# Interacting with Massive Numbers of Student Solutions

**Elena Leah Glassman**
MIT CSAIL
elg@mit.edu

## ABSTRACT

When teaching programming or hardware design, it is pedagogically valuable for students to generate examples of functions, circuits, or system designs. Teachers can be overwhelmed by these types of student submissions when running large residential or recently released massive online courses. The underlying distribution of student solutions submitted in response to a particular assignment may be complex, but the newly available volume of student solutions represents a denser sampling of that distribution. Working with large datasets of students' solutions, I am building systems with user interfaces that allow teachers to explore the variety of their students' correct and incorrect solutions. Forum posts, grading rubrics, and automatic graders can be based on student solution data, and turn massive engineering and computer science classrooms into useful insight and feedback for teachers. In the development process, I hope to describe essential design principles for such systems.

### Author Keywords

data mining; programming exercises; MOOCs

### ACM Classification Keywords

H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

## INTRODUCTION

When teaching programming or hardware design, it is pedagogically valuable for students to generate examples of functions, circuits, or system designs. However, when running large residential or massive online courses, teachers can be overwhelmed by these types of student submissions. Summarizing, exploring, and assessing these types of solutions to assigned problems, even those that can be run through a battery of test cases, involve unsolved challenges.

This work focuses on engineering course assignments that have a behavioral specification students must meet, and allow for a broad range of internal designs. There may be several distinct, correct solutions, some of which may be unanticipated by teachers.

The underlying distribution of student solutions to a particular assignment may be complex, but the newly available volume of student solutions represents a denser sampling of the distribution. The increasing scale of the classroom creates a research opportunity. For example, if we attempt to classify solutions with a Support Vector Machine (SVM), the volume of labeled training data (solutions labeled by teachers) is key to its performance. Such classifiers could be incorporated into the user interfaces I am designing for teachers to explore students' solutions, and could even be trained further based on teachers' interactions with the interfaces.

I am guided by the following questions:

1. How do we help teachers understand the space of solutions generated by students? What features of solutions, and user interface designs, are useful for visualizing and clustering alternative solutions?

2. How do we help teachers understand whether students absorbed a particular lesson or principle relevant to the internal design of their solutions? How do we help teachers respond to common problems, improve a specific assignment, or refine a grading rubric?

3. As a side-effect of discovering common variations of solutions, as well as bugs and misunderstandings, how can peer-to-peer teaching and assistance be enhanced by this knowledge?

**Thesis Statement** A system that empowers teachers to explore the variety of their students' correct and incorrect solutions will enable data-driven refinements to teaching materials. Forum posts, grading rubrics, and automatic graders can be based on student solution data, and turn massive engineering and computer science classrooms into useful insight and feedback for teachers.

## RELATED WORK

There is a growing body of work on both the front end and back end required to manage and present the large volumes of solutions gathered from MOOCs, intelligent tutors, online learning platforms, and large residential classes. The back end necessary to analyze solutions expressed as code has followed from prior work in fields such as program analysis, compilers, and machine learning. A common goal of this prior work is to help teachers monitor the state of their class, or provide solution-specific feedback to many students. However, there has not been much work on developing interactive user interfaces that enable an instructor to navigate the large space of student solutions.

*UIST'14 Adjunct*, October 5–8, 2014, Honolulu, HI, USA.
ACM 978-1-4503-3068-8/14/10.
http://dx.doi.org/10.1145/2658779.2661167

Huang et al. [7] worked with short Matlab/Octave functions submitted online by students enrolled in a machine learning MOOC. The authors generate an abstract syntax tree (AST) for each solution to a programming problem, and calculate the tree edit distance between all pairs of ASTs, using the dynamic programming edit distance algorithm presented by Shasha et al. [13]. Based on these computed edit distances, clusters of syntactically similar solutions are formed. The algorithm is *quadratic* in both the number of solutions and the size of the ASTs. Using a *computing cluster*, the Shasha algorithm was applied to just over a million solutions.

Codewebs [11] created an index of "code phrases" for over a million submissions from the same MOOC and semi-automatically identified equivalence classes across these phrases, using a data-driven, probabilistic approach. The Codewebs search engine accepts queries in the form of subtrees, subforests, and contexts that are subgraphs of an AST. A teacher labels a set of AST subtrees considered semantically meaningful, and then queries the search engine to extract all equivalent subtrees from the dataset.

Both Codewebs [11] and Huang et al. [7] use unit test results and AST edit distance to identify clusters of submissions that could potentially receive the same feedback from a teacher. These are non-interactive systems that require hand-labeling in the case of Codewebs, or a computing cluster in the case of Huang et al.

