Mining Student-Generated Alternative Implementations

Elena Glassman
MIT CSAIL
elg@mit.edu

11th June, 2014

Abstract

This is an HCI approach to dealing with multiple solutions generated by students. Many coding problems have a specification that the student's solution must meet, but give the student a broad range of freedom for the internal design of that solution. There may be several distinct, correct solutions, some of which may be unknown to the teaching staff or intelligent tutor designer, making it potentially difficult for staff to help students reach their own correct solutions. My approach will be to visualize hundreds or thousands of student solutions to discover alternatives, and then classify the solutions, and the paths that students take to each solution, either by machine learning or human staff. The visualization and classifications will inform assistance given to students, in the form of staff-student discussions, peer-pairing, and/or automated help.

1 INTRODUCTION

Many coding problems, in the context of a course or online game, have a specification that the student's solution must meet, but give the student a broad range of freedom for the internal design of that solution. There may be several distinct, correct solutions, some of which may be unknown to the teaching staff or intelligent tutor designer. This raises problems for helping students and giving feedback. In face-to-face situations, if teaching

assistants don't recognize the students' solution paths, then they may redirect the students completely, costing them work and possibly derailing novel, valid solutions. In an intelligent tutor or massively open online course (MOOC), the automated hint generators may not recognize the unexpected solution paths, and will generate unhelpful hints.

This research aims to first develop tools and principles that support teachers' knowledge of the design space their students inhabit. This knowledge can enhance teachers' in-person interactions with students, by inspiring conversations about alternative design possibilities. This knowledge can also inform teacher-trained classification algorithms that automatically identify and send feedback and guidance to particular students. The feedback and guidance would be written by the teachers in response to particular types of solutions, but once classifiers can reliably identify those types of solutions, the same feedback and guidance can be applied to other current students, and to future students. These solution-classifiers could also inform peer-pairing, when students must help one another.

I first will discuss related work from the fields of active (machine) learning, computer science education, and the learning sciences. I will then explain the chosen thesis problem, and describe the research methods selected for creating a solution to this problem. Interspersed with these research methods will be the descriptions of preliminary experiments which influenced the research method selections. Finally, I will conclude by laying out my expected contributions and a timeline for completion.

2 BACKGROUND & RELATED WORK

Since terminology across research domains can vary, I will define the terms in which I will describe previous research and my own:

- A solution is code that a particular person wrote in response to a prompt or problem description.
- Solution clusters represent 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 path is a series of code snapshots generated while a person is working toward meeting a particular input-output behavior specification.

- The "space of student solutions" refers to the aggregation of student-generated solutions and the solution clusters they form.

2.1 Comparing and Contrasting Examples

Marton et al.'s variation theory [?] holds that in order to learn something, one must see examples that vary along particular dimensions: "contrast," as in pairing it with something it is not; "generalization," as in presenting multiple instances of the object or concept to be learned, varying only that which is irrelevant; "separation," as in presenting multiple instances of the object or concept, varying only that which can vary internally without changing the object or concept into something else; and "fusion," as in seeing multiple examples in which previously analytically separated aspects must be processed together to recognize the object or concept. The aspects which are related to these dimensions of variation and therefore define the object or concept are called "critical features."

Peer reviews and assessments, surveyed in [?], are one of the existing pedagogies in which teachers ask students to compare and constrast examples. The pedagogical method of comparing and contrasting ways of approaching a solution has now been validated in the literature of mathematics education research [?], cognitive science [?, ?, ?], and computing education research [?, ?].

Given Marton et al.'s rubric for effective patterns of variation, and the identification of "critical features," one can discern between more or less theoretically effective examples of the object or concept given to a student to learn. On this basis, Luxton-Reilly et al. [?] suggest that identifying distinct clusters of solutions can help instructors select appropriate examples of code for teaching purposes.

While this research has focused on the effects of multiple, varying examples on student learning, it is also, as suggested by Luxton-Reilly et al. [?], helpful for teachers' own understanding and quality of feedback and guidance. Facilitating the discovery or identification of critical features, which are possibly both teacher-specific and task-specific, is a major challenge I will address in this thesis.

2.2 Feature Engineering

In order to discover or identify critical features, it is necessary to generate a set of candidate features. A variety of methods have been employed in the literature to select or engineer features, but those which I describe here are focused on features that humans can perceive by looking at the actual code submission.

2.2.1 Abstract Syntax Trees, Dependency Graphs, and Control Flow Graphs

Aggarwal et al. [**?**] address the critical nature of feature engineering in the context of machine-learning-based automated grading. The grading rubric is based on the authors' understanding of how humans perceive and assess programs. They describe how humans first look for certain "signature features," such as certain control structures, dependencies, and keywords. If the necessary features are in place, then more fine-grained assessment can be made. Are the correct structures used, and are statements ordered properly? This increasingly fine-grained assessment can continue on, to include terminating conditions and dependencies between data structures, until the human is satisfied.

Aggarwal et al. suggest extracting these features from a code submission's Abstract Syntax Tree (AST), Control Flow Graph, Data Dependence Graph, and/or Program Dependence Graph. The authors assert that these graphs will be helpful even if the code represents only a partial solution. Note that these features are targeted at labeling each solution with a numerical score based on correctness, not on disguishing between equally correct solutions representing different approaches to a problem.

