Clustering and Visualizing Solution Variation in Massive Programming Classes

by

Elena L. Glassman

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2016

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

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

Accepted by . . . .

Chairman, Department Committee on Graduate Students

Clustering and Visualizing Solution Variation in Massive Programming Classes

by

Elena L. Glassman

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

Abstract

In 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, who have each supported me in their own way, from hugs, to proofreading papers, to being technical sounding-boards, and finally, to demonstrating, as my brother has, how to switch fields, pick up a whole new set of skills, and look good doing it.

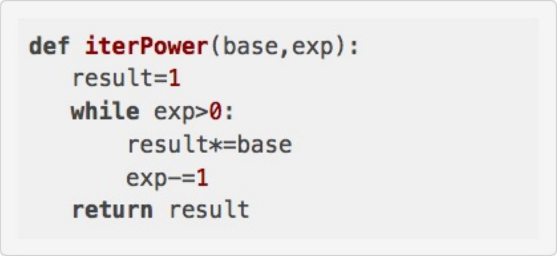
Chapter 1

Introduction

Introductory programming classes are big and hard to teach. Programming classes on some college campuses are reaching hundreds or thousands of students [**?**]. Massive Open Online Courses (MOOCs) on programming have drawn tens or hundreds of thousands of students [**?**]. 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 **??** represents 1538 solutions [**?**]. 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 **??** 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 **??** 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 **??** 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
```

Recursive Solution

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

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
```

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

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

Solution with Unnecessary Statement

```python
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

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

Uncommon solution

```python
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 **??** 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`
- recursive 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 **??**.

1.2 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 [**?**]. 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.

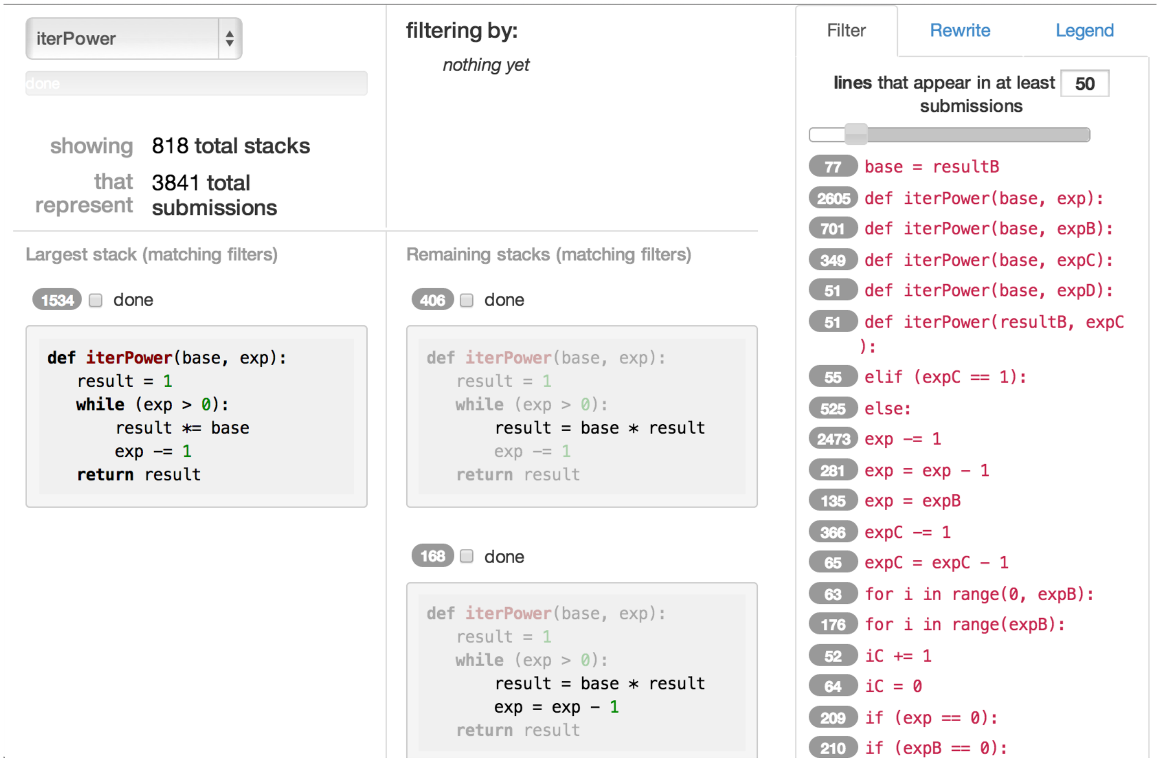
OverCode, shown in Figure **??** renders thousands of small student Python solutions to the same problem and makes them explorable. With OverCode, teachers can better understand the distribution of common and uncommon solutions without reading every student solution. The OverCode analysis pipeline uses 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 studys, 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 **??**, 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



showing 818 total stacks

that represent 3841 total submissions

filtering by:

nothing yet

Largest stack (matching filters)