Several user interfaces have been designed for providing grades or feedback to students at scale, and for browsing large collections in general, not just student solutions. Basu et al. [1] provide a novel user interface for *powergrading* short-answer questions. Powergrading means assigning grades or writing feedback to many similar answers at once. The back end uses machine learning that is trained to cluster answers, and the front end allows teachers to read, grade or provide feedback to those groups of similar answers simultaneously. Teachers can also discover common misunderstandings. The value of the interface was verified in a study of 25 teachers looking at their visual interface with clustered answers. When compared against a baseline interface, the teachers assigned grades to students substantially faster, gave more feedback to students, and developed a "high-level view of students' understanding and misconceptions" [3].

At the intersection of information visualization and program analysis is an interactive visualization embedded in the MathWorks' Cody[1], an informal learning environment for the Matlab programming language. The Cody programming challenge does not have any teaching staff associated with it but does have the interactive *solution map* visualization to help participants discover alternative ways to solve the programming problem, after they submit at least one function that passes all test cases. A solution's parse tree size is the arbitrary metric by which solutions are ranked. Some participants try to race others to the solution with the smallest parse tree, through both their own ingenuity and the mining of alternative code snippets from other solutions revealed in the solution map. The solution map plots each solution as a point against two axes: time of submission on the horizontal axis, and parse tree size on the vertical axis. Despite the simplicity of this metric, solution maps can provide quick and valuable insight when assessing large numbers of solutions [4].

This work has also been inspired by information visualization projects like WordSeer [9, 10] and CrowdScape [12]. WordSeer helps literary analysts navigate and explore texts, using query words and phrases [8]. CrowdScape gives users an overview of crowdworkers' performance on tasks. An overview of crowdworkers each performing on a task, and an overview of submitted code, each executing a test case, are not so different, from an information presentation point of view.

## VISUALIZING, CLUSTERING, AND EXPLORING CODE

I led the development of OverCode, an interactive visualization for the many code solutions submitted to a programming exercise in a massive online course [5]. Without tool support, a teacher may not read more than 50-100 solutions before growing frustrated with the tedium of the task. In the MOOC datasets we tested the tool on, there were at least a thousand solutions per programming problem. Given a relatively small sample size of the spectrum of solutions, teachers cannot be expected to develop a thorough understanding of the variety of strategies used to solve the programming problem, or produce instructive feedback that is relevant to a large proportion of learners. They are also less likely to discover unexpected, interesting solutions.

With OverCode, teachers can explore the variation in hundreds or thousands of programming solutions generated by students attempting a set of Python programming exercises in a large university course or MOOC. Understanding the wide variation in students' solutions is important for providing appropriate, tailored feedback, refining evaluation rubrics, and exposing corner cases in automatic grading tests.

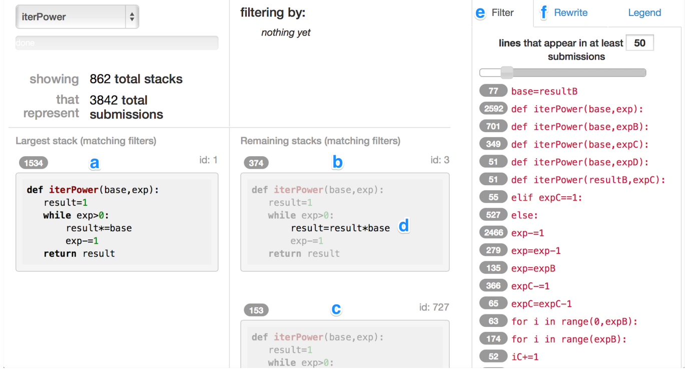
### Implementation

OverCode's novel back end cleans up student solutions for easier visualization by renaming variables. The back end tracks each solution's local variables during execution on the same test case. During renaming, variables that behave 'the same' across different solutions are automatically given the same name, as a function of the students' original naming choices.

With lightweight static analysis after renaming variables, the back end creates clusters of functionally identical solutions. The cleaned solutions are readable, executable, and describe every solution in their cluster. The algorithm's running time is *linear* in both the number of solutions and the size of each solution. In contrast to CodeWebs [11] and Huang et al. [7], OverCode's pipeline does not require hand-labeling and runs in *minutes on a laptop*, then presents the results in an interactive user interface.