2.2.2 Thematic Analysis

Instead of grading submissions based on a rubric of human acceptability and correctness, Luxton-Reilly et al. [**?**] used thematic analysis to capture the variation between correct solutions in their dataset. Thematic analysis comes out of the field of psychology as a way to build theory from observing patterns in organic, free-form statements from subjects, or, in this case, submissions for coding assignments. Braun and Clarke [**?**] argue that its application to qualitative data outside psychological research is justified. It is in direct

contrast to methods in which a hypothesis or theory is first declared, and then evidence for and against it is gathered from the data.

Luxton-Reilly et al. [?] aim to discover the kinds and degree of variation between student-generated solutions that fulfilled the specifications of short introductory Java programming exercises. By thematic analysis of student submissions, the authors generated a taxonomy that captures the variation between correct solutions in their dataset. They then created an Eclipse plug-in for classifying new code examples based on their taxonomy.

2.2.3 Compiler Concepts

Some of the Java exercises in the corpus studied by Luxton-Reilly et al. [?] were as simple as writing a function which takes two integers and returns their sum. Within these simple tasks, variation was still found in the way students used parentheses, declared and initialized variables, and made assignments. In order to use established, non-ambiguous terms for their observations, Luxton-Reilly et al. reference compiler concepts, such as tokens, classes of tokens, and control flow graphs.

They labeled types of variations as structural, syntactic, or relating to presentation. The control flow graphs represent structural variation. The nodes of control flow graphs are blocks of code that have a single entry point, single exit point, and no internal branching. The flow between blocks of code is represented by the edges connecting the control graph nodes. If the control graph (structure) of two solutions is the same, then the syntactic variation within those blocks of code are compared by looking at the sequence of token classes. Presentation-based variation, such as variable names and spacing, is only examined when two solutions are structurally and syntactically the same.

Luxton-Reilly et al.'s Eclipse plugin takes as input a collection of Java source files and creates a library of structurally unique Java code, which becomes the categories of files. On their corpus, they found a small number of highly populated categories, which did not always line up with the category of the instructor's implementation.

2.2.4 Adding Input-Output Behavior

Huang et al. [?] considered tens of thousands of solutions submitted to Stanford's Fall 2011 Machine Learning MOOC, and identified clusters of solutions

based on measures of syntactic but also functional similarity. The syntactic similarity was the edit distance between solutions' ASTs, using the tree edit distance function described in Shasha et al. [**?**]. Code submissions were also grouped by their success or failure on a battery of unit tests (input-output behavior). By pairing behavioral descriptions with structure-based distance measures between submissions, the authors got a fine-grained breakdown of submissions, across correctness and structure.

2.2.5 Program Comprehension

Yet another field is relevant when looking for the internal design variation of identically behaving solutions. Over the course of several papers, culminating in their most recent article [**?**], Taherkhani et al. have drawn from the field of program comprehension to develop several algorithm recognition methods. Two methods feature prominently in their work: (1) a method that creates a feature vector for the *predefined target solution* based on roles of variables, beacons, and various other software metrics and (2) a method that scans solutions for *predefined schemas* that are associated with algorithms of interest. In [**?**], Taherkhani et al. have combined the two approaches into a more robust version which only compares software metrics and beacons on the code that have the same schema. While the software metrics used in these approaches are relevant to my thesis, the use of predefined schemas and target solutions does not fit in with my approach of purely mining student solutions.

2.2.6 Clone and Plaigiarism Detection

Recognizing algorithms is similar to clone detection, which is also associated with plaigiarism detection. There are multiple types of clones. The simplest clone is an exact copy of another section of code. A parameter-substituted clone only differs from its copy by the values of identifiers and literals. A structure-substituted clone is a copy with something new swapped in for an entire subtree of the syntax tree. The most sweeping algorithm recognition methods can identify clones that are modified beyond just structural substitutions [**?**]. These latter two types of clones are difficult to identify with current state-of-the-art techniques [**?**, **?**].

One strategy which has worked well for discovering similar code segments in larger pieces of source code is document fingerprinting [**?**], where func-

tions of small sections of code are considered "fingerprints." A fingerprint is constructed by computing hashes for all n-grams in a document (for some chosen n). Similar code will contain similar fingerprint components. These fingerprints are potential features for classifying or clustering code.

2.3 Mathematical Modeling Applied to Solutions

While critical features are chosen based on their ability to enhance teachers' understanding or guide the choice of example solutions, it is also necessary to select features that support classification or clustering. Specifically, it is necessary to select features that support classification or clustering that is understandable and acceptable to the human in the loop.

2.3.1 Active and Interactive Machine Learning

On his interactive machine learning (IML) course webpage, Dr. Brad Knox describes IML as "machine learning with a human in the learning loop, observing the result of learning and providing input meant to improve the learning outcome." Active learning is a subset of semi-supervised machine learning in which the algorithm can query the human in the loop. Active/interactive machine learning techniques, which can take advantage of human experts in the loop to resolve uncertainties, have been deployed for de-duplication in Stonebraker et al.'s Data Tamer [**?**] and in a cardiac ECG-based alarm system [**?**]. The work on applying interactive machine learning to the educational context is not as mature, but actively being pursued. For example, Basu et al. [**?**] have simulated an interactive machine learning framework for helping teachers grade large numbers of clustered free response textual answers.

