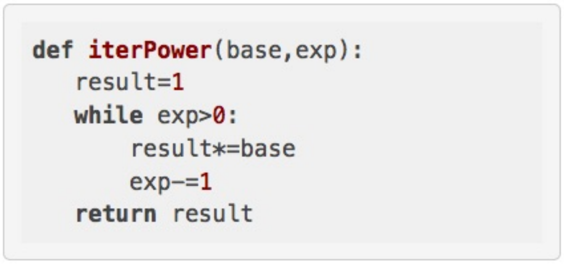
# Chapter 1

# Introduction

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

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



```
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.

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 platonic solutions that each represent entire stacks of solutions. For example, the solution shown in Figure 1-1 represents 1538 solutions [? ]. OverCode is the first of several systems for clustering and visualizing solutions and solution variation that were developed in this thesis.

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

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

## 1.1 Solution Variation

~~Learners individually solving programming or digital hardware design problems can collectively generate a wide variety of possible bugs and solutions. A single programming or digital~~

~~hardware design exercise may yield thousands of student solutions that vary in many ways, some superficial and some fundamental. For teachers, this variation can, for example, be a source of pedagogically valuable examples and corner casesthat slipped past an automatic grader. For students, the variation in a large class means that other students may have struggled along a similar solution path, hit the same bugs, and can offer hints based on that earned expertise~~The taxonomy for *solution variation* used in this thesis has three branches: 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.

~~Understanding large-scale variation in student solutions requires identifying appropriate features of solutions,developing ways to automatically extract those features from each solution, user interface design to communicate the results to teachers or students, and possibly the collection of additional information from students through learnersourcing. This thesis includes the development of four systems that take advantage of the solution variation in large classes. All four systems have been evaluated using data or live deployments in on-campus or edX courses with hundreds or thousands of students~~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.

~~This work began after midnight in the basement computer lab of the Stata Center, where a student was struggling to create a Turing Machine that could detect whether a string of open and closed parentheses was properly balanced. If he was not successful within the next day or two, he would fail Computer Architecture (6.004). I will refer to him as Paul. I had a hard time helping Paul turn his Turing machine into one that would pass all the automatic grader's test tapes. I could not tell if his approach could work with a little debugging or was fatally flawed. I did not want to demoralize him by telling him to abandon his efforts so far and start over unless it was truly necessary. And what if he was on the path to a novel solutionno one elsehad created before?~~

~~While not a necessary condition for the correctness of his intended approach, finding a similar working solution in the pool of previous student submissions would serve as an existence proof and evidence that he should persevere. I had copies of hundreds of other students' correct Turing machines, but it was difficult to see by inspection whether any of these solutions were successful implementations of his intended approach~~

~~) students design their own symbol library and state names for the finite state machine portion of their Turing machine (2)students can list behavioral specifications in terms of their custom symbol and state names in any arbitrary order. Each behavior-specifying rule is a computer-readable version of the form "if you're in state X and you read symbol Y on the tape, overwrite symbol Y with symbol Z, move the tape-reader in direction W, and transition to state Q." Dynamic superficial differences arise because even after translating these textual statements into a finite state machine diagram, the diagrams look very different from one correct solution to the next.~~

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. ~~This was due to both superficial static and dynamic differences between solutions. There are two sources of static superficial differences: (~~

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

**Iterative Solution**

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

**Recursive Solution**

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

**Poorly Written Solution**

```
\DIFaddFL{def power(base, exp):
    tempBase=base
    result = base
}\DIFaddendFL

     \DIFaddbeginFL \DIFaddFL{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)
}

    \DIFaddFL{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**

```
\DIFaddFL{def power(base, exp):
  return base**exp
}
```

**Solution with Unnecessary Statement**

```
\DIFaddFL{def power(base,exp):
 result=}\DIFaddendFL 1
  \DIFaddbeginFL \DIFaddFL{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**

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

**Uncommon solution**

```
\DIFaddFL{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.

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 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.

