture class.

5.1.1 Design Questions

When should students 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 [129].

How should students 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 requests help. The hints could be algorithmically selected based on the student's work so far and the hints they

have already received. This chapter demonstrates both the 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 the system designer ask, or allow, a student to generate? In cases where the student *start state* prior to overcoming an obstacle and *end state* after overcoming the obstacle are known or inferred, the system can ask the student to write a hint that would help someone else overcome the same obstacle. For example, in the case of circuit design, a student who has recently fixed a bug by resolving a particular *autograder error* may be capable of writing a debugging hint for other students who get that autograder error. An autograder error occurs when a student solution does not return the expected values.

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. This chapter explores both of these directions by asking each student to do both comparisons. To keep hint generation relevant to the student's current task and to minimize cognitive load, students were not asked to generate hints between pairs of solutions when they were familiar with neither solution.

How should hints be indexed? There are many possible ways to index the hints for more accurate retrieval later by students in need. For example, debugging hints could be indexed by the autograder error they are intended to resolve. Optimization hints could be indexed by the student start state, end state, or both with respect to performance. 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.

Two systems, Dear Beta and Dear Gamma, were built for students in an undergraduate digital circuit programming class, which required making several specific indexing choices. In that

course, students run their buggy solutions through teacher-designed sequences of test inputs. Dear Beta indexes debugging hints by the first autograder error that disappears once the bug is resolved. During optimization, the goal is not simply to attain a correct solution, but to arrive at a more optimal correct solution. Dear Gamma indexes optimization hints by both start and end states, which is 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. 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 [123], 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 an assignment, the system could *always* give the student hints that help them create the *next optimal solution* found by other students. This would hopefully be an optimization challenge that falls within the student's zone of proximal development [125]. 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. Dear Gamma uses 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. All students were required to generate optimization hints as part of a reflection activity immediately after submitting their first correct circuit. In contrast, writing debugging hints was voluntary. Some bugs, like random typos, do not warrant required reflection.

How should the variation in hint quality be handled? When it is possible to collect data on helpfulness, collect it. For example, in a user interface, students can upvote hints and hints can be sorted by student upvotes. When hints are being pulled by students in need and there is little or no data about hint quality, the system could deliver multiple hints at a time. Even if one or more hints in the delivered batch are of poor quality, their aggregate message might still be helpful for a student. If most of the hints in the batch are about a feature that is irrelevant to the student receiving the hints, the remaining hint(s) might be about something different and more relevant. Limiting the size of the batch to, for example, five, may prevent students from feeling overwhelmed.

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 [24]. For simplicity, Dear Beta and Dear Gamma both hide student identities.

5.1.2 Self-Reflection Workflow

In the self-reflection workflow, students iteratively modify their solutions to pass as many teacher-created tests as possible. For any autograder 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 autograder 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 autograder 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 [21], self-explanation helps students integrate new information into existing knowledge.

5.1.3 Comparison Workflow

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

Dear Beta and Dear Gamma each have their own user interface. *Dear Beta* is a Meteor web application that serves as a central repository of debugging advice for and by students in the class. It learnersources debugging hints with the self-reflection workflow. The name alludes to both the “Dear Abby” advice column and the Beta processor that students create in the class. *Dear Gamma* learnersources optimization hints with the comparison workflow. To collect hints, the Dear Gamma web interface requires students to complete a solution comparison activity after submitting their final correct circuit for a class assignment.

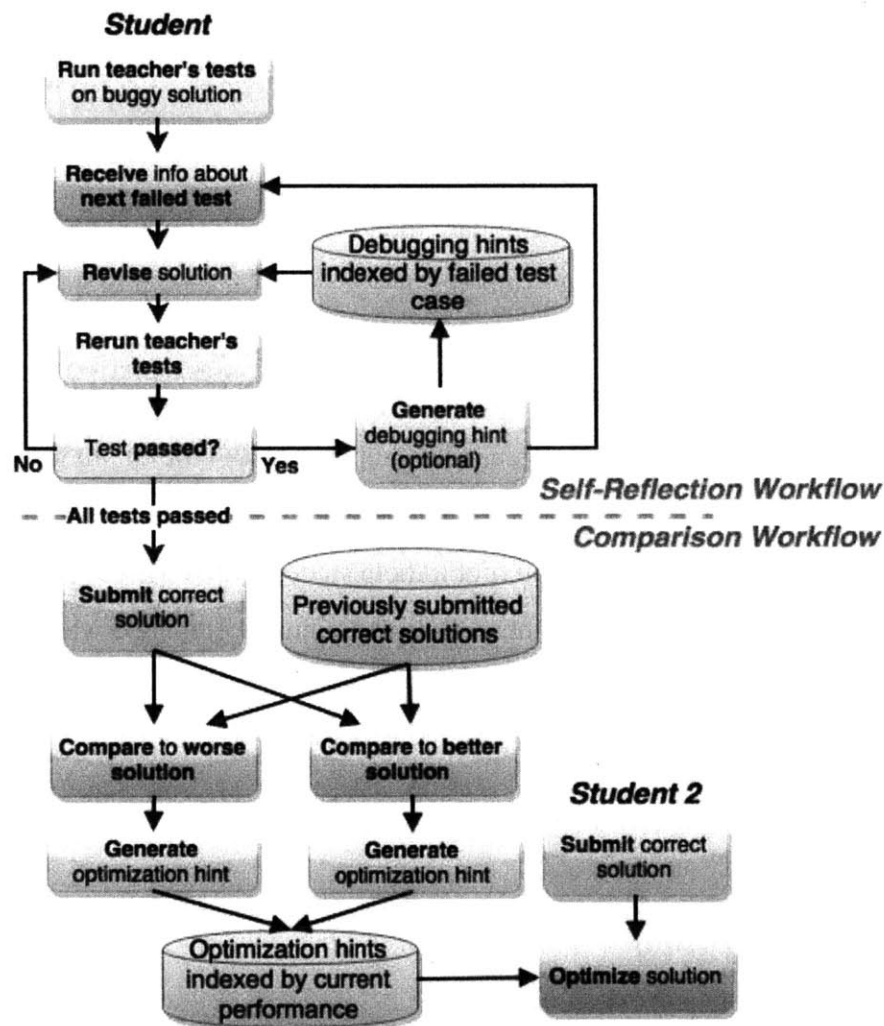


Figure 5-1: In the *self-reflection* workflow, students generate hints by reflecting on an obstacle they themselves have recently overcome. In the *comparison* workflow, students compare their own solutions to those of other students, generating a hint as a byproduct of explaining how one might get from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

5.2.1 Dear Beta

Dear Beta is an application of 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 an autograder error: for a particular input (test number n), y was the expected output and the student's circuit returned z .

