# Toward Facilitating Assistance to Students Attempting Engineering Design Problems

Elena L. Glassman
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139 USA
elg@mit.edu

Ned Gulley
The MathWorks, Inc.
3 Apple Hill Dr.
Natick, MA 01760 USA
ned.gulley@mathworks.com

Robert C. Miller
MIT CSAIL
32 Vassar St.
Cambridge, MA 02139 USA
rcm@mit.edu

## ABSTRACT

In engineering design courses, many problems have a specification that the student's implementation must meet, but give the student a large range of freedom for the internal design of that implementation. There may be several distinct, correct strategies for solving them, some of which may be unknown to the teaching staff or intelligent tutor designer. When a student is pursuing an unrecognized strategy and begins to struggle, staff may redirect them, costing unnecessary work, and automated hint generators may offer unhelpful feedback. We have taken a first step toward discovering these alternate correct strategies by visualizing many student solutions together, using dynamic and static features of these solutions, so that the teaching staff can understand the space of correct strategies. This approach has been applied to two domains: an online Matlab programming challenge and an undergraduate computer architecture course. We discuss these initial investigations and pose discussion questions to the community about potential enhancement and application of this analysis.

## Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*

## Keywords

Problem Solving Process, Digital Design, Computer Science Education, Pattern Recognition

## 1. INTRODUCTION

In engineering design courses like computer architecture and programming, problems often have a specification that the student's implementation must meet, but give the student a large range of freedom for the internal design of that implementation. These problems may have several different but still correct strategies for solving them, some of which may even be unknown to the teaching staff of the course. This raises problems for helping students and giving feedback. In face-to-face situations, if a teaching assistant doesn't recognize the student's strategy, then they may redirect the student completely, costing them work. In an intelligent tutor or massively open online course (MOOC), the automated hint generators may not recognize the unexpected strategies, and will generate unhelpful hints.

A first step toward discovering these alternate correct strategies is visualizing many student solutions together, so that the teaching staff can understand the space of correct strategies. We have taken this approach in two domains: an online Matlab programming challenge and an undergraduate computer architecture course. We have found that plotting execution behavior or static features such as parse tree size is enough, in these initial examples, to separate students' solutions into clear clusters representing different strategies. In the Matlab challenge, visualizing code size provides insight into common strategies as well as successful and unsuccessful outliers. We see these visualizations being mined by the users themselves to improve their coding skills. In the computer architecture course, this led to better education of the teaching staff, and as a result, they now ask a simple question to identify the student's approach before trying to help them.

In this discussion paper, we present these two motivating examples, consider some of their implications, and propose a number of questions for discussion at the conference, summarized here and expanded on later in this paper:

- What features are useful for visualizing or automatically clustering engineering design solutions? For programming domains, for example, features could include measures of program complexity, stack depth, and/or runtime characteristics. For digital logic and analog circuit domains, features could include graph metrics and voltage traces on intermediate nodes.
- How can teaching staff be trained to quickly recognize the strategy of a given student, in order to give tailored feedback?
- If teaching staff are in short supply (as in a MOOC), how can peers help each other in a space where there are multiple good solution strategies? For example, how can an algorithm running on top of a discussion forum automatically pair students who need help with others who successfully used the same strategy? Conversely, how can a discussion forum-based algorithm broaden students' understanding by exposing them to students with different strategies?

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ICER'13,* August 12–14, 2013, San Diego, California, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2243-0/13/08
http://dx.doi.org/10.1145/2493394.2493400 ...$15.00.

- If peer help is not feasible, then how can we provide automated help based on strategy recognition in a design space where multiple correct strategies are possible?

## 2. RELATED WORK

Courses deployed on MOOC platforms, such as edX and Coursera, include introductory programming, an introduction to electrical circuits, and human-computer interface classes. These classes each require some level of design.

Visualization and automatic recognition of the multiple ways by which a student can approach a solution is an area of active research. Kiesmueller et al. [4] 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. [3] 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.

Tackling problems with multiple solutions directly, Taherkhani et al. [6] demonstrated the practicality of differentiating between multiple solutions, i.e., different sorting algorithms, in students' solutions to a particular engineering design problem using a supervised machine learning method.

Weld et al. [7] speculate about crowd-powered personalization in the context of online education. Specifically, they mention student competency measures as a way to inform peer-pairing and resource recommendations.

Singh et al. [5] are pushing the state of the art of automated feedback for introductory programming assignments, like those assigned in 6.00x. However, their software is currently only differentiating between solutions based on their input-output characteristics, not the strategy used. 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 identify which functionally equivalent variant of a correct solution a student is approaching.