2.3.2 Probabilistic Approaches

Using automated classification methods, Piech et al. [**?**] found distinct development paths students take to achieve working solutions that fulfilled the specifications of short introductory programming exercises in Java. Students' incremental paths were classified by pipeline that included milestone discovery, Hidden Markov Modeling of the students' process, and clustering of solution paths. These identified paths are visualized as finite state machine transition diagrams. The evaluation focused on predicting midterm exam grades and detecting milestone difficulty.

Sudol et al.'s new metric, Probabilistic Distance to Solution, and its successful application to introductory programming exercises, is a second example of the feasibility of classifying and mapping out distinct paths to solutions [**?**]. After using a Markov Model to generate a "problem state graph," the authors applied their Probabilistic Distance to Solution (PDS) metric to the graph to estimate the number of states between an observed program model and the model of a correct solution.

2.3.3 Learning Students' Process and Behavior

The following examples highlight research that is further from relevance to this thesis because the paths to a working solution are classified by behavior rather than the type of final solution found. Kiesmueller et al. [**?**] attempted to recognize strategies at a very high level, which are not specific to the challenge at hand. Example high-level problem-independent strategies were a top-down or bottom-up programming style. Helminen et al. [**?**] introduced novel interactive graphs for examining the problem solving process of students working on small programming-like problems. However, problems with multiple solutions were outside the scope of their investigation.

2.4 Feedback to Students

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 [**?**], and Klemmer et al. demonstrate peer assessments' scalability to large online design-oriented classes [**?**].

Generating tailored feedback to students in large classes tackling problems even as short as introductory programming assignments requires many man-hours of repetitive work. Singh et al. [**?**] are pushing the state of the art of automated feedback for short introductory programming assignments. However, their software is currently only differentiating between solutions based on their input-output characteristics. For example, this system cannot currently differentiate between two different sorting algorithms. If there are common dead-ends that have been identified by looking at incorrect student solutions to a particular problem, by hand, this system can identify that a student is very close to a known dead-end approach, but it cannot iden-

tify *which* functionally equivalent variant of a correct solution a student is approaching.

Singh et al.'s automated feedback represents one end of the spectrum for providing tailored feedback to students because hints are algorithmically generated. Luxton-Reilly et al. [?], Huang et al. [?], and Basu et al. [?] represent the other end, by "force multiplying" human-generated feedback or "powergrading." By clustering syntactically similar solutions which fail on the same input-output tests, Huang et al. aim to enable the sending of appropriate teacher-written feedback to entire clusters of solutions. Basu et al. [?] focus their work on grading short textual free-response questions, but the idea of reducing the number of actions necessary for the expert labeler is the same.

3 STATEMENT OF THESIS/PROBLEM

Visualization and classification of the multiple solutions that students generate will improve hints and answers to students' questions, whether they are provided by peers, staff, or automation.

I plan to explore the following aspects of this claim:

- What features are useful for visualizing or automatically classifying alternative solutions?
- How do we facilitate teachers' understanding of the design space generated by students?
- How do we get students to think about alternatives not taken?
- How can peers help each other when there are multiple good solutions?
- How can we provide automated help based on archived solutions?

4 RESEARCH GOALS & METHODS

The approach will be to visualize hundreds or thousands of student solutions to discover alternatives, and then classify the solutions, and the paths that students take to each solution, either by machine learning or human staff. The classifications will inform assistance given to students, in the form of staff-student discussions, peer-pairing, and/or automated help.

4.1 Exploratory Data Analysis

As a researcher, my first step toward discovering the space of correct solutions for a given assignment will be to look at the raw data. I have access to anonymized code submissions from the Spring 2013 semester of both a virtual hardware (6.004) and software (6.005) design class. I also have access to submissions to an online Matlab programming game, Python submissions to Introduction to Computer Science and Programming (6.00x), and also potentially the Python submissions to Introduction to Electrical Engineering and Computer Science I (6.01).

By poring over many examples of students' code, I can get a sense for what design decisions partition the space of implemented solutions into useful clusters. Useful, in this case, refers to clusters which help me conceptualize the space of solutions and identify where new students fall into that space.

I have also given randomly chosen subsets of correct solutions to teaching staff or knowledgeable experts, for clustering by hand. By studying the ways in which humans, given the same set of solutions, come up with different partitions of the solution space, I can assess which features I need in order to support any or all of those distinctions.

I have already taken this approach to an assignment in 6.004, an undergraduate computer architecture course in which virtual hardware is designed at the MOSFET and CMOS gate level and coded in a variant of SPICE called JSIM. Note that, when looking at raw JSIM code submissions to 6.004, human-made clusterings had little relationship to basic measurements, like the number of MOSFETs instantiated. This is unsurprising, since solution size is not readily apparent when looking at the raw code submitted. When I clustered solutions while knowing their relative size, size was a very helpful variable for clustering, and contributed heavily to my mental model of the design space. The relationship between the representation of solutions, in the form of measurements or features, and the human's mental model built while examining solutions, is the first of several reasons why HCI is the approach being taken for this problem.

This process was simultaneously performed by Rishabh Singh on 6.00x Python submissions. 6.00x is a Massively Open Online Course, or MOOC, that serves as an introduction to programming. The MOOC platform provided a web-interface for students to write and test their code in the browser itself, and students were provided 30 attempts to submit their solutions for each of the exercises. Singh obtained thousands of submissions to the course

Problem Description	Total Submissions	Correct Submissions
`comp-deriv`	3013	1433
`hangman-guess`	1746	1118
`iter-power`	8940	3875

Table 1: Dataset of 6.00x problems from Fall 2012.

exercises during the Fall 2012 semester. He filtered out all incorrect submissions, as indicated by input-output testing, so that we could focus on variations in correct solutions. The number of correct submissions analyzed for each problem is shown in Table. **??**.