Students eliminate autograder 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 autograder 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 autograder 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 5-2). 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 autograder 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 autograder 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 autograder 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 5-3). Their new hint may be a clearer rephrase of an existing hint or hint at yet another way to resolve

Dear Beta: An Advice Column

Test Number [Find or Add Error](#) [Instructions](#) [Submit Feedback](#)

Request	File	Test Number	Action
1	lab5/beta	29	Add a hint!
1	lab5/beta	33	Add a hint! Upvote 2 If you're using muxes to choose Ra/Rb and 0, make sure the you're not using the same selector for both muxes. Upvote 2 Make sure that you're setting R31 to be 0 Upvote 0 Make sure to check if BOTH Ra and Rb are R31.
0	lab5/beta	38	Add a hint! Upvote 0 Make sure your XP value is being set correctly.
0	lab5/beta	43	Add a hint!

Figure 5-2: *Dear Beta* serves as a central repository of debugging advice for and by students, indexed by autograder errors. In this figure, there are three learnersourced hints, sorted by upvotes, for a autograder error on test 33 in the ‘lab5/beta’ checkoff file.

Request 4 lab5/regfile 2 My hint is... [Submit](#)

Figure 5-3: After fixing a bug, students can add a hint for others, addressing what mistake prevented their own solution from passing this particular autograder test.

the autograder error. Given the variety of processor designs and implementations, there may be several ways any given autograder error may be thrown.

Teacher Feedback on Early Prototypes of Dear Beta

After deploying initial prototypes of Dear Beta for two semesters, Teaching Assistants were invited to share their complaints, requests, and experiences. Four TAs gave interviews, in person or by email. Their feedback and experiences informed Dear Beta’s final design.

Both TA_1 and TA_3 adapted to the Dear Beta 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 their problem with Dear Beta 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 the Dear Beta 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 the Dear Beta 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

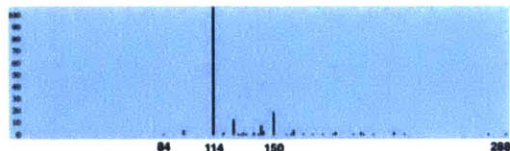
To learnersource optimization hints, students participate in the mandatory Dear Gamma hint writing activity right after they first pass all autograder tests for a particular digital circuit,

the Full Adder. The student has reached correctness so they are now in a position to think about optimality.

To collect Dear Gamma hints, 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 author of the less optimal solution might have missed. For example, the student might write, *Remember DeMorgan's Law: you could replace the 'OR' of 'ANDs' with a 'NAND' of 'NANDs.'* When the student's 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 future student who authored a suboptimal solution like their own.

The comparison workflow was 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 5-4 shows an example of the page for a student with a 114-transistor solution. This was a required activity for all students during one semester of the course and 435 student-written hints were collected from approximately two hundred students who were enrolled in the course. The results of this deployment of the modified workflow is shown in Figure 5-5 as a Sankey diagram of the distribution of solutions connected with student-written hints.

Your design has a total of 114 transistors. For comparison, the graph below shows the number of designs submitted in a previous semester (y-axis) vs. their transistor count (x-axis).



Below we will ask you to compare your design with a few other designs submitted previously. First, for reference, here's your code:

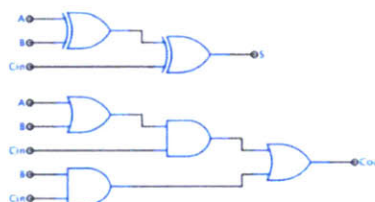
```
*HANG
.subckt nand2 a b z
MPD1 z a 1 0 NENH sw=2 sl=1
MPD2 1 b 0 0 NENH sw=2 sl=1
MPU1 z a vdd vdd PENH sw=2 sl=1
MPU2 z b vdd vdd PENH sw=2 sl=1
.ends

*nor
.subckt nor2 a b z
```

Comparison #1:

If we used the design shown at the right for the FA module, a 3-bit adder would require 132 transistors, larger than your design by 18 transistors.

Imagine you're an Lab Assistant in a future semester of CompArch and a student submits a solution like the shown on the right. What advice would you give them on how to make their solution as good as yours?

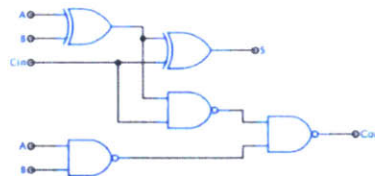


... enter your advice here

Comparison #2:

If we used the design shown at the right for the FA module, a 3-bit adder would require only 96 transistors, smaller than your design by 18 transistors.

Imagine you're an LA in a future semester of 6.004 and a student submits a solution like yours. What advice would you give them on how to make their solution as good as the one in the figure to the right?



... enter your advice here

Figure 5-4: 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.