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

  \DIFaddFL{returns: int or
     float, base^exp
  '''
  result = 1
  while exp > 0:
      result *= base
      exp -= 1
  return result
}
```

```
\DIFaddFL{def iterPower(base, exp):
  wynik = 1
}

  \DIFaddFL{while exp > 0:
      exp -= 1  #exp argument is counter
}

      \DIFaddFL{wynik *= base
}

  \DIFaddFL{return wynik
}
```

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

*Student design choices* affect correctness, approach, and readability. Examples include choosing `for` over `while`, `a *= b` over `a = a*b`, and recursive over 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.

~~After the semester ended, I looked into this further. I ran all the correct two-state Turing machinescollected during the semester on the same test tape containing a string of open and closed parentheses. The movement of the tape-reading head across this input was logged in coordinates relative to the common starting point, at the left end of the test tape, and displayed along the vertical axis. The horizontal axis represents the number of steps taken by the Turing machine on its way to completing the task. These discrete steps are analogous to time.~~

## 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 ~~majority (88%) of the solutionsemploy one of two mutually exclusive strategies. These strategies were identified just from visual inspection of the movement of many Turing machines across the common test tape. Figure **??** shows their locations on the common input tape over time, separated by strategy into Figures **??** and **??**. These two strategies for determining whether or not the tape's string of parentheses is balanced are (1) matching the innermost open parenthesis with the innermost closed parenthesis, i. e., standard~~

~~semantic interpretation in math, and (2) matching the $n^{th}$ open parenthesis with the $n^{th}$ closed parenthesis. The remaining 12~~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 ~~(not shown) include~~ included less common strategies. ~~This analysis also exposes room for improvement in the suite of test tapes: at least two strategies in this group are known to pass the test tapes but are actually incorrect because they cannot handle an arbitrary depth of nested parentheses.~~

~~Staff members were aware of the Turing machine solution they each had found on their own time, and yet were~~ Two solutions passed all test cases but subverted teacher instructions. Many teaching staff members were not aware that there were ~~two mutually exclusive common solutions~~ 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.

~~Based on my analysis, I was able to brief fellow 6.004 staff members on the space of solutions–both good and bad–in preparation for subsequent semesters and could suggest additions to the test tape suite to catch submissions that were previously erroneously marked as correct.~~

## 1.4 Research Questions

~~This experience was a critical and formative one for me. It has inspired the design approach and technical approach taken in many of the systems that follow.~~ 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.

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

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

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

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

~~In MOOCs, a single programming exercise may produce thousands of solutions from learners. Understanding solution variation is important for providing appropriate feedback to students at scale. The wide variation among these solutions can be a source of pedagogically valuable examples, and can be used to refine the autograder for the exercise by exposing corner cases. We developed OverCode to visualize and explore thousands of small Python programs that solve the same problem.~~ The OverCode and Foobaz systems both give teachers a better understanding of student solutions and how they vary in approach and readability. OverCode ~~uses both static and dynamic analysis to cluster similar solutions , and lets teachers further filter and cluster solutions based on different criteria. We evaluated OverCode against a non-clustering baseline in a within-subjects study with 24 teaching assistants, and found that the OverCode interface~~ allows teachers to more quickly develop a high-level view of student understanding and misconceptions~~, and to provide feedback that is~~. 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.

OverCode also helps answer 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.

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

~~Traditional feedback methods, such as hand-grading student code for substance and style, are labor intensive and do not scale. We created a new user interface that addresses feedback at scale for a particular and important aspect of code quality~~The research question **R3** asks how we can personalize feedback for individual students because personalization, in the form of one-on-one tutoring, has been a gold standard in educational psychology for decades [**?** ]. For 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~~(see Figure **??**). Foobaz distinguishes variables by their behavior in the program, allowing teachers to comment not only on poor names, but also on names that mislead the reader about the variable 's role. We ran two lab studies of Foobaz, one with teachers and the other with students. In the first study, 10 Python teachers used Foobaz to comment on variable names in thousands of studentsolutions from an introductory programming MOOC. In the second study, 6 students composed fresh solutions to the same programming problems, and immediately received personalized variable-name quizzes composed in the~~

~~previous user study.~~

~~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."~~