Our work is relevant to complex design tasks with multiple correct solutions, so it may be of particular interest to teaching staff with a constructivist perspective. Such staff members may be attempting to elicit each student's envisioned solution strategy, and then help them toward a working solution that employs that strategy [1]. The algorithms, tools, visualizations, and resulting insights from this and future work are intended to augment this constructivist approach to teaching.

## 3. VISUALIZING SUBMISSIONS TO AN ONLINE PROGRAMMING GAME

The MathWorks' Cody[1] is an informal learning environment that does not have associated teaching staff or the overarching structure of a course. However, it can still support recognition of multiple correct solutions. Cody is an online programming game based on the MATLAB programming language, which poses algorithmic design problems on the web. The public is welcome to submit solutions in MATLAB code. Given the internal range of freedom for implementation, these fit our definition of engineering design problems.

[1] `mathworks.com/matlabcentral/cody`

### 3.1 Cody Background

Cody, developed for the MATLAB language, was originally conceived as a replacement for an older MATLAB online programming competition. This preceding competition consisted of one difficult programming challenge offered to the community every six months. This contest was popular, but players consistently asked for a competition that could be played more frequently. We met this request by offering a large number of smaller problems that could be solved at any time. And, crucially, we allowed the community to add their own problems so the problem pool would not stagnate.

The primary motivation for Cody (and its predecessor) is to make it fun to learn to code in MATLAB. Much of the learning comes from examining solutions provided by other people to the problem that you just solved. There's a marvelous teachable moment there where people say, "That never would have occurred to me, but I can see how it's useful..." Cody's solution map facilitates this process.

### 3.2 Data Collection

Cody originally launched in January of 2012 and has grown steadily since that date. There are currently more than 13,000 registered players. Cody launched with 96 problems that were created by the Cody team. Since that time, members of the Cody community have added another 894 problems. To these 1000 or so problems, players have submitted some 218,000 solutions.

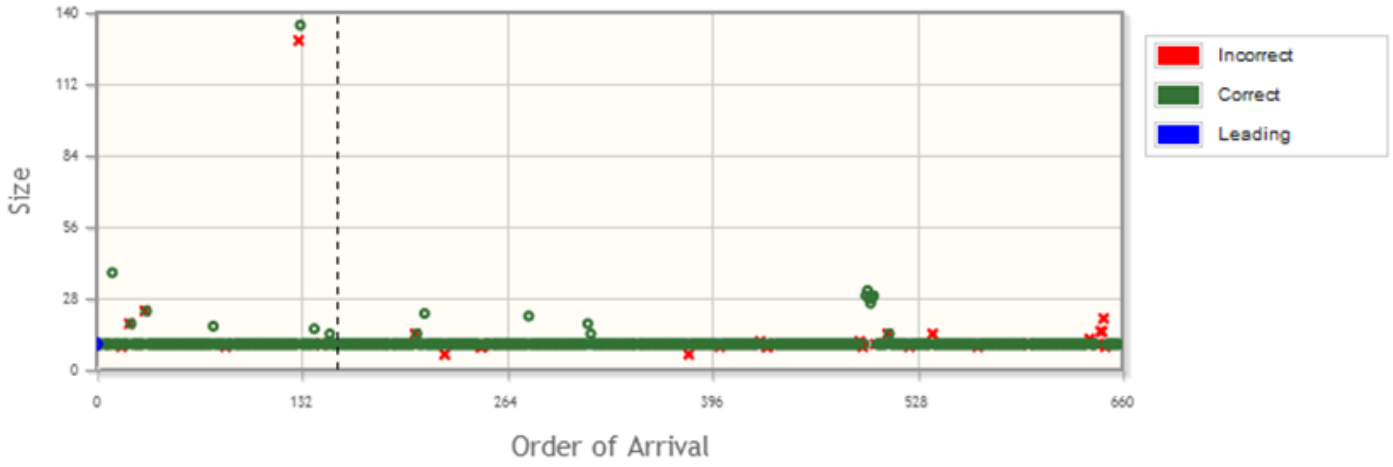
### 3.3 Solution Maps

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 1a and 1b.