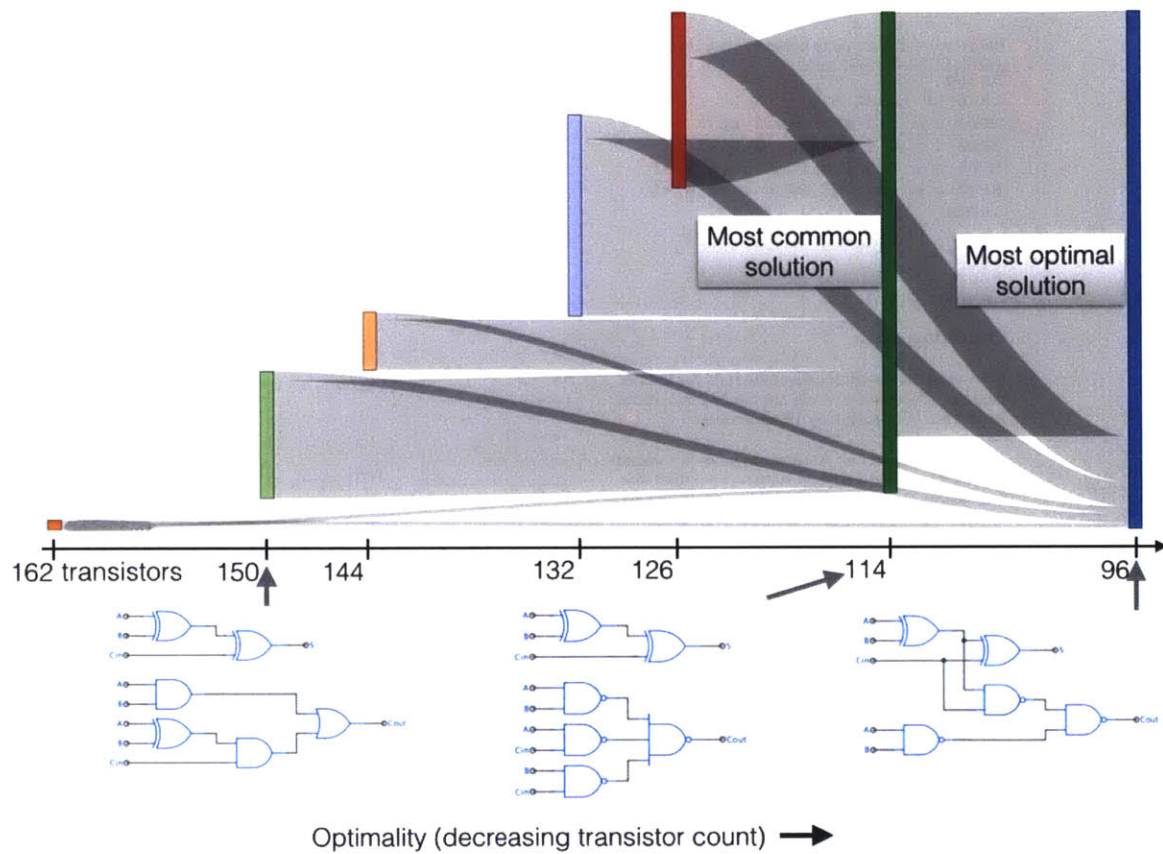


Figure 5-5: Sankey diagram of hints composed between types of correct solutions, binned by the number of transistors they contain. The optimal solution has only 21 gates and 96 transistors while the most common solution generated by students has 24 gates and 114 transistors.

5.3 Evaluation

To evaluate the extent to which learnersourced hints can support problem solving, Dear Beta and Dear Gamma were deployed to the computer architecture class, which had an enrollment of more than two hundred students. Dear Beta was deployed for 6 weeks. During that time, the system collected student-generated debugging hints. The Dear Gamma optimization hint collection interface was released to students as a mandatory part of a particular lab. During a later offering of the same course, nine new students were recruited to take part in a lab study designed to shed light on how they solve a typical engineering problem using the learnersourced optimization hints. The questions the evaluation sought to answer are: (1) *What are the characteristics of student-generated hints?* and (2) *Can students solve problems using those hints?*

5.3.1 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 computer lab. It was left up for the remainder of the semester for students to refer to, if completing work late. The system tracked student logins and engagement with site features. An initial prototype of Dear Beta was also deployed for two consecutive semesters prior to this final system design and study.

5.3.2 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, the hints collected with Dear Gamma were decomposed by hand into the three kinds of hints that typically comprise a hint sequence: 1) *pointing hints* direct student attention to the location of error in case the student understood the general principle but did not realize they could apply it there; 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 [123].

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), the hint was assigned to the pure category (i.e., pointing) that was in common between the two coders' labels. When the categories assigned by each coder did not overlap, the hint was discarded. The minority of hints (3.2%) labeled as irrelevant or unhelpful were excluded.

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 and offered \$30 for participating in one hour-long session. The author of this thesis ran all sessions as the experimenter. While obtaining informed consent from each student, the experimenter explained that the study would assess the effectiveness of previously collected hints for optimizing circuits, not the skills of the student. Students were not told that the hints were written by students in a previous semester of the same class specifically for the Adder

problem. Instead, the experimenter presented Dear Gamma hints as anonymous, potentially helpful messages.

Students began the study by reviewing their solution to the Adder problem, which was due earlier in the term. All recruited students had completed it. During the study, three batches of hints were shown in the order of pointing, teaching, and bottom-out, but randomly selected within each category. The experimenter noted 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."* These questions appeared 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, the post-study questionnaire included 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.

5.3.3 Limitations

These studies do not have control groups, so the results cannot support any claims about student learning. Dear Beta was deployed in a real classroom setting while students worked on a relevant class assignment, and the results contain qualitative and quantitative measures of student engagement with the system only. Some of those observed behaviors and opinions may be derived from the participants' sense of novelty, rather than the underlying value of the system.

5.4 Results

5.4.1 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 the Dear Beta release date and the lab due date, users added 76 autograder 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 user 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,

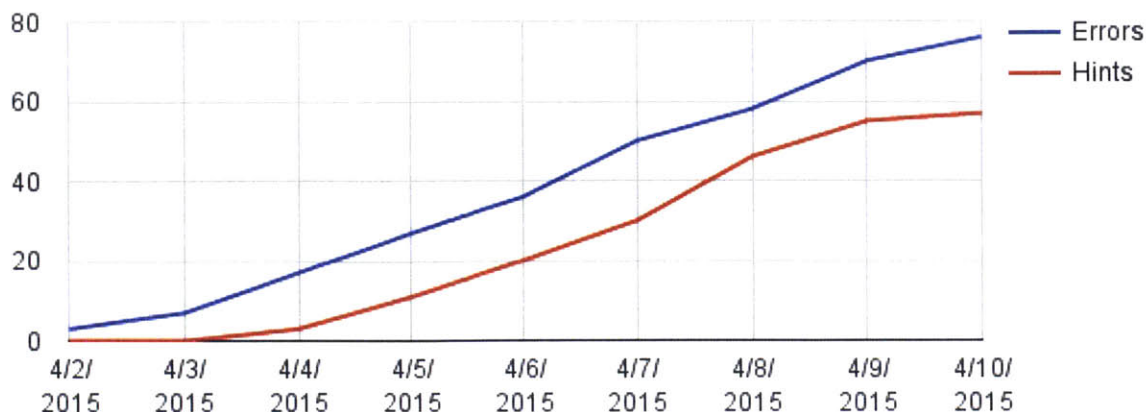


Figure 5-6: Between the Dear Beta release date (4/2) and the lab due date (4/10), autograder errors were consistently being entered into the system. The addition of hints followed close behind.

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.

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

Hint Distribution Figure 5-5 is the Sankey diagram of the optimization hints collected by Dear Gamma. The number of hints between certain key transitions, such as between

Hint type	Count	(%)	Representative examples
Pointing (<i>p</i>)	62	14%	“You don’t have to keep S and Cout as two separate/independent CMOS gates.”
Pointing and teaching (<i>pt</i>)	81	19%	“Instead of making the S and Cout components individual, combine them together to save computation power.”
Teaching (<i>t</i>)	111	26%	“Instead of recalculating values, save computation results to save time!”
Teaching and bottom-out (<i>tb</i>)	19	4%	“Via application of demorgan’s theorem, NAND2 (XOR A B) Cin is equivalent to NAND3(NAND2 A B) Cin.”
Bottom-out (<i>b</i>)	78	18%	“Use the output of a xor b for one of the nand2 gates.”
Unhelpful or 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.

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 student 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 (five pointing hints, followed by five teaching hints, followed by five 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 examples. The Cohen’s κ [25] inter-rater reliability was 0.54, which indicated that the two coders had moderate agreement across the six hint categories [124].

