Interacting with Massive Numbers of Student Solutions

Elena Leah Glassman MIT CSAIL, elg@mit.edu

ABSTRACT

Massive online courses, such as MOOCs, enable instructors to reach thousands of students at once. However, if teachers go beyond multiple choice answers and assign problems that require students to generate answers in the form of a natural or programming language, it becomes difficult to view all of the solutions or get a satisfactory summary. Grading the material submitted by these students, even automatically-verifiable material such as computer programming assignments, involves several unsolved challenges. Specifically, many engineering and computer programming assignments have a specification that must be met, but give the student a broad range of freedom for the internal design of their solution. There may be several distinct, correct solutions, some of which may be unanticipated by teachers, making it potentially difficult for staff to help students reach their own correct solutions. It also complicates the process of getting help from peers. My approach is to visualize hundreds or thousands of student solutions to discover alternatives, and classify the solutions through interactive machine learning. The visualization and classifications can inform assistance given to students, in the form of staff-student discussions, peer-pairing, and/or automated help.

Author Keywords

Information visualization; knowledge discovery; interactive machine learning; program analysis

ACM Classification Keywords

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

INTRODUCTION

Massive online courses, such as MOOCs, enable instructors to reach thousands of students at once. Some of the earliest MOOCs were programming and engineering classes, like Professor Ng's Machine Learning programming-based course and edX's Circuits and Electronics (6.002x) engineering course. Computer science and engineering education matters. In light of both the rise of MOOCs and the growth of computer science [13], computer science and engineering education is also getting larger in scale.

I am building novel user interfaces to improve engineering and programming education when students vastly outnumber their teachers, as they are, for example, in MOOCs. In order to better support teachers, and their students, I combine techniques from HCI with techniques from machine learning, program analysis, natural language processing, and information visualization. I am guided by the following questions:

1. How do we help teachers understand the space of programming solutions generated by students? Specifically, how can we help teachers understand whether the students are learning the right things; how to improve their teaching materials or respond to common problems; how to improve a specific assignment or the grading scheme; etc.?
2. What features of solutions are useful for visualizing and clustering alternative solutions?
3. What user interface design choices specifically help teachers visualize and cluster alternative solutions?
4. How can students help each other (a) debug or (b) explore alternative designs?

Thesis Statement Clustering and visualizing student solutions at scale will improve the support available to students, provided by staff, peers, and automation. That support can be in the form of

- tools providing teachers with a new way to explore their students' solutions,
- websites that allow students to guide each other to better solutions, or
- data-driven refinements to automated feedback mechanisms, like auto-graders.

RELATED WORK

There is a growing body of work on both the frontend and backend required to manage and present the large volumes of solutions gathered from MOOCs, intelligent tutors, online learning platforms, and large residential classes. The backend 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.

Huang et al. [6] 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 problem, and calculate the tree edit distance between all pairs of ASTs, using the dynamic programming edit distance algorithm presented by Shasha et al. [12].

Paste the appropriate copyright statement here. ACM now supports three different copyright statements:
• ACM copyright: ACM holds the copyright on the work. This is the historical approach.
• License: The author(s) retain copyright, but ACM receives an exclusive publication license.
• Open Access: The author(s) wish to pay for the work to be open access. The additional fee must be paid to ACM.
This text field is large enough to hold the appropriate release statement assuming it is single spaced.

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 [10] 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 [10] and Huang et al. [6] 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 backend uses machine learning that is trained to cluster answers, and the frontend 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" [2].

At the intersection of information visualization and program analysis is Cody[1], an informal learning environment for the Matlab programming language. Cody does not have a teaching staff but does have a *solution map* visualization to help students discover alternative ways to solve a problem. A solution map plots each solution as a point against two axes: time of submission on the horizontal axis, and code size on the vertical axis, where *code size* is the number of nodes in the parse tree of the solution. Despite the simplicity of this metric, solution maps can provide quick and valuable insight when assessing large numbers of solutions [3].