OverCode's front end presents the cleaned solutions as each cluster's unique descriptor, which otherwise would be difficult to automatically generate. In Fig. 1, (a), (b), and (c) are

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


iterPower
done
filtering by:
nothing yet
showing 862 total stacks
that represent 3842 total submissions
Largest stack (matching filters)
Remaining stacks (matching filters)
1534
a
id: 1
374
b
id: 3
d
153
c
id: 727
e Filter
f Rewrite
Legend
lines that appear in at least 50 submissions

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

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

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

```
77   base=resultB
2592 def iterPower(base,exp):
701  def iterPower(base,expB):
349  def iterPower(base,expC):
51   def iterPower(base,expD):
51   def iterPower(resultB,expC):
55   elif expC==1:
527  else:
2466 exp-=1
279  exp=exp-1
135  exp=expB
366  expC-=1
65   expC=expC-1
63   for i in range(0,expB):
174  for i in range(expB):
52   iC+=1
```

Figure 1. OverCode user interface.

cleaned solutions with renamed variables representing clusters of 1534, 374, and 153 solutions, respectively. The differences between clusters (d) are highlighted. Given those initial clusters, OverCode lets teachers further filter (e) and merge (f) clusters based on rewrite rules.

### Value for Teachers

The value of OverCode for teachers comes in the form of confidence and improved feedback to students. Compared to a non-clustering baseline, OverCode allowed teachers to more quickly develop a high-level view of students' understanding and misconceptions, and plan course forum posts with feedback that is relevant to more students. We are currently extending OverCode's pipeline to include other languages, such as Java and hardware description languages, and more complex coding assignments. I hope OverCode will help teaching staff continue to gain a deeper insight into their students' design choices.

### Value for Students

While in the role of the 'teacher' working with OverCode, I learned new Python syntax. With modification to the interface, students may also learn from interacting directly with OverCode. This model of learning echoes the interactive solution maps in Cody, the MathWorks' programming game, which has no teaching staff. In future work, I can investigate what changes to OverCode are necessary to create a visualization of fellow students' solutions that is beneficial for students' learning, instead of teachers' understanding. This may also address the question of how students can help each other explore alternative designs.

## HELPING STUDENTS GUIDE EACH OTHER

Teaching peers, also called peer learning, is pedagogically valuable [2]. I am currently pilot testing various systems for matching students, keeping in mind that different students may be working on different, but equally valid, correct solutions to the same problem. The following prototypes motivate further systematic data collection, on which more theoretical models could be based.

### Evaluating Alternative Solutions

In MIT's Computation Structures course, students create digital circuits in a hardware description language. Through exploration of hundreds of previous student solutions to an early, basic lab assignment, I found that the space of alternative correct circuit designs is nearly completely separable by the number of device primitives, i.e., transistors, in each design. Picking from previous student designs, I was able to automatically present current students with design alternatives that were better or worse than their own. Students were asked to give advice to a future student about how to improve the poorer of the two designs. Their explanations gave a rich window into their understanding. Students in the Spring '14 offering of the course gave strikingly cogent advice to future students. A question I plan to further explore is: how best do we close this loop, so that students benefit from the design alternatives and advice generated by classmates?

### Pairing Students Based on Their Solutions

In another assignment in the same course, students are asked to create a Turing machine that determines whether a string of parentheses is balanced, i.e., has a closing parenthesis for every open one. I visualized the dynamic behavior of over a hundred students' Turing machines, and found that there were two distinct common designs. Several fellow teachers were only aware of one. At least one teacher admitted steering students away from designs that, in retrospect, may have been the 'other' common solution they did not know about. In addition to better preparing teachers, can we automatically recognize which design a student is working on? When they need help, should we pair them with another student who is working on, or has already finished, the same design? How do we support, or at least not interfere with, students working toward novel designs? I hope to address these questions by integrating a program analysis-based user interface like OverCode with systems for social hint-giving and receiving between students.

### Debugging Advice Based on Test Cases

In the same course, students ultimately build entire simulated processors composed of logic gate primitives. These solutions, expressed as pages' worth of an in-house declarative programming language, can become so complex as to be very challenging to debug even with the one-on-one help of a seasoned teaching staff member. Students who have previously resolved a bug can be in a better position than a staff member to help a fellow student with the same bug, if the staff member has never encountered that bug before.

"Dear Beta" is a website I built so that students can post explanations of their own resolved bugs, indexed by the failed test cases the bug caused. Providing the explanation is pedagogically useful, and students struggling with a bug can reference it for advice. When students sought help and found one of their fellow students' hints helpful, they had the option of upvoting it. Both website usage statistics and anecdotal evidence suggest that students find the website to be helpful. What are the necessary factors to consider when generalizing this peer-helping framework to additional software design courses?