The `comp-deriv` and `hangman-guess` problems were part of week 3 problem set exercises, whereas `iter-power` was an in-lecture exercise for the lecture on iteration. The `comp-deriv` 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. The `hangman-guess` 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. The `iter-power` problem asks students to write a function to compute the exponential $\texttt{base}^{\texttt{exp}}$ iteratively using successive multiplication.

He also gave randomly chosen subsets of correct solutions to knowledgeable experts, for clustering by hand. Regardless of the clustering(s) found, the intuition gained through this first exploratory step, akin to thematic analysis [**?**], guide the steps that follow.

4.2 Feature Engineering

Feature engineering is a critical challenge; the "critical features" discussed in the learning sciences literature are not yet known, and may be domain, task, and teacher dependent. This is another reason for the HCI approach of this thesis: there will need to be a tight loop between the human, presumably a member of the teaching staff, and the representation of the solutions, in order to extract meaning.

4.2.1 Function Calls, Solution Size, and Control Flow

Working in collaboration with Singh, we now generate feature vectors for Python programs that include the size of different sub-trees of the abstract syntax tree (AST), the control flow of the program, and the usage of Python libraries and constructs. In total, we computed around 60 features for the Python programs. Some of these features include: number of program variables, number of conditional and loop statements, type of comparisons, number of AST nodes, the control structure of if-loop statements, number and type of library functions used, etc.

The Python AST size is analogous to the Matlab parse tree size which collaborator Ned Gully found to be helpful in visualizing and distinguishing between Matlab solutions, as shown in the next section.

In JSIM, the hardware description language, students can wire together MOSFETs to form subcircuits. Subcircuits can also contain other subcircuits. The internal wiring diagram of each subcircuit can be represented as a graph, where internal nodes and devices become graph nodes. Using Matlab's graph isomorphism function in the Bioinformatics toolbox, I collected a library of all the topologically distinct subcircuits across all correct submitted implementations of the 4-bit adder. Similarly to how the number of calls to various library functions in Python became features for the Python dataset, the number of declarations of each topologically distinct subcircuit in our library is used to represent each student's 4-bit adder submission.

4.3 Visualization

A next step toward discovering the space of correct solutions is visualizing many student solutions together. If the appropriate feature(s) are chosen, simultaneously viewing representations of hundreds or thousands of solutions to the same problem helps humans and machines extract patterns. The choice of features and graphical representation is initialized by exploratory data analysis but strongly influenced by later results of machine learning algorithms and other mathematical modeling techniques.

I have already taken this approach to multiple labs in 6.004. I have also observed this approach in a collaborator's work on an online Matlab programming challenge called Cody. Plotting dynamic behavior or static features such as parse tree size is enough, in these initial and relatively simple examples, to separate students' solutions into clear clusters representing

different solutions.

4.3.1 Cody Solution Maps

Solutions submitted to Cody are immediately evaluated and scored. Once the code is validated with a suite of input-output constraints, it can be displayed on a solution map. Two examples of solution maps are shown in Figures **??** and **??**.

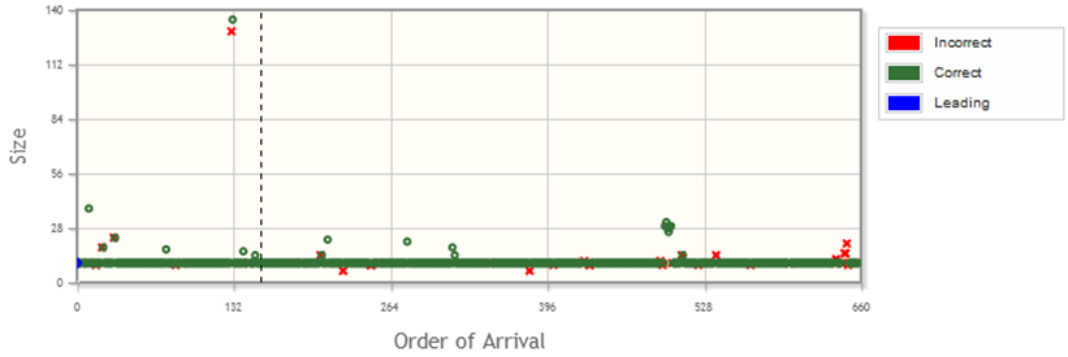
The solution map plots solutions as points against two axes: order of arrival (on the horizontal axis) and code size (on the vertical axis). Correct answers are green circles. Incorrect answers are red x's. We define *code size* as the number of nodes in the parse tree of the solution. Despite the simplicity of this metric, code size can provide quick and valuable insight when assessing large numbers of solutions. An instructor is likely to be interested in common responses, both correct and incorrect, as well as extreme outliers, and the solution map reveals these and other interesting patterns.

For example, Figure **??** shows a solution map for a problem which has a single obvious solution. Almost all the solutions are correct and exactly the same size. This presents as a great many correct solutions in a single line, packed so close together that they blur into a single rail of green circles.