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 using the optimization hints, S_6 was the only student who significantly deviated from the correct line of thought by removing all inverting gates. Optimal solutions have only inverting gates. As soon as S_6 saw that their transistor count had increased rather than decreased, the student revisited the hints, realized their mistake, and correctly optimized their circuit. None of the hints themselves were incorrect, though some were deemed irrelevant

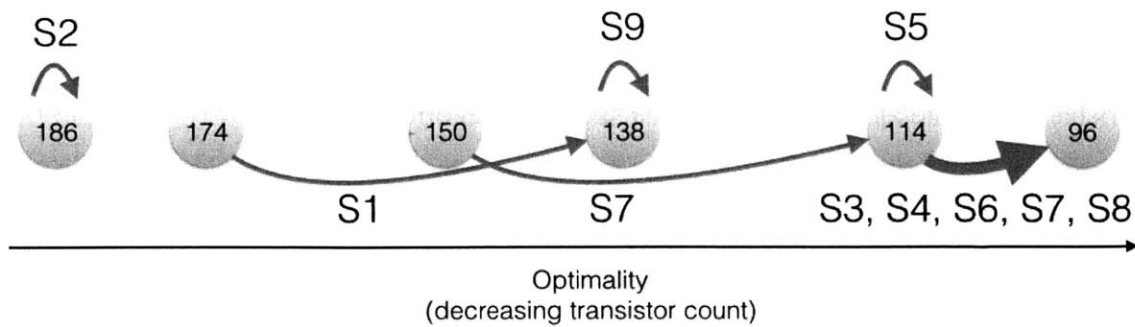


Figure 5-7: Six of the nine lab study subjects were able to improve the optimality of their circuits with the help of the Dear Gamma hints. Subject S7 was able to make two leaps—one to a common solution with 114 transistors and another from the common solution to the most optimal solution at 96 transistors.

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 instead. Based on these hints, the student got the final optimization step without understanding, and appeared to feel cheated out of 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” ($\mu=6$, $\sigma=1.1$ on a 7-point Likert scale). The remaining three subjects with suboptimal gates disagreed with the same statement ($\mu=2.6$, $\sigma=2.1$ 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” ($\mu=5.4$, $\sigma=1.6$).

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 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.

5.5 Discussion

This section reviews the research questions and critical design decisions, including prompt clarity, index choice for hints, representation choice for alternative solutions, and hint progressions.

5.5.1 Answers to Research Questions

Our study sought to evaluate the characteristics of student-generated hints. Table 5.1 shows 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.

Second, the studies sought to evaluate whether students 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 student 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.

5.5.2 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 students 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 prompt for hints. In the comparison workflow, transistor counts did not always capture lower-level reasons for a suboptimal circuit. This resulted 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, machine learning methods could classify a given hint type. Alternatively, hint type classification could be learnersourced.

5.5.3 Generalization

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.

5.6 Conclusions

This paper enriches learnersourcing by shaping the design space for learnersourcing personalized hints, and presenting two workflows that engage students in hint creation while reflecting on their own work as well as that of peers. Dear Beta and Dear Gamma are applications of these workflows to the creation of debugging and optimization hints, matching students to the appropriate hint creation task given their current progress. Results from the

deployment study and subsequent lab study demonstrate the feasibility of these workflows, and indicate that student-generated hints are helpful to students. 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

Bayesian Clustering of Student Solutions

The original vision of OverCode was to discover pedagogically valuable themes of variation within thousands of student solutions to the same programming problem. It summarizes the most common solutions well because they are grouped into large clusters. It also produces a long tail of small clusters. Pulling out themes of variation from the long tail can be challenging.

There are a variety of clustering methods that could be applied to pulling themes out of the long tail of small OverCode clusters. Some methods create hierarchical clusterings. At higher levels of the hierarchy, clusters contain a larger diversity of solutions. Some methods break solutions down into clusterable components or allow solutions to partially belong to several clusters, so that each solution does not need to be strictly associated with a single cluster. Bayesian methods treat student solutions as examples sampled from some underlying distribution of student solutions. Each of these families of methods have appealing properties.

The recent development of Bayesian clustering methods built explicitly for interpretability spurred my exploration of Bayesian methods for clustering student solutions. This chapter explores the application of two Bayesian clustering methods to the normalized solutions produced by OverCode.

6.1 OverCode as Clustering Algorithm

In OverCode, each Python solution is normalized and clustered. All solutions that share the same set of normalized lines of code are clustered together. The clustering is a strict partition of the solutions. This clustering process is hard and agglomerative but not hierarchical. It was designed with interpretability in mind. Hierarchical clustering may also be appropriate in this context, as long as human interpretability does not suffer. The OverCode clustering pipeline is also interactive, because the results are shown in an interface that allows the human to tell the system that some lines of code are not different in meaningful ways. If any clusters differ only by those lines of code that the human considers to be the same, those clusters will be merged.

In OverCode, what can vary and what is invariant across all the solutions within a stack is clear, just by reading the platonic solution that represents the stack. Syntactic differences split apart groups of solutions into smaller stacks so that each stack's platonic solution matches the syntax used in the solutions it represents. This can also produce more stacks than teachers can read. OverCode does normalize and cluster correct solutions so that they can be more easily understood as a group, but it does not pull out larger themes of variation as clearly as initially hoped.

6.2 The Applicability of Bayesian Clustering Methods

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. Two latent variable models have been explored in a preliminary way for this purpose: the Bayesian Case Model (BCM) [59] and Latent Dirichlet Allocation (LDA) [9].

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 platonic solution that shares the same set of normalized lines with all other members of the cluster, BCM learns a subspace of features that it determines are most characteristic of the solutions within that cluster.

BCM has an interactive variant called iBCM. iBCM allows the teacher to directly modify the prototype and the subspace chosen by BCM if it disagrees with their domain knowledge or preferences. This teacher action 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 the BCM Dirichlet prior is set to promote sparsity. That means that the mixture distributions over solutions will be biased to 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.

Since teachers can be poor at clustering solutions, i.e., inconsistent with each other [102], it may also be difficult for any algorithm to find a clustering that appeals to most teachers. However, there are statistical models, like LDA, that cluster components of solutions instead of entire solutions. LDA was selected as an alternative to BCM in order to explore how solution component clustering might better support teachers who do not agree on whole-solution clusterings.

BCM was built explicitly with interpretability in mind, but it is still not clear whether LDA or BCM are more interpretable in the context of clustering OverCode stacks. LDA has some known interpretability problems. As part of its training process, LDA learns distributions over features, also known as *mixture components*. It can be difficult to interpret exactly what an LDA mixture component means just by looking at its distribution over features.