This work has also been inspired by information visualization projects like WordSeer [8, 9] and CrowdScape [11]. WordSeer helps literary analysts navigate and explore texts, using query words and phrases [7]. CrowdScape gives users an overview of crowdworkers' performance on tasks. An overview of crowd-workers 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 from a large-scale programming exercise [4]. Without tool support, a teacher may not read more than 50-100 solutions before growing frustrated with the tedium of the task. 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 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 backend cleans up student solutions for easier visualization by renaming variables. The backend tracks each solutions' 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 backend creates clusters of functionally identical solutions. The cleaned solutions are readable, executable, and describe every solution in its cluster. The algorithm's running time is *linear* in both the number of solutions and the size of each solution. In contrast to CodeWebs [10] and Huang et al. [6], OverCode's pipeline does not require hand-labeling and runs in *minutes on a laptop*, then presents the results in an interactive user interface.

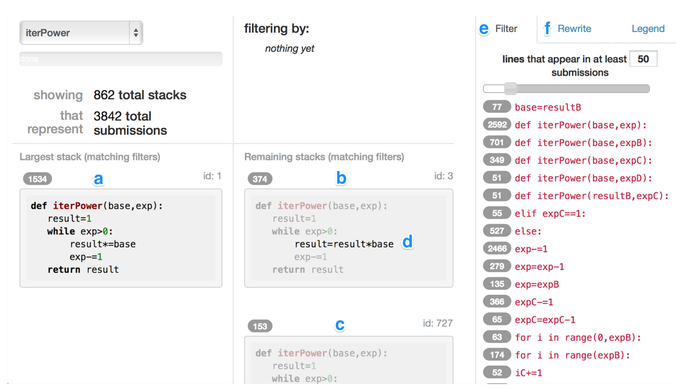

Figure 1. OverCode user interface.

OverCode's frontend 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`

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 get 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 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

I also ask how can students help each other debug, within this space of many potential solutions. I am currently pilot testing various ways for students to help their peers, while also benefiting themselves from the process of synthesizing explanations. Social system engineering refers to the design of these social systems, especially their affordances and incentives that affect students' behavior and, potentially, their learning outcomes. The following are prototypes and preliminary experiments that are motivating systematic data collection, from which I hope to generate more theoretical models.

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, 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, including a student's unprompted class forum thank you note, suggest that students find the website to be very 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

On a domain as complex as student solutions to engineering and programming assignments, traditional machine learning is simply inappropriate. Teachers each have their own internal metrics for what is and is not important when sorting through student solutions. For example, when I consulted a small sample of introductory programming course teaching assistants 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 [5]. Second, program analysis and manually specifying 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 different stacks 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, rather than requiring teachers to specify each one by hand. These decisions can become the subject of staff and classroom discussions, teaching materials for students, and incorporated into automated grading rubrics.

Similarly, IML techniques may be able to make bigger leaps by mining student-to-student interactions in the social systems described in the previous section. If students are each working on debugging their own distinct processor designs, and a group of students all mark a debugging hint, provided by another student, as helpful, then those designs are related by the relevance of that particular hint. IML techniques may be able to suggest combinations of the same features used in OverCode to predict which designs are similar, 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 get 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 guide each other and discuss their solutions.

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. 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.
3. 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).
4. 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).
5. 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.
6. 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).
7. Muralidharan, A., and Hearst, M. Wordseer: Exploring language use in literary text. *Fifth Workshop on Human-Computer Interaction and Information Retrieval* (2011).
8. Muralidharan, A., and Hearst, M. A. Supporting exploratory text analysis in literature study. *Literary and linguistic computing 28*, 2 (2013), 283–295.
9. Muralidharan, A. S., Hearst, M. A., and Fan, C. Wordseer: a knowledge synthesis environment for textual data. In *CIKM* (2013), 2533–2536.
10. 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.
11. Rzeszotarski, J. M., and Kittur, A. Crowdscape: interactively visualizing user behavior and output. In *UIST* (2012), 55–62.
12. 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.
13. Zweben, S. Computing degree and enrollment trends. Tech. rep., Computing Research Association, Washington, DC, 2013.