## USING SYSTEM INTERACTIONS TO TRAIN MACHINE LEARNING ALGORITHMS

In a domain as complex as student solutions to engineering and programming assignments, traditional machine learning may fail. Teachers each have their own internal metrics for what is and is not important when sorting through student solutions. For example, when a small sample of introductory programming course teaching assistants were consulted about how to cluster students' solutions to simple programming assignments, their clusterings often disagreed with each other on how to group and explain student variation.

From the prototypes that culminated in OverCode, two things are clear. First, program analysis can do rigorously what was not possible with unsupervised clustering techniques running on student solution feature vectors [6]. Second, program analysis and manually specified rewrite rules can only get teachers so far. Teachers' ability to interrogate the thousands of student solutions available to them in our datasets was limited by their patience to specify what they believed to be irrelevant differences between clusters of solutions.

This is an application ideally suited to interactive machine learning (IML) techniques. By logging teachers' interactions with the data, IML techniques could suggest or predict additional helpful feature equivalences and rewrite rules, rather than requiring teachers to specify each one by hand. These decisions can become the subject of staff and classroom discussions, and be incorporated into teaching materials and automated grading rubrics.

IML techniques may also 'learn' from student-to-student interaction logs gathered from the social systems described in the previous section. If students are each working on debugging their own distinct solutions, and a group of students all mark a debugging hint, provided by another student, as helpful, then those solutions are related by the relevance of that particular hint. IML techniques may be able to suggest merging particular clusters of solutions in OverCode, based on shared hint relevance.

## SUMMARY

At the conclusion of my graduate work, I hope to have built systems that help both teachers and students in large-scale engineering and programming courses. In the development process, I hope to describe essential design principles for such systems, and show that teachers using the systems gain deeper insight into their students' thoughts and designs, allowing richer conversations with students about their design choices. To better support learners when teachers are vastly outnumbered by students, or when teachers are simply not present, I hope that these systems help students discuss their solutions and guide each other.

## ACKNOWLEDGMENTS

This material is based, in part, upon work supported by the National Science Foundation Graduate Research Fellowship (grant 1122374), the Microsoft Research Fellowship, the Bose Foundation Fellowship, and by Quanta Computer as part of the Qmulus Project. Any opinions, findings, conclusions, or recommendations in this paper are the authors', and do not necessarily reflect the views of the sponsors.

## REFERENCES

1. Basu, S., Jacobs, C., and Vanderwende, L. Powergrading: a clustering approach to amplify human effort for short answer grading. *TACL 1* (2013), 391–402.
2. Boud, D., Cohen, R., and Sampson, J. *Peer learning in higher education: Learning from and with each other*. Routledge, 2014.
3. Brooks, M., Basu, S., Jacobs, C., and Vanderwende, L. Divide and correct: using clusters to grade short answers at scale. In *Learning at Scale* (2014), 89–98.
4. Glassman, E. L., Gulley, N., and Miller, R. C. Toward facilitating assistance to students attempting engineering design problems. In *Proceedings of the Tenth Annual International Conference on International Computing Education Research*, ICER '13, ACM (New York, NY, USA, 2013).
5. Glassman, E. L., Scott, J., Singh, R., Guo, P. J., and Miller, R. C. Overcode: Visualizing variation in student solutions to programming problems at scale (in submission). *ACM Trans. Comput.-Hum. Interact.* (2014).
6. Glassman, E. L., Singh, R., Gulley, N., and Miller, R. C. Feature engineering for clustering student solutions. CHI 2014 Learning Innovations at Scale Workshop, 2014.
7. Huang, J., Piech, C., Nguyen, A., and Guibas, L. J. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED Workshops* (2013).
8. Muralidharan, A., and Hearst, M. Wordseer: Exploring language use in literary text. *Fifth Workshop on Human-Computer Interaction and Information Retrieval* (2011).
9. Muralidharan, A., and Hearst, M. A. Supporting exploratory text analysis in literature study. *Literary and linguistic computing 28*, 2 (2013), 283–295.
10. Muralidharan, A. S., Hearst, M. A., and Fan, C. Wordseer: a knowledge synthesis environment for textual data. In *CIKM* (2013), 2533–2536.
11. Nguyen, A., Piech, C., Huang, J., and Guibas, L. J. Codewebs: scalable homework search for massive open online programming courses. In *WWW* (2014), 491–502.
12. Rzeszotarski, J. M., and Kittur, A. Crowdscape: interactively visualizing user behavior and output. In *UIST* (2012), 55–62.
13. Shasha, D., Wang, J.-L., Zhang, K., and Shih, F. Y. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man and Cybernetics 24*, 4 (1994), 668–678.