6.3 Clustering Solutions with BCM

BCM was applied to the platonic solutions generated by OverCode. The platonic solutions encode both static and dynamic information, i.e., the syntax carries the static information and the variable naming encodes dynamic information. The platonic solution is tokenized and represented as a binary vector indicating the existence of the features, including renamed variables and language-specific keywords, such as normalized variable names like `listA` and Python keywords like `assert` and `while`. The result is a clustering of OverCode platonic solutions.

The screenshot shows a web interface for iBCM. At the top, it says "dot product" and "Ready for Input". Below this, there are two main sections: "Cluster Prototypes and Subspaces" on the left and "Cluster members" on the right.

Cluster Prototypes and Subspaces: This section displays three Python code snippets for a dot product function. The first snippet is highlighted in blue, indicating it is the selected cluster. In this snippet, the keywords `while`, `len`, and `return` are enclosed in red boxes. The second snippet uses `for` and `zip`. The third snippet uses `if` for a length check.

Cluster members: This section shows the code for the selected cluster (the blue one). It includes a "Show all stacks" button and a "Promote to Prototype" button. The code is:


```
def dotProduct(listA, listB):
    length=len(listA)
    iB=0
    total=0
    while iB<length:
        total=total+listA[iB]*listB[iB]
        iB=iB+1
    return total
```

 Below this, there is another "Promote to Prototype" button and a second code snippet:


```
def dotProduct(listA, listB):
    iB=0
    total=0
    while iB<len(listA):
        product=listA[iB]*listB[iB]
        total=total+product
        iB=iB+1
    return total
```

Figure 6-1: This interface displays iBCM clusterings of OverCode platonic solutions and affords user feedback on cluster quality. 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 identifies as characteristic of the cluster represented by that prototype. When a user hovers the cursor over a keyword or variable name, e.g., `len`, it is highlighted in a blue rectangle, indicating that it can be clicked.

6.3.1 Preliminary Evaluation

BCM was run on three different programming problems selected from those previously analyzed in Chapter 3. An interface, shown in Figure 6-1, displayed the results, with iBCM

continuing to run in the backend, ready to update the clustering in response to teacher feedback. The interface afforded promoting a member of a cluster to be its prototype and clicking on a token within a prototype, such as a variable name or keyword, to toggle whether or not it is labeled a characteristic feature of that cluster.

In order to determine whether or not these clusterings and interface affordances are useful to teachers, a small pilot study was run on three teachers of introductory Python programming. For the pilot, two more interfaces were introduced as controls. One is a duplicate of the control interface used in the OverCode studies, i.e., a long list of syntax highlighted raw solutions in a random order in the browser. The second interface differs from the previous control by one factor: the raw solutions are replaced with their platonic equivalents.

The teachers were each given sets of Python solutions to view in each of the three interfaces. For each problem, they were asked to create a grading rubric and provide helpful comments for the students of the type that might be mentioned in a class-wide forum post.

The teachers 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 teachers more ways to modify the clustering, e.g., allowing teachers the option to kick an intruder out of a cluster and rerun BCM. Subjects also requested richer or higher-level features than variable names and keywords. Been Kim's PhD thesis [58] describes a follow-up full user study comparing the efficacy of BCM and iBCM on clustering OverCode platonic solutions.

6.4 Clustering Solution Components with LDA

Other researchers have documented a lack of agreement across human-made clusters of student solutions [102]. One possible explanation for poor consistency is that solutions are mixtures of design choices and teachers care differently about various aspects of solutions. 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 [9] as implemented in the Gensim toolbox [96] was chosen as the model to evaluate in this preliminary work.

Like BCM, LDA was run on the platonic 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.

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

In these examples, the input argument `base` would be considered a variable common to all three programs, but the input argument `exp` would not be. The variable `result` would

```
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 student solution using the Python keyword `while`.

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 design decisions about syntax.

LDA is often applied to corpora of textual documents, where the corpus is represented as a $W \times N$ 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.


```
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 student solution using the Python keyword `while`, where the student has not modified any input arguments, i.e., better programming style.

- **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

Figure 6-5: The sequences of variable values recorded by OverCode while executing `iterPower(5, 3)`.

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, student solutions to a particular problem are first run on a set of test cases within the OverCode analysis pipeline. The OverCode pipeline produces a set of platonic solutions and a set of features for each solution, including which variable sequences were observed during execution. A script extracts the variable-by-solution matrix. LDA is then run repeatedly with different values for the parameter that sets the number of latent mixture components and then inspected by hand. The results of running LDA on platonic solutions is inspected by hand because cluster validity metrics like perplexity and held-out likelihoods are not necessarily good proxies for human interpretability [19].

Inspection by hand can also be challenging. When LDA learns mixture components in this

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)

SOL- UTION	5	1, 5, . . . 25, 125	3, 2, 1, 0, . . . 1, 2, 3	3, 2, 1, 0	3
FIGURE 6-2	1	0	1	0	0
FIGURE 6-3	1	1	0	1	0
FIGURE 6-4	1	1	0	1	1

context, those components are distributions over variable behaviors. Rather than inspect these distributions, entire solutions containing high amounts of that mixture component were inspected instead. By comparing entire solutions with high and low amounts of each component, one can infer what each learned component captured. This method for understanding the results is similar to the recommendations of variation theory.

6.4.1 Preliminary Evaluation

3875 student solutions to `iterPower` were run on a set of test cases within the OverCode analysis pipeline. The OverCode pipeline produced a set of 977 platonic solutions and sets of features for each solution. LDA was run on the resulting variable-by-solution matrix, using different parameter values.

The LDA results were inspected by hand. 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 unnecessary conditional statements and the common, more concise solution.

6.5 Discussion and Future Work

LDA applied to a variable-by-solution matrix is a promising method for identifying variation within corpora of solutions to the same programming problem. However, the assumptions made in LDA, 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 [8] or the Hierarchical Dirichlet Process [117] will ultimately be a better model fit for this purpose. In the future, new user interfaces would be helpful for this task, especially one which helps humans compare the output of models with different parameter values.

6.6 Conclusion

The clustering described in the original OverCode work was relatively limited in scope, but it did produce platonic 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.

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 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.

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 an n -dimensional space, where each dimension cor-

responds to a design decision, e.g., whether to solve a problem iteratively or recursively. The design choices students make could be thought of as discrete points along each dimension in that solution space. While the number of distinct combinations of design choices students choose can be large, the number of dimensions in this space, which represent underlying design decisions, grows 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 `for` or `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 decisions, 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 likelihood that any two solutions occupy the same location in that space will go down. The regularity of code discussed in 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 vision of the solution space inspired and is shaped by the concrete systems built and described in this thesis. In each system, the dimensions of solution variety have been tamed and orderd by choosing what features to ignore and what features to index by. For example,

OverCode ignores whitespace, comments, variable names, and statement orders. It indexes by normalized lines. In order to assign a variable name quiz, Foobaz looks at the behavior of the variables in the student solution, ignoring everything else about it. 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.