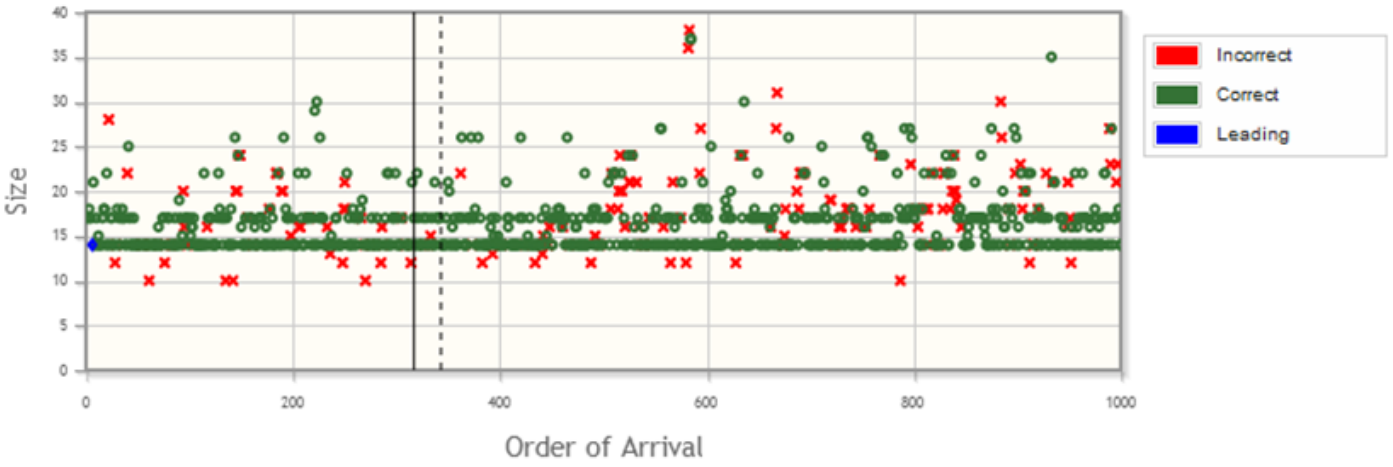
The solution map plots solutions as points against two axes: order of arrival (on the horizontal axis) and code size (on the vertical axis). Correct answers are green circles. Incorrect answers are red x's. We define *code size* as the number of nodes in the parse tree of the solution. Despite the simplicity 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 1a 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 1b 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.




(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.

# 4. PRELIMINARY INVESTIGATION OF AN UNDERGRADUATE LAB ASSIGNMENT

Our second example comes from an undergraduate introduction to computer architecture taught in a computer science department at a large engineering school. Roughly two hundred students, mostly sophomores, enroll per semester.

In this course, students complete labs that span a variety of computer architecture-related design challenges. Students begin by designing CMOS gates at the transistor level and eventually implement a simulated RISC processor at the digital gate level. As an optional design project, students can optimize their processor with respect to a metric combining circuit size and speed. To complete the design component of each lab, the student first runs a staff-provided suite of tests on their solution. If the student's solution passes all the tests, it is uploaded to a server associated with the course, for safekeeping.

## 4.1 The Turing Machine Lab

One mid-semester lab 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. Each student designs their own symbol library and state names for the finite state machine portion of their Turing machine, and lists behavior specifications. Each line in the behavioral specification boils down to the following: 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.

The Turing machine lab is difficult to help students with because many sets of state-transition rules behave identically, given the same input tape of parentheses. Even after looking at hundreds of Turing machines, which 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.

## 4.2 Data Collection

While conducting in-person check-off interviews for the Turing machine lab, we noticed that the dynamic behavior of the Turing machines had some recurring patterns. To investigate further, we visualized this dynamic behavior for all students' two-state Turing machines submitted during the Spring 2011 semester. Of the 194 processed from a single semester, we restricted ourselves to the 148 two-state Turing machines to eliminate confounding factors.

## 4.3 Visually Representing Dynamic Behavior for Strategy Identification

In order to visualize this dynamic behavior, we 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 2 shows their locations on the common input tape over time, segregated by strategy into Figures 2b and 2c. 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.

## 4.4 Observed Benefits for Teacher-Student Interactions

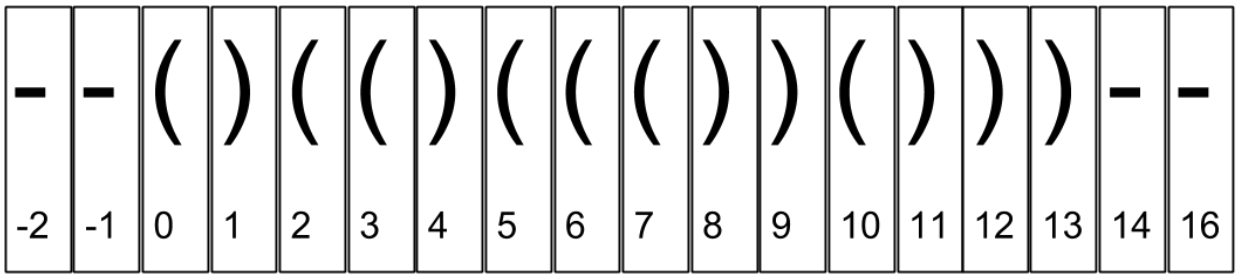
The benefits of this additional knowledge have already been felt. We continue to guide students through this lab, and now ask struggling students a simple question first, to determine their solution strategy. Are they matching the innermost open paren with the innermost closed paren or matching the $n^{th}$ open paren with the $n^{th}$ closed paren? This typically starts off a strategy-level discussion, which helps the author suggest edits that preserve the student's already chosen, valid strategy. Common strategy identification also allows the author to quickly recognize notable, novel alternative solutions, which have occasionally appeared in the course of talking to hundreds of students.