1534 □ done

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

Remaining stacks (matching filters)

406 □ done

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

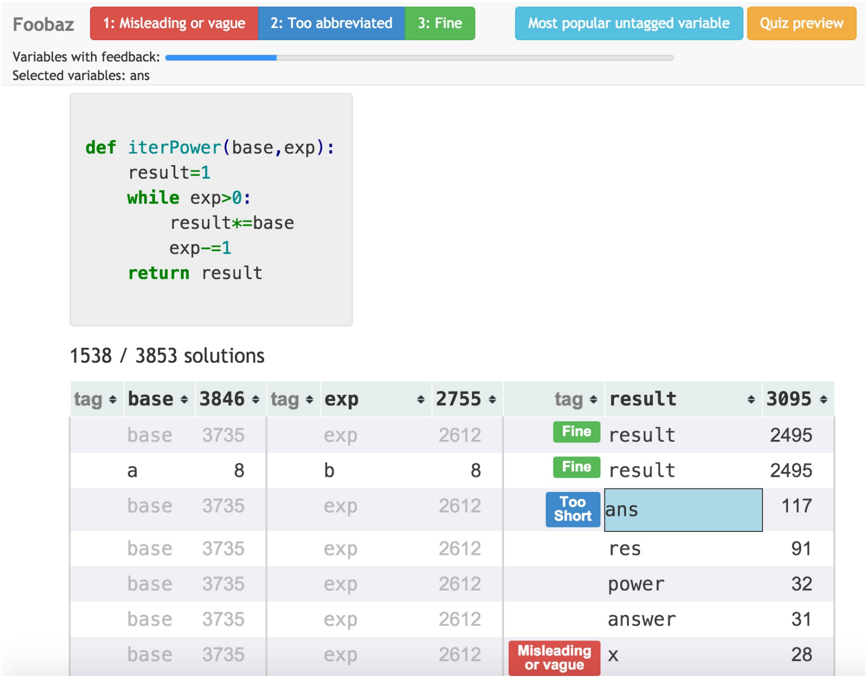
168 □ done

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

lines that appear in at least 50 submissions

77	base = resultB
2605	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.



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

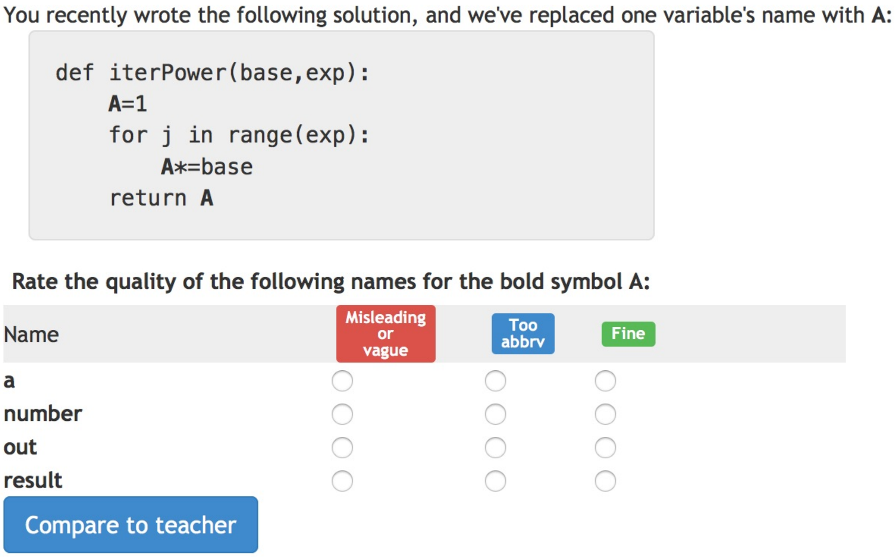
1538 / 3853 solutions

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 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 **??**. 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 abbrv	Fine
a	○	○	○
number	○	○	○
out	○	○	○
result	○	○	○

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 [?]. 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 **??**.

Unlike prior incarnations of assigning tasks to and collecting data from students, i.e., learnersourcing [?], 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 [?] and currently lacking in the engineering education status quo [?]. 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 [?] and analogical learning theory [? ?].

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 teachersâĂŹ personalized assistance.

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.