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.

Computer scientists are often looking for better ways to find needles in haystacks and computationally-minded social scientists are trying to characterize the haystack [127]. One could think of Codex [35] as a mechanism for finding needles in haystacks because it helps find particular code fragments in large collections of open source code. One could also think of Webzeitgeist [65] as a mechanism for finding needles in haystacks because it helps find web design choices in large collections of web pages. OverCode, Foobaz, Dear Beta, Dear Gamma, and GroverCode are trying to faithfully represent the haystack and support needle-finding.

OverCode characterizes the haystack by identifying and displaying the relative popularity of different correct solutions. It also provides filtering and cluster-merging tools that help teachers find needles, e.g., interesting and unusual solutions. Similarly, GroverCode helps teachers understand the haystack of correct and incorrect solutions by displaying the distribution of solutions as a function of their correctness on test cases.

Foobaz characterizes the haystack of variable naming choices by displaying the relative popularity of different names in each OverCode stack of solutions and across the entire dataset of solutions. The feature of sorting by popularity allows teachers to see the common names as well as the uncommon names that are excellent or terrible needles for teachers to find.

The learnersourcing workflows helped students, in addition to teachers, understand the distribution of solutions and bugs. Dear Beta reveals some information about the distribution of failed test cases. The variety of student debugging hints for each failed test case reveals some information about how many different possible problems might cause each failed test case. In Dear Gamma, the optimality of circuit solutions was quantified as the number of transistors. The haystack was characterized by visualizing the distribution of solutions from least to most optimal. This distribution was examined by teachers and given to students after they completed their own solution. The distribution helped teachers identify the most optimal solution, which made up a very small proportion of all solutions. In other words, the edge of the solution distribution contained a valuable needle.

7.3 Helping Teachers Generate Solution-Driven Content

The future generations of tools like OverCode, Foobaz, Dear Beta and Dear Gamma might allow teachers to continuously mine the solutions of thousands of current and previous students, generating days worth of lessons just by extracting themes from both the innovative and problematic design decisions found in the growing pile of student solutions. This would be an opportunity to discuss and reflect on the various trade-offs that exist in each different solution or optimization.

Turning teacher actions and annotations into personalized feedback for students need not be restricted to variable names. For example, when a teacher writes a rewrite rule in OverCode, OverCode could treat that as an implicit annotation that one syntactic form is semantically equivalent and stylistically superior to another. Like variable name quizzes in Foobaz, the

teacher commentary, both implicit and explicit, could be sent to current and future students as a personalized quiz or just a helpful suggestion. Ideally, all the work that a teacher does in an interface to make sense of student solutions could be automatically translated into solution-driven discussion content and rich feedback.

These solution-driven lessons and feedback would be reuseable as long as the problem specifications do not change. This aligns well with the lifecycle of many courses that are eventually offered at scale. First, the material is iteratively developed and debugged in residential, face-to-face teaching environments. When the course is stable, it can be put up online for a larger number of remote students to engage with. Finally, the course can be archived in such a way that any new student can move through the material at their own pace, but there is no guarantee of staff or a cohort of fellow students moving through the material synchronously. Tools like OverCode, Foobaz, Dear Beta, and Dear Gamma accumulate solutions, commentary, and hints so that, after enough students and teachers have used them for a given problem, new students can benefit from them without synchronous peers or teachers.

7.4 Writing Solutions Well

The focus of introductory Python programming courses is often just on 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 to help students make good design decisions. For example, in *The Sense of Style*, Steven Pinker suggests picking apart examples of good writing to understand what makes them good [93]. The software industry uses one-on-one peer review, often called code review, to find bugs and judge design decisions. It serves as an informal teaching opportunity for software engineers. Some communities create design studios, where students give and receive constructive criticism from peers. 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.

Written rubrics can serve as a complement to peer review and design studio practices. For example, some software companies compose and maintain style guides against which new code is carefully compared. However, these guides may be more prescriptive than descriptive. In other words, they are not necessarily built on data about how software engineers actually design and write their code.

Design studios and peer reviews are not yet a standard practice in programming education. However, peer review practices are now starting to be used in large online courses to make up for low teacher-to-student ratios. At least one residential software engineering course, 6.005 at MIT, has set up infrastructure for peer review. Peer review forces students to engage with some sampling of other student solutions.

The work in this thesis takes this idea farther. Rather than hope that the randomly assigned peer review experiences provide 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 perspective by commenting on the popular and rare choices. Students can be overwhelmed by choices. For example, they might ask themselves, *Should I use range or xrange? Does it matter?* With concrete examples, teachers can help students identify what matters and what does not.

7.5 Clustering and Visualizing Solution Variation

The OverCode pipeline does unsupervised clustering. There are distinct failure modes when using clustering in the context of teachers reviewing solutions. Two are most relevant in this work. First, the representation of the solution and the measure of similarity between solutions can ignore, hide, or otherwise fail to capture what the teacher cares about. Second, when (1) the teacher cannot easily decide what a cluster means to them based on its members and (2) 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. OverCode is designed to reveal to the teacher what their thousands of students' solutions actually look like, modulo whitespace and comments. These solutions are rendered 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 accumulating 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 create separate clusters within the OverCode pipeline nearly ensures that any syntactic choices a teacher is interested in have 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, there is a clear interpretation of what an OverCode cluster can and cannot include, based on its platonic solution. Specifically, all solutions in the cluster have the same set of lines as the platonic solution after normalizing variable names, standardizing formatting, and removing comments. Second, the differences between clusters is made clear by highlighting which lines make the non-reference clusters different from the reference cluster. Third, the OverCode 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.

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 course improvements, including autograder updates and new examples for students to engage with.

7.6 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. Gulwani et al. [44] and [105] have also authored semantic analyses of programming languages that include variable behavior. The languages were C++ and Pascal, respectively. This suggests that OverCode and Foobaz could also be applied to other 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.

7.7 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: executable programs that solve the same programming problem. 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 teacher 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.

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 courses. 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 generalizable 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 ongoing programming course.

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 platonic solutions 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 the OverCode normalization 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 in which the same programming problem is assigned.

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 (1894)

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 thousands of students’ solutions actually look like, modulo whitespace and comments. These solutions are rendered for the teacher using the most common variable names and statement orders.

However, as programming problems escalate from the simple to the complex, OverCode’s utility breaks down. It can still create platonic solutions and display them in a way that helps teachers spot contrasts, but the clusters themselves become smaller and smaller. As programming problems 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. A critical part of forming OverCode stacks

is identifying which variables are semantically equivalent, based on their behavior during execution on the same test cases. Overcode then renames variables to their most common name across all solutions. This normalization process could be extended to subexpressions and code blocks as well, using semantics-preserving transformations, e.g., compiler optimizations, and probabilistic semantic equivalence (PSE) [88]. OverCode could, in the future, replace semantically equivalent or PSE subexpressions with their most common form across all solutions. Alternatively, this process could be controlled by the user, who decides how much they would like to hide less common semantically equivalent subexpressions. An interface like Foobaz could allow teachers to view and compose feedback on the distribution of semantically equivalent subexpressions, just as Foobaz allowed teachers to do for variable names.