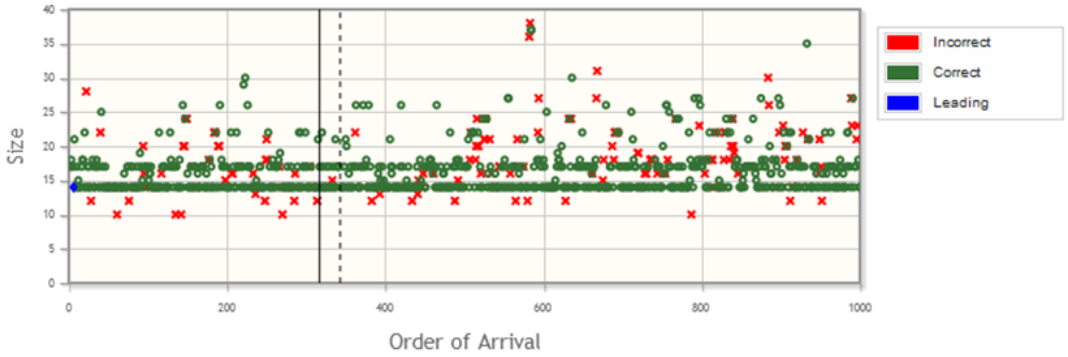
When a problem has two common solutions, we might see two rails. Figure **??** shows the solution map for the Triangular Number problem. The computationally efficient solution for the nth triangular number is $n(n+1)/2$. A less efficient but simple MATLAB solution is to create and then sum the series from 1 to n. These solutions are the two rails evident in this solution map for the Triangular Number problem.

4.3.2 Turing Machine Analysis

The second example comes from 6.004, but is not coded in JSIM. Rather, it is coded in a custom language for describing the behavior of Turing machines. The assignment requires that students write the state-transition rules for a Turing machine that halts on the symbol '1' if the input is a string of matched parentheses and halts on a '0' otherwise. Students design their own symbol library and state names for the finite state machine portion of their Turing machine, and list behavior specifications. Each line in the behavioral specification boils down to the following: if you're in state X and you read



(a) The solution map for a problem which has a single obvious solution.



(b) The solution map for a problem which has two common solutions.

Figure 1: Solution map examples. The *leading* solution is the earliest, smallest correct solution.

symbol Y on the tape, overwrite symbol Y with symbol Z, move the tape-reader in direction W, and transition to state Q.

It is difficult to help students with the Turing machine assignment because many sets of state-transition rules behave identically, given the same input tape of parentheses. Even after looking at hundreds of Turing machines that all give the same correct final answers, there is very little a human can discern simply by looking at the students' code. The same is true even after translating these textual statements into a diagram of state transitions.

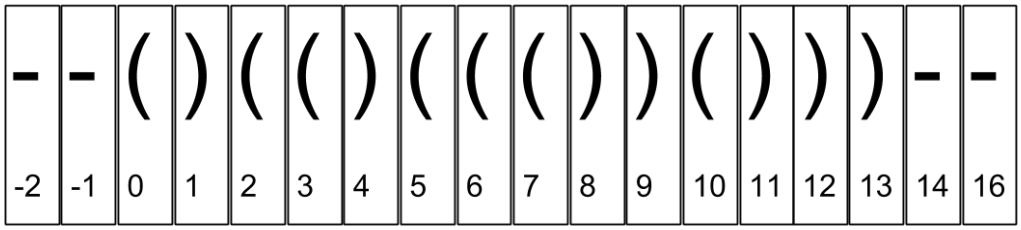
Each staff member is encouraged to complete the lab on their own before counseling students. Staff members were aware of the solution they each found, and yet were not aware that there were two mutually exclusive common solutions. At least one staff member admitted steering students away from solutions they did not recognize, but in retrospect may have indeed been valid solutions.

In order to visualize this dynamic behavior, I ran all the two-state machines 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.

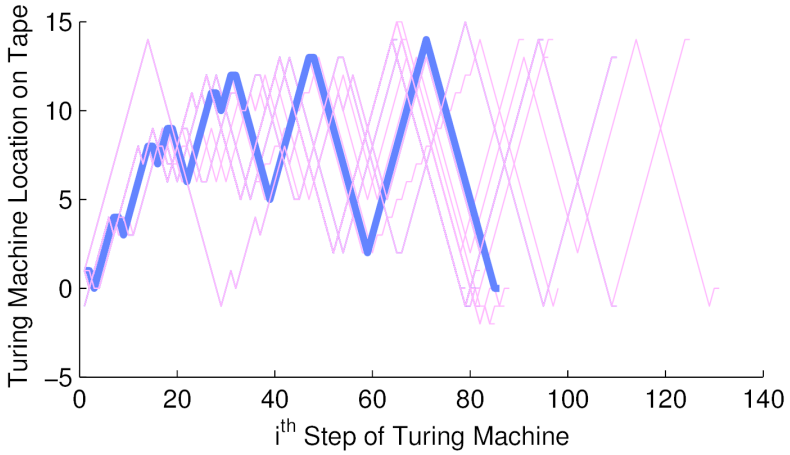
The majority (88%) of the solutions employed one of two mutually exclusive strategies. These strategies were identified by visual inspection of the movement of many Turing machines across a common input tape. Figure **??** shows their locations on the common input tape over time, segregated by strategy 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 and (2) matching the n^{th} open parenthesis with the n^{th} closed parenthesis, as is the case in standard mathematical notation. The remaining 12% of solutions included less common strategies. At least two strategies in this group are known to pass the provided, fixed test suite but are wrong, because they cannot handle an arbitrary depth of nested parentheses.