# 5. DISCUSSION

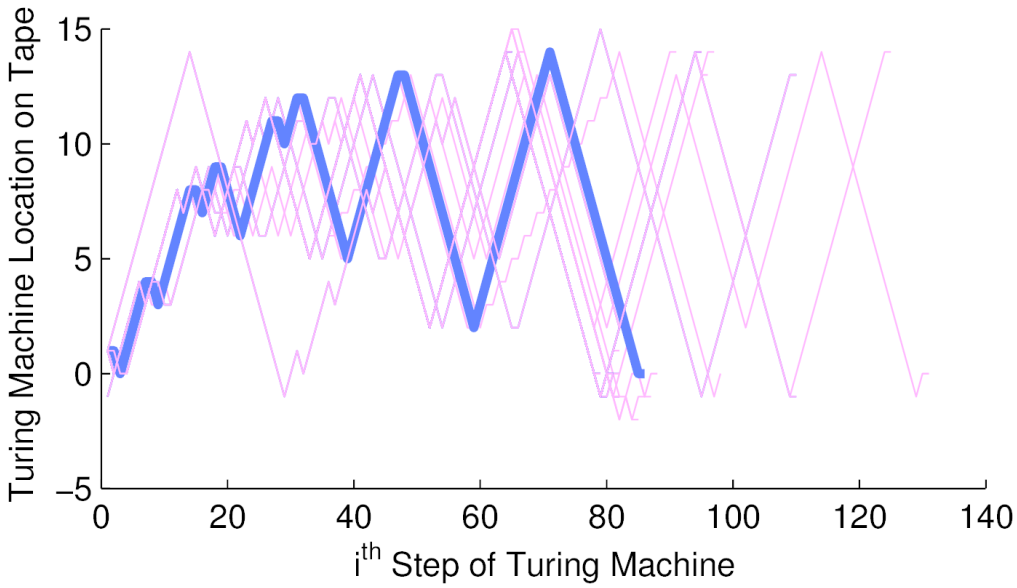
The questions that drive our investigations have developed concurrently with the presented examples. In this section we will revisit these questions and discuss potential future work for answering them.

*What features are useful for visualizing engineering design solutions?* It is not yet clear what features will prove most generalizable across domains for differentiating between strategies. Measures of program complexity, stack depth, and runtime characteristics, along with features of graphs representing component connectivity may each be necessary in some engineering design domains. It is therefore possible that teaching staff will need to be supported by software that facilitates interactive data visualization. By plotting many solutions on coordinate axes described by different subsets of these features, teachers may be able to find the feature subsets by which solutions cluster by strategy.

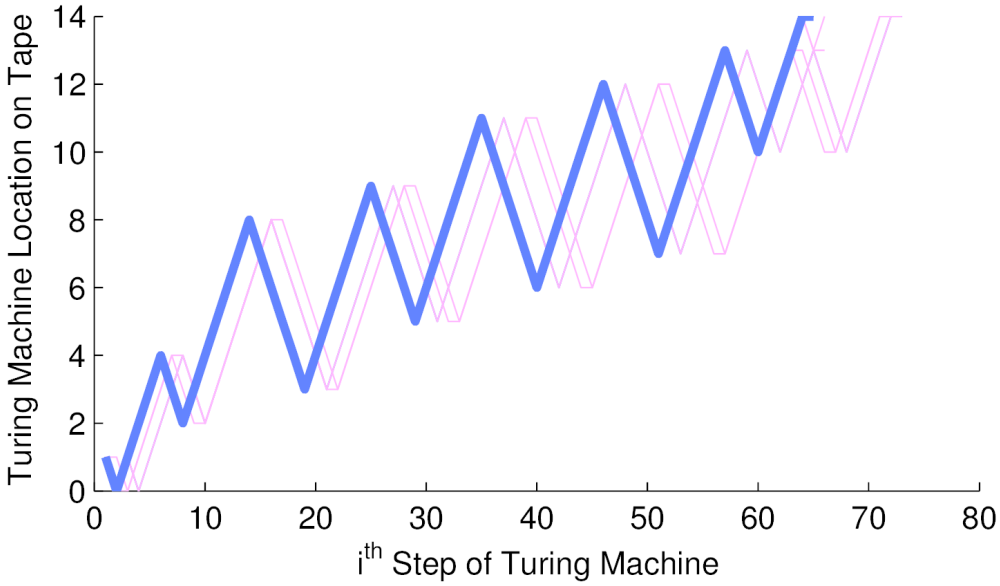
Some of these features may need to be designed specifically for distinguishing between partial solutions. As of the Spring 2013 term, the course software now saves complete snapshots of student solutions-in-progress whenever a student saves or runs tests. Over the course of these snapshots, each solution-in-progress evolves into a complete solution employing one of possibly several 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.