The second promising avenue for creating larger clusters is continuing to model the corpus of solutions to the same programming problem as samples from an underlying distribution of solutions shaped by teacher explanations, student 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 that 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-occurring variable behaviors 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 teacher to interpret using the language of filtering.

When visualizing the resulting clusters, it is important to keep in mind the ease with which teachers can interpret the results. Simple modifications to the OverCode interface can go a long way, such as highlighting the sub-expressions or normalized 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 normalized variables. Since the variable behavior is used to normalize 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 Kevin Kwok's (unpublished) Python notebook demo, 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 [23], 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 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 the GroverCode sorting mechanism, the OverCode interface could support sorting clusters by their similarity to the reference as well as by cluster size.

8.2.2 GroverCode

The normalization 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 normalizing incorrect solutions in addition to correct solutions, without using any additional technology.

AutoGrader [109] could also 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. If a small subset of those corrections changes a solution's input-output behavior to match a reference solution, AutoGrader has diagnosed the bug(s) and can automatically correct the solution. It does not take into account which sets of corrections are more common, unlike the design philosophy that 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 normalized 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.

A variety of interface updates have been requested by the teachers who used GroverCode during live deployments. For example, some teachers requested the ability to dynamically add new tests to the test suite and to edit student solutions in the interface without switching to an external editor.

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

Dear Beta and Dear Gamma were most successful when directly integrated into teachers' existing mandatory assignments or actively promoted by staff. In future classes with supportive teachers, a module can be added to the Dear Beta workflow that identifies when a student is in a good place to supply a hint and prompts them to do so. 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.

Appendix A

GroverCode Deployment Programming Problems

A.1 Midterm Problems

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

A staff solution is:

```
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`.

A staff solution is:

```

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 dictionary is:
        one more than 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.

A staff solution is:

```

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

```

A.2 Final Exam Problems

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

A staff solution is:

```

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.

A staff solution is:

```

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`.

A staff solution is:

```

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.

A staff solution is:

```
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)
```


Bibliography

- [1] 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
- [2] 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. 50, 51
- [3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002. 52
- [4] S. Basu, C. Jacobs, and L. Vanderwende. Powergrading: a clustering approach to amplify human effort for short answer grading. *TACL*, 1:391–402, 2013. 62, 74
- [5] 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. 55
- [6] 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*

- Conference 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>. 58
- [7] 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. 61
- [8] D. Blei and J. Lafferty. Correlated topic models. *Advances in neural information processing systems*, 18:147, 2006. 188
- [9] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003. ISSN 1532-4435. URL <http://dl.acm.org/citation.cfm?id=944919.944937>. 46, 180, 184
- [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. 35, 63, 70, 153
- [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](http://dx.doi.org/10.1016/S0169-7552(97)00031-7). URL <http://www.sciencedirect.com/science/article/pii/S0169755297000317>. 58
- [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. 63, 104
- [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. 52
- [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>. 69
- [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>. 70
- [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. 69
- [17] J. Case. Education theories on learning: an informal guide for the engineering education scholar. 2008. 72
- [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. 66
- [19] J. Chang, S. Gerrish, C. Wang, J. L. Boyd-Graber, and D. M. Blei. Reading tea leaves: How humans interpret topic models. In *Advances in neural information processing systems*, pages 288–296, 2009. 186
- [20] J. C. Chen, D. C. Whittinghill, and J. A. Kadowec. 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. 64
- [21] 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. 63, 159
- [22] 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. 64

- [23] 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. 56, 60, 61, 62, 203
- [24] 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. 158
- [25] J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960. 173
- [26] 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>. 64
- [27] J. Dewey. How we think: A restatement of the relation of reflective thinking to the educational process. *Lexington, MA: Heath*, 1933. 35, 64
- [28] 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. 70
- [29] 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>. 67
- [30] A. Driver, K. Elliott, and A. Wilson. Variation theory based approaches to teaching subject-specific vocabulary within differing practical subjects. *Interna-*

- tional Journal for Lesson and Learning Studies*, 4(1):72–90, 2015. doi: 10.1108/IJLLS-10-2014-0038. 69
- [31] 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. 56
- [32] 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. 57
- [33] 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>. 56
- [34] R. M. et al. 6.005 software construction. 49
- [35] 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. 50, 51, 53, 61, 191
- [36] 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>. 58, 59
- [37] 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. 74
- [38] 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. 30, 61
- [39] E. L. Glassman, L. Fischer, J. Scott, and R. C. Miller. Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th annual ACM symposium on User Interface Software and Technology, UIST '15*, New York, NY, USA, 2015. ACM. 131
- [40] E. L. Glassman, A. Lin, C. J. Cai, and R. C. Miller. Learnersourcing personalized hints. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW '15*, New York, NY, USA, 2015. ACM. 153
- [41] E. L. Glassman, J. Scott, R. Singh, P. Guo, and R. C. Miller. Overcode: visualizing variation in student solutions to programming problems at scale. *Transactions on Computer-Human Interaction*, 2015. 25, 73
- [42] 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. 64
- [43] S. Gross, B. Mokbel, B. Hammer, and N. Pinkwart. Feedback provision strategies in intelligent tutoring systems based on clustered solution spaces. *DeLFI 2012: Die 10. e-Learning Fachtagung Informatik*, 2012. 60, 70
- [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>. 57, 197, 203

- [45] S. Gulwani, I. Radiček, and F. Zuleger. Automated clustering and program repair for introductory programming assignments. *arXiv preprint arXiv:1603.03165*, 2016. 58
- [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. 84, 91
- [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. 70
- [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>. 50, 51
- [49] E. W. Høst and B. M. Østvold. The java programmer’s phrase book. In *Software Language Engineering*, pages 322–341. Springer, 2008. 53
- [50] E. W. Høst and B. M. Østvold. Debugging method names. In *ECOOP 2009—Object-Oriented Programming*, pages 294–317. Springer, 2009. 53
- [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. 56, 59, 60, 61, 74, 93
- [52] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010. ISSN 0167-8655. doi: <http://dx.doi.org/10.1016/j.patrec.2009.09.011>. URL <http://www.sciencedirect.com/science/article/pii/S0167865509002323>. Award winning papers from the 19th In-

- ternational Conference on Pattern Recognition (ICPR)19th International Conference in Pattern Recognition (ICPR). 43, 44, 45
- [53] D. M. Jones. Operand names influence operator precedence decisions (part. 2008. 52
- [54] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani. Semi-supervised verified feedback generation. *arXiv preprint arXiv:1603.04584*, 2016. 56
- [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. 55, 58
- [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. 67
- [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. 67
- [58] B. Kim. Interactive and interpretable machine learning models for human machine collaboration, 2015. PhD thesis. 183
- [59] B. Kim, C. Rudin, and J. A. Shah. The bayesian case model: A generative approach for case-based reasoning and prototype classification. In *Advances in Neural Information Processing Systems*, pages 1952–1960, 2014. 180
- [60] 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. 35, 71, 154

- [61] S. Klemmer. Scaling studio critique: success, bruises, and future directions. MIT CSAIL Seminar, 2012. 71
- [62] J. L. Kolodner. Educational implications of analogy: A view from case-based reasoning. *American psychologist*, 52(1):57, 1997. 66
- [63] C. Kulkarni, J. Cambre, Y. Kotturi, M. S. Bernstein, and S. R. Klemmer. Talkabout: Making distance matter with small groups in massive classes. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work and Social Computing, CSCW '15*, pages 1116–1128, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2922-4. 71
- [64] 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. 70
- [65] 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, 56, 191
- [66] K. J. Kurtz, C.-H. Miao, and D. Gentner. Learning by analogical bootstrapping. *The Journal of the Learning Sciences*, 10(4):417–446, 2001. 67
- [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. 35, 66
- [68] 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. 61
- [69] 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>. 69
- [70] M. Ling Lo. *Variation theory and the improvement of teaching and learning*. Göteborg: Acta Universitatis Gothoburgensis, 2012. 68
- [71] 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>. 61
- [72] 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. 69
- [73] 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. 67
- [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. 35, 66
- [75] 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. 59, 60
- [76] A. Mabanta, C. Hunt, S. Najmabadi, and K. Ridenoure. How big is uc berkeley's biggest class?, 2013. <http://www.dailycal.org/2013/09/03/how-big-is-uc-berkeley-s-biggest-class/>. 25, 131

- [77] F. Marton and S. A. Booth. *Learning and awareness*. Psychology Press, 1997. 35, 67, 68
- [78] F. Marton and M. F. Pang. On some necessary conditions of learning. *The Journal of the Learning sciences*, 15(2):193–220, 2006. 69
- [79] F. Marton, A. B. Tsui, P. P. Chik, P. Y. Ko, and M. L. Lo. *Classroom discourse and the space of learning*. Routledge, 2004. 68
- [80] F. Marton, A. Tsui, P. Chik, P. Ko, and M. Lo. *Classroom Discourse and the Space of Learning*. Taylor & Francis, 2013. ISBN 9781135642334. 74, 78
- [81] 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>. 71
- [82] 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>. 69
- [83] 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. 50
- [84] 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. 71, 154
- [85] A. Muralidharan and M. Hearst. Wordseer: Exploring language use in literary text. *Fifth Workshop on Human-Computer Interaction and Information Retrieval*, 2011. 62
- [86] A. Muralidharan and M. A. Hearst. Supporting exploratory text analysis in literature study. *Literary and linguistic computing*, 28(2):283–295, 2013. 62
- [87] A. S. Muralidharan, M. A. Hearst, and C. Fan. Wordseer: a knowledge synthesis environment for textual data. In *CIKM*, pages 2533–2536, 2013. 62

- [88] 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. 25, 55, 61, 74, 202
- [89] 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. 69
- [90] F. G. Paas and J. J. van Merriënboer. 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/>. 66
- [91] 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. 70
- [92] T. Peters. The zen of python. In *Pro Python*, pages 301–302. Springer, 2010. 49
- [93] S. Pinker. *The sense of style: The thinking person’s guide to writing in the 21st century*. Penguin Books, 2015. 194
- [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. 66
- [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>. 56
- [96] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. <http://is.muni.cz/publication/884893/en>. 184

- [97] 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. 47
- [98] 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. 66
- [99] K. Rivers and K. R. Koedinger. Automatic generation of programming feedback; a data-driven approach. In *AIED Workshops*, 2013. 71
- [100] 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. 55
- [101] 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>. 59
- [102] S. Rogers, D. Garcia, J. F. Canny, S. Tang, and D. Kang. Aces: Automatic evaluation of coding style. Master’s thesis, EECS Department, University of California, Berkeley, May 2014. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-77.html>. 56, 59, 60, 61, 181, 184
- [103] J. M. Rzeszotarski and A. Kittur. Crowdscape: interactively visualizing user behavior and output. In *UIST*, pages 55–62, 2012. 62
- [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. 57, 197
- [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. 58
- [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. 59
- [108] V. J. Shute. Focus on formative feedback. *Review of educational research*, 78(1): 153–189, 2008. 65
- [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>. 204
- [110] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013. 70
- [111] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, Sept. 1984. ISSN 0098-5589. 55
- [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. 168
- [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. 54

- [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. 56, 59
- [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. 69
- [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. 56, 57, 197
- [117] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Hierarchical dirichlet processes, 2005. URL <http://www.cs.berkeley.edu/~jordan/papers/hdp.pdf>. 188
- [118] S. Terman. *GroverCode: Code Canonicalization and Clustering Applied to Grading*. PhD thesis, MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 2016. 73, 120, 123, 126
- [119] K. Topping. Peer assessment between students in colleges and universities. *Review of Educational Research*, 68(3):pp. 249–276, 1998. 71
- [120] 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. 35, 64
- [121] 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. 66
- [122] 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. 66
- [123] 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. 157, 168
- [124] A. J. Viera and J. M. Garrett. Understanding interobserver agreement: the kappa statistic. *Fam Med*, 37(5):360–363, 2005. 173
- [125] L. Vygotsky. Zone of proximal development. *Mind in society: The development of higher psychological processes*, 5291, 1987. 157
- [126] H. J. Walberg. Improving the productivity of america’s schools. *Educational leadership*, 41(8):19–27, 1984. 72
- [127] H. Wallach. Big data, machine learning, and the social sciences: Fairness, accountability, and transparency, 2014. URL <https://medium.com/@hannawallach/big-data-machine-learning-and-the-social-sciences-927a8e20460d#.tls6r9ixy>. 191
- [128] 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. 65
- [129] 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. 71, 154, 155
- [130] 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 *Pro-*

ceedings of the 4th Human Computation Workshop (HCOMP '12) at AAI, 2012.

71

- [131] Wikipedia. Cluster analysis — Wikipedia, the free encyclopedia, 2016. URL https://en.wikipedia.org/wiki/Cluster_analysis. [Online; accessed 4-August-2016]. 45
- [132] 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. 65
- [133] 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. 70
- [134] R. Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on neural networks*, 16(3):645–678, 2005. 45, 46
- [135] 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. 55
- [136] 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. 59