4.3.3 Analysis of a 4-Bit Adder Construction Lab

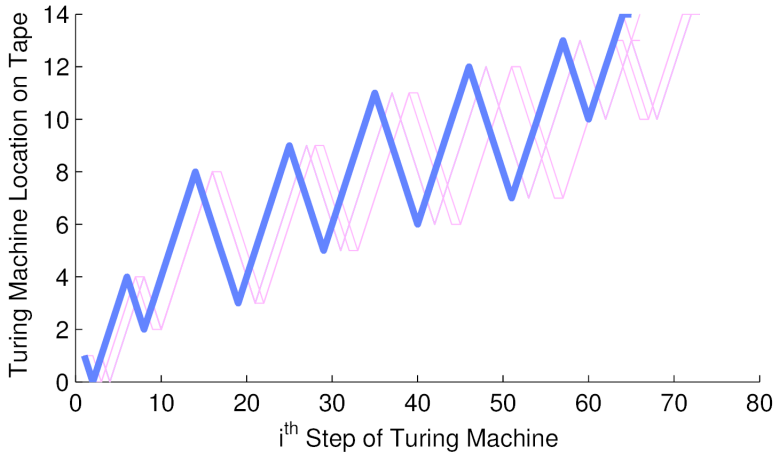
My third example also comes from 6.004. From submission logs of the Spring 2013 semester, I studied students' final correct solutions, which are imple-



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



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



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

Figure 2: 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.

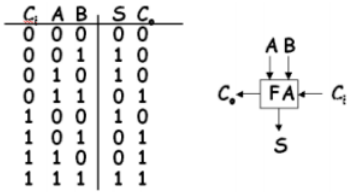
mentations of a CMOS circuit that performs addition on two unsigned 4-bit numbers and produces a 5-bit result. This assignment is coded in JSIM, the hardware description language similar to SPICE. These circuits are created by students declaring internal nodes connected by P- and N-type MOSFETs in an appropriate topology. The students' programming environment includes a simple editor for entering a circuit description and a waveform browser to view the results of a simulation. Nearly two hundred students submitted correct implementations of the 4-bit adder.

It is suggested in the assignment's lab handout that students first build a library of logic gates, such as NAND, NOR, XOR, and XNOR, out of N- and P-FETs. From that library of logic gates, the lab describes how to build the logic gates that compute the sum and carry bits of a 1-bit adder, also called a full adder. The lab pictorially represents how these full adders can be chained together to form one 4-bit ripple carry adder circuit, as shown in Figure. **??**.

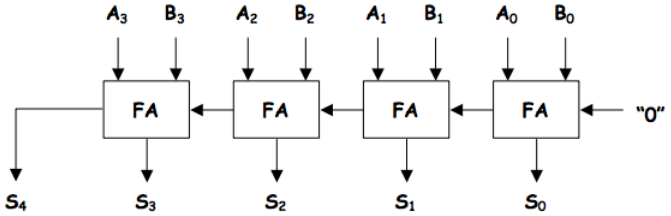
One can see in the histogram of solutions by size, quantified by the number of MOSFETs instantiated in the simulated device, shown in Figure **??**, that there is a major cluster, several smaller clusters, and multiple solution size outliers, large and small. On closer examination of the solutions in each of these peaks, the tallest peak represents two similar but distinct solutions, one of which is a direct transcription of the assignment's description into JSIM code. The smallest solutions represent students who have identified intermediate nodes that can be shared by multiple subcircuits. By identifying and removing redundant nodes, students produced small, difficult-to-read circuits. Students who produced relatively large solutions instantiated unnecessary gates and/or used "positive logic" rather than inverting logic. Since CMOS gates can only implement inverting functions like NAND and NOR, additional inverters are necessary to make ANDs and ORs. To create solutions with positive logic, students composed AND and OR gates, and chained them together, rather than reasoning about how to chain NANDs and NORs together to implement the same function.

4.3.4 Assisting Students to Completion and Encouraging Revision

As of the beginning of the Spring 2013 term, the 6.004 course software now saves complete snapshots of student solutions-in-progress whenever a student saves or runs tests. Over the course of these snapshots, each solution-in-progress evolves into a complete solution employing one of possibly sev-



(a) The truth table for the sum and carry bits of a full adder component.



(b) The chain of full adders that forms the ripple-carry 4-bit adder.

Figure 3: A ripple-carry 4-bit adder.

eral distinct, correct strategies. Informative features of partial solutions would enable supervised machine learning algorithms to successfully predict which strategy a particular solution-in-progress will evolve toward. I hope to demonstrate generality on similar datasets from 6.005, a software engineering course which also has collected snapshot sequences of students' code.