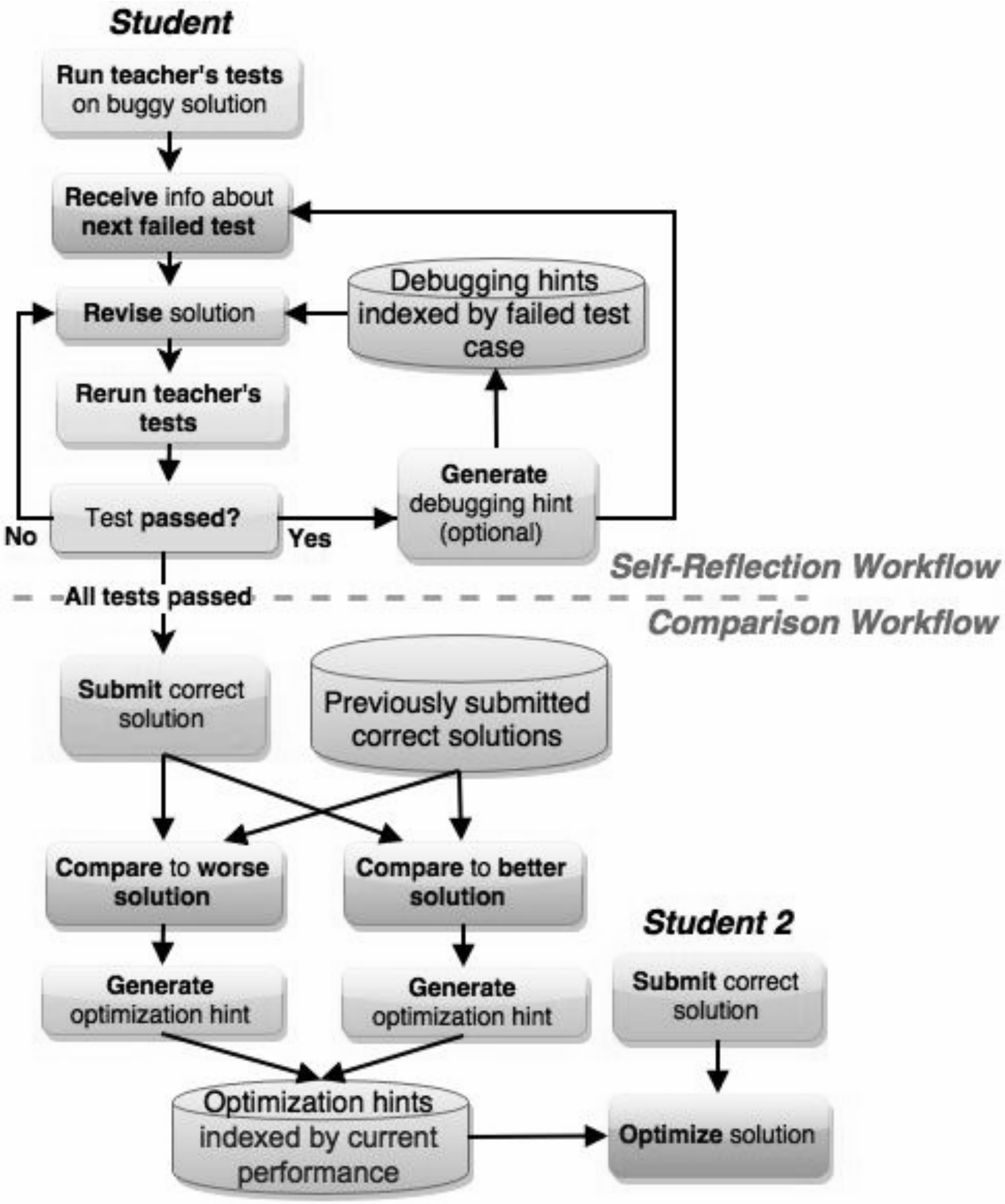

Student
Run teacher's tests on buggy solution
Receive info about next failed test
Revise solution
Debugging hints indexed by failed test case
Rerun teacher's tests
Test passed?
No
Yes
Generate debugging hint (optional)
Self-Reflection Workflow
All tests passed
Comparison Workflow
Submit correct solution
Previously submitted correct solutions
Compare to worse solution
Compare to better solution
Student 2
Generate optimization hint
Generate optimization hint
Submit correct solution
Optimization hints indexed by current performance
Optimize solution

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 **??** 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 **??**) 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 **??**) 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 **??** 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 **??** describes Bayesian clustering and mixture modeling algorithms applied to the OverCode pipeline output for extracting additional insight into patterns within student solutions.

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