~~In order to perform more comprehensive clustering, several extensions of the original OverCode clustering methods have been explored and some have been fully implemented. GroverCode is an extension of OverCode built to canonicalize and cluster both correct and incorrect solutions~~ . ~~This was also done in the context of hundreds of exam-level solutions instead of thousands of solutions to "finger exercises" and problem set problems.This change in regime was inspired by need-finding interviews with the lecturer who teaches both the online and residential versions of MIT's introductory Python programming class. She was appreciative of the insights OverCode could give her extracted from the thousands of correct solutions submitted by her online students , but the semi-regular pain of hand-grading the hundreds of incorrect solutions submitted by tuition-paying residential students was a more immediate concern. Adapting to the needs of a team of graders grading a smaller number of more complex solutions, many of which were incorrect, required significant changes to the pipeline and interface. The resulting system was deployed to the staff for use during two grading sessions. The results of this field deployment are also promising; the staff welcomes its continued use in future semesters. More preliminary explorations of statistical methods for clustering solutions are also described.~~ 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. These exercises are called *variable name quizzes*. Foobaz helps answer **R3** by demonstrating how teachers can compose automatically personalized feedback on an aspect of readability to students at scale.

The final research question, **R4**, is addressed by two novel workflows that collect and deliver personalized hints. 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. The two workflows are built on that insight. Unlike prior incarnations of assigning tasks to and collecting data from learners, 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 [? ? ].

~~Personalized support for students is a gold standard in education, but it scales poorly with the number of students. Prior work on *learnersourcing* presented an approach for learners to engage in human computation tasks while trying to learn a new skill. Our key insight is that students, through their own experience struggling with a particular problem, can become experts on the particular optimizations they implement or bugs they resolve. The students can then generate hints for fellow students based on their new expertise. ClassOverflow uses new workflows to harvest and organize students' collective knowledge and advice for helping fellow novices through design problems in engineering (see Figure ??). ClassOverflow was evaluated in an undergraduate digital hardware design class with hundreds of~~

## 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 ~~. We show that, given our design choices, students can create helpful hints for their peers that augment or even replace teachers' personalized assistance, especially when that assistance is not available.~~ can be prompted to generate analysis of their own and others' solutions, for the benefit of themselves and current and future students.

~~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.~~

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.

~~Since terminology across research domains can vary, I will define the terms in which I will describe previous research and my own~~The main contributions of this thesis are:

- ~~A solutionis code that a particular person wrote in response to a prompt or problem description.~~ 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.
- ~~Solution clustersrepresent different patterns of implementation. For example, there may be two distinct solution clusters, both achieving the same input-output behavior but by different means.~~
- A ~~solution pathis a series of code snapshots generated while a person is working toward meeting a particular input-output behavior specification.~~
- ~~The space of student solutionsrefers to the aggregation of student-generated solutions~~

~~and the solution clusters they form. Learnersourcing...~~

~~The main contributions of this thesis are:~~

- ~~A~~ novel visualization that ~~shows~~ highlights similarity and variation among thousands of Python solutions ~~, with cleaned code shown~~ while displaying platonic solutions for each variant.
- ~~An algorithm that uses the behavior of variables to help cluster Python solutions and generate the cleaned code for each cluster of solutions.~~
- Two user studies that show this visualization is useful for giving teachers a ~~birds-eye~~ 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 ~~a slice~~ an aspect, i.e., variable names and roles, of each cluster exposed, revealing the ~~features~~ details that are relevant to the task. ~~In this application, the relevant features are variable names and roles.~~
- 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.
- ~~A working system which implements~~ An implementation of the above technique and method for ~~datasets from both MOOCs and large residential classes on introductory Python programming.~~ variable naming.
- Two lab studies which evaluate both the ~~teachers' and students'~~ teacher and student experience of the ~~variable name feedback workflow .~~ 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, in detail, the four systems developed, as well as their evaluation on archived data or in the field.

- OverCode (Chapter **??**) visualizes thousands of programming solutions using static and dynamic analysis to cluster similar solutions. It lets teachers quickly develop a high-level view of student understanding and misconceptions and provide feedback that is relevant to many student solutions. It also describes GroverCode, an extension of OverCode optimized for grading correct and incorrect student solutions.
- Foobaz (Chapter **??**) clusters variables in student ~~programs~~ 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.
- ~~ClassOverflow (Chapter **??** ) collects and organizes~~ Chapter **??** describes two workflows that collect and organize solution hints indexed by (1) the autograder test that failed or (2) a performance characteristic like size or speed. It helps students reflect on their debugging or optimization process, generates hints that can help other students with

the same problem, and could potentially bootstrap an intelligent tutor tailored to the problem.

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

Chapter **??** discusses ~~the design lessons that apply to the entire collection of~~ some of the insights that came out of building and testing the systems in this thesis. Chapter ~~**??**~~ **??** outlines avenues of future work ~~this thesis work~~on the systems and ideas in this thesis, in combination with the complementary work of others in this space~~, paves the way for.~~

.