4.4 Mathematical Modeling

Once feature vectors are composed for each solution in a dataset, it is possible to apply machine learning and other mathematical modeling techniques to cluster the data or otherwise extract relationships between solutions and between features. As an initial exploration of feasibility, Rishabh Singh and I applied k-means, an unsupervised clustering method, to feature vectors representing correct 6.00x Python submissions and separately to feature vectors representing 6.004 4-bit adders written in JSIM. We compared the k-means clusterings to each human expert's clusterings, and also compared the human expert-generated clusterings' adjusted mutual information with each other. The clusterings of Python solutions produced by k-means were comparable to human clusterings, with respect to this metric. However, the clusterings of 4-bit adders were not. This has triggered a re-design of the features computed

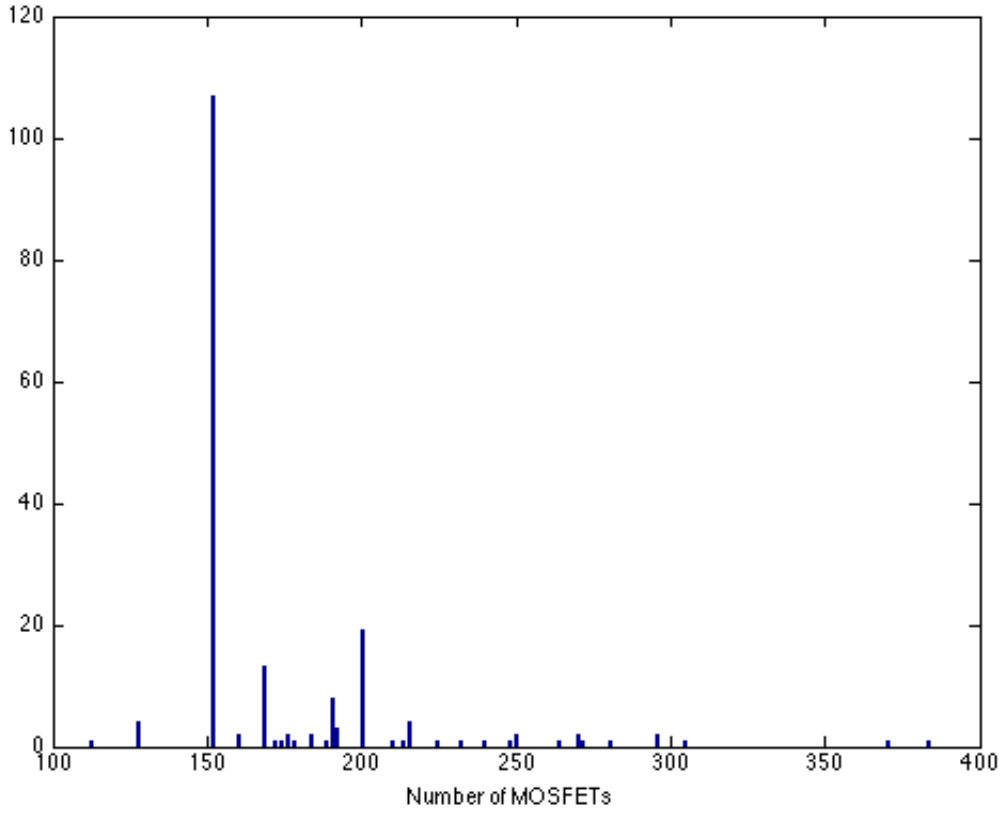


Figure 4: Histogram of ripple-carry 4-bit adders.

for JSIM submissions.

Going forward, I would to like experiment with supervised machine learning methods, including the same supervised, intermittently active learning process that Jenna Wiens successfully used on cardiac data [**?**]. This allowed experts to label a relatively small number of examples, while still generating high classification accuracy. This method is patient-specific, which may be transferrable to the domain of student solutions where we will likely need problem-specific classifiers.

4.5 Human-Computer Interaction

After repeating this process on several domains and tasks within those domains, including 6.004's design assignments written in a hardware description language and 6.00x's software designs written in Python, I propose to generalize the experience and build a tool for teachers which supports this kind of exploratory analysis of student solutions regardless of the teachers' specific domain and assignment.

The eligible domains to which this thesis could potentially apply have several characteristics. Assignments must have a specification that the student's solution must meet. At the same time, it makes most sense to apply the priniciples and methods described in this thesis to domains where students have a broad range of freedom for the internal design of their solution. The exploratory data analysis and feature selection will power machine learning tools, such as clustering algorithms and interactive machine learning modules for teachers to correct the systems' classification of new solutions.

4.6 Generating Data-Driven Assistance

In addition to contributing to teachers' and students' understandings of the solution space, it is also possible to use solution cluster classifiers and solution path recognition to provide semi- and completely automated assistance to students. There are multiple possible projects that would explore the value of this kind of assistance, which is informed by the existence of multiple solutions. While it is unclear which project(s) will ultimately become part of the thesis product, I will enumerate the current possibilities here. Most of these projects would be tight collaborations with other researchers at CSAIL.

The first is the possibility of "force-multiplying" [**?**] teachers' feedback, based on discovered clusters. When teacher(s) observe a new cluster in their

students' submissions, they can write a custom feedback message that is propagated to all current and future students whose solutions fall into that cluster, as determined by the teacher-trained classification algorithms. These messages can critique the approach, address a bug, or encourage a re-design and re-submission for additional points or prestige, similar to the two-stage feedback currently used in 6.005 code submissions.

A future M.Eng. student in my research group is working on helping TAs send custom feedback messages to multiple students, based on a common error in 6.005's code-reviewing software, Caesar. This system could be combined with my cluster-discovery interface for force-multiplying solution-based feedback to students in an existing class.