(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 2b and 2c show tape head position over time on the tape illustrated in Fig. 2a. The bold trajectories represent particularly clean examples.

It is an open question as to how much a student's strategy can be inferred from a partial solution. However, if common strategies have already been identified from analysis of previous students' solutions, a teacher could hand-write a multiple choice question or two inquiring at a high level about the strategy of the student.

*If teaching staff are in short supply (as in a MOOC), how can peers help each other in a space where there are multiple good solution strategies?* For students on campus waiting in a long queue for help from a lab assistant or as one of potentially tens or hundreds of thousands of other online students, it may be helpful to pair a struggling student with (1) a fellow student facing the same challenge or (2) a student who has already finished the assignment. Peer-to-peer collaboration or tutoring can be a powerful way to scale up to massive virtual classrooms without proportionally increasing staff. Students who participate in peer collaboration and tutoring on the teaching side may benefit as much or more than those whose expertise is not yet as well-developed.

An open question in this context is how to best pair students based on identified strategies. Students who have mastered one strategy may be most beneficial to struggling students pursuing the same. Consider a stellar student who has sufficiently mastered the material and already implemented or helped implement one or more solutions using the same strategy. This stellar student may acquire an enhanced understanding of the trade-offs of different strategies if he or she is directed to help a struggling student pursue an alternative strategy.

*If peer help is not feasible, then how can we provide automated help based on strategy recognition in a design space where multiple correct strategies are possible?* Consider a situation in which a student's successive partial solutions appear to be converging on a particular cluster of solutions which all use a particular strategy. If the student's partial solutions veer away from that cluster, the student could be trailblazing and creating a solution that reflects a novel strategy. If students ask for a hint, perhaps one form of automated assistance would be to reset them to a snapshot from a point just prior to their departure from the established path toward one of several correct solutions. If students cannot generate alternatives to the strategy they tried and failed to implement, additional assistance might come in the form of suggested additions, based on similar complete solutions and solutions-in-progress associated with an established correct strategy.

In a recent semester of this same undergraduate computer architecture course, a student discovered and published on the course forum a processor optimization which was unknown to the teaching staff and adopted by many fellow students. The staff helped students implement this trailblazer's design. These rare students establish new solution strategies within the course. We hope that the interventions developed in our future work allow rather than dissuade students from creative, novel directions. Improvements to algorithms taught at the undergraduate level are still being published. For example, a new sorting algorithm, the Library Sort [2], was published in 2006. Students will continue to uncover new strategies to solve classic problems, and the challenge to teachers and supporting software is responding appropriately to unique solution strategies that may originate with students.

# 6. ACKNOWLEDGMENTS

Leslie Kaelbling and Chris Terman, Professors of EECS at MIT, and Martin Glassman, Principal Systems Engineer at BAE Systems, provided valuable comments, ideas, and assistance. This material is based, in part, upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1122374.

# 7. REFERENCES

[1] M. Ben-Ari. Constructivism in computer science education. *SIGCSE Bull.*, 30(1):257–261, Mar. 1998.

[2] M. A. Bender, M. Farach-Colton, and M. A. Mosteiro. Insertion sort is o(n log n). *Theory Comput. Syst.*, 39(3):391–397, 2006.

[3] J. Helminen, P. Ihantola, V. Karavirta, and L. Malmi. How do students solve parsons programming problems? an analysis of interaction traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research*, ICER '12, pages 119–126, New York, NY, USA, 2012. ACM.

[4] U. Kiesmueller, S. Sossalla, T. Brinda, and K. Riedhammer. Online identification of learner problem solving strategies using pattern recognition methods. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 274–278, New York, NY, USA, 2010. ACM.

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

[6] A. Taherkhani, A. Korhonen, and L. Malmi. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, pages 83–92. ACM, 2012.

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