The lecturer for 6.005 has also built a continuous build system, called Didit, to which students are encouraged to commit often. It provides them with feedback messages, as well as grades. It could be an alternative system in which to integrate solution identification for the purposes of distributing solution-specific feedback to students.

Yet another possible feedback mechanism to students is using knowledge of the solution space to inform peer-pairing, when staff are busy or not present. Hubert Pham, while leaving CSAIL, has rewritten the 6.004 help queue software with the future application of a peer-pairing option in mind. Peer-pairing could be done by human inspection of solutions, or by teacher-trained solution classifiers. The most straightforward strategy is to pair students whom the system predicts are on *similar* paths to a solution.

Solution path recognition can inform more automated forms of assistance as well. By incorporating all known solutions as reference solutions in Singh et al.'s AutoGrader [**?**], the authors of that work believe it may be possible to overcome its current inability to consider which distinct solution a solution is closest to. Alternatively, code differences between a code snapshot in an unfinished solution path and the nearest code snapshot that belongs to a completed path may be a more simple form of automated help when a student has exhausted all other resources.

4.7 Evaluation

4.7.1 Measuring Teachers' Understanding of the Space of Student Solutions

In order to evaluate the contribution of this interactive data visualization environment for teaching staff, I will run a user study on staff members of courses, at MIT and possibly other institutions, in which the student submissions are in machine-readable code and there is non-trivial latitude given to students for the internal solution design. By interacting with representations of many students solutions' at once, users will attempt to identify clusters. If staff independently find similarly meaningful, persistent patterns in their students' solutions using the tool, I will consider it a success. I will also follow up with qualitative interviews with course staff who did and did not use the tool, about their understanding of the space of solutions to a particular assignment they are aware of, after the conclusion of that assignment.

4.7.2 Representing the Solution Space for Learners

Inspired by both the literature on the educational value of comparing and constrasting alternative designs, as well as a personal desire to incorporate higher level discussions of design choices and trade-offs into 6.004, among other classes, I will adapt the teacher-targeted exploratory data analysis and visualization tool into a student-targeted explanatory data visualization tool. This tool will show students where their solution lies in the space of solutions. The role of this visualization will be to serve as a rich conversation piece over which students and staff, and also eventually student pairs and small groups, can discuss design decisions and solutions that are alternatives to their own.

One measure of the success of this tool will be to quantify the accuracy and comprehensiveness of short written statements comparing and constrasting students' own solutions with other solutions. If the accuracy and comprehensiveness of self-reflective statements are higher when a student can interact with the visualization tool than when students are given a portfolio of printed out raw code examples to leaf through, then I can conclude that the visualization provides student-accessible information.

4.7.3 Evaluating Data-Based Interventions

By comparing metrics of students' progression toward a solution when paired based on solution paths, relative to random pairing and no pairing, I can measure the interventions' effect on the completion of the current lab and subsequent related labs. A similar process can be applied to students who receive automated help based on the identification of which of multiple solutions that student appears to be pursuing.

5 EXPECTED CONTRIBUTIONS

I hope to discover the qualities of critical features of coding assignments in the course of this research, develop a process by which teachers can identify specific critical features and solution clusters in their own settings, and then channel that understanding back into an enhanced student experience. More specifically, this process, instantiated as stand-alone software tools, web applications, or plug-ins to existing systems, will help teachers across multiple domains discover the full space of alternatives, recognize novel or better solutions in their students' solutions, and provide feedback to students that is informed by that knowledge. I also hope to extend the software to help students understand alternative design choices and their tradeoffs.

While the tangible deliverables will be specific instances of software, I will report on the generalizability of visualization and classification of the multiple solutions as a tool for improving hints and answers to students' questions, whether they are provided by peers, staff, or automation. I will also report on the character of "critical features" found to be helpful for understanding the space of solutions, the methods by which students were brought to reason about and comprehend design alternatives, and how peers can be effectively paired within a space where individual solutions can vary widely.

6 TIMELINE

- **Fall '13** Deploy visualizations of correct student submissions during check-offs with me in 6.004. Log interactions with the visualization and record conversations with students while they compare their design to other options. Engineer features of JSIM, Python, Java, and/or

Matlab that capture the information that humans appear to use when clustering raw code submissions by hand.

- **Spring '14** Use Bose Fellowship to improve and deploy visualization web applications for LAs and TAs, interview them on their knowledge of the solution space pre- and post-interaction with the visualization. Add machine learning components as necessary, for helping humans identify patterns. Explore use of teacher-written force-multiplied feedback to partial as well as completed labs. Experiment with teacher-based peer-pairing.

- **Summer '14** Complete an internship outside Boston, within the space of ML and HCI.

- **Fall '15** Use Bose Fellowship to improve and deploy visualization web applications for all interested LAs and TAs, continue experimentation with force-multiplication mechanisms in 6.004 or 6.005, and attempt auto-suggestion of peer-pairings for students based on partial solutions. Compare student outcomes based on whether and which assistance type they receive.

- **Spring '15** Write thesis. Tie up loose ends. Defend thesis. Celebrate. Graduate!

7 Acknowledgments

This work is supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374 and the MIT Amar Bose Teaching Fellowship. It has also been made possible by countless discussions and pair-research sessions with my advisor Rob Miller, fellow members of CSAIL HCI, Rishabh Singh, Prof. Leslie Kaelbling, Prof. Chris Terman, Prof. John Guttag, and Marty Glassman.