Chapter 1

Conclusion

This thesis demonstrates several ways to capture distributions of student solutions and make this information useful to teachers and students in large programming ~~classes~~courses. Capturing commonalities can save teachers time and exhaustion by summarizing many solutions with a single representative. It can also direct teachers to where the majority of the class is within the space of solutions or errors, so they can respond to student needs based on data, not hunches. Capturing variation allows teachers to see how students deviate from the common choices in ways that are superior and deserving celebration or inferior and requiring corrective feedback. These deviations can be used to inspire or directly seed exercises based on modern theories of how students learn generalizeable knowledge from concrete examples. It also saves teachers the time and effort of generating pedagogically valuable examples themselves. Systems in this thesis also demonstrate how the inherent variation in student solutions can be automatically presented back to students, as an active reflection exercise that can benefit both current and future students.

The technical challenges tackled in order to capture and display distributions of solutions include designing solution representations, measures of similarity between solutions, mechanisms for grouping or filtering them, and human-interpretable displays. Challenges in workflow design centered around simple, minimally-intrusive ways to perform targeted learnersourcing in an on-going programming ~~class~~course.

1.1 Summary of Contributions

The original OverCode system included a novel visualization that shows similarity and variation among thousands of Python solutions, backed by an algorithm that uses the behavior of variables to help cluster Python solutions and generate human-interpretable ~~canonicalized code~~ platonic solutions for each cluster of solutions. The Foobaz system demonstrated how, even when an aspect of variation is hidden in one view of a distribution of solutions, it can be exposed, in context, in a separate view. Additional follow-on work expanded ~~OverCode's canonicalization~~ the OverCode normalization process to incorrect solutions and explored additional human-interpretable mechanisms for identifying population-level design alternatives.

Foobaz contributed a workflow for generating personalized active learning exercises, emulating how a teacher might socratically discuss good and bad choices with a student while they review the student's solution together. The Foobaz system demonstrates one concrete way in which a conversation about design decisions, i.e., variable naming, that might only happen in one-on-one interactions with an instructor, can be scaled up to an arbitrarily large class, personalized to each student, be based on the distribution of naming choices found in solutions composed by the teacher's current class or previous classes, and deployed to future classes who assign the same programming ~~exercise~~problem.

Finally, ~~ClassOverflow, which collectively refers to the Dear Beta and Dear Gamma workflows , contributes the concept~~ the self-reflection and comparison workflows demonstrate the potential of targeted learnersourcing. The ~~Dear Beta~~ self-reflection workflow allows students to generate hints for each other by reflecting on an obstacle they themselves have recently overcome while debugging their own solution. The ~~Dear Gamma~~ comparison workflow prompts students to generate design hints for each other by comparing their own solutions to alternative designs submitted by other students. ~~Results from these studies~~ Studies of the Dear Beta and Dear Gamma systems show that personalized hints collected through these workflows can be viably learnersourced, and that these hints serve as helpful guides to fellow students encountering the same obstacle or attempting to reach the same

goal.

1.2 Future Work

""“One never notices what has been done; one can only see what remains to be done..."”” - Marie Curie (1984)

There are many directions in which this work can be grown. Some of the research that has been published in parallel with this work can be directly incorporated into future systems. At the same time, many of the existing features can be improved without incorporating any new techniques, based on already collected feedback during user studies, deployments, and other design critiques.

1.2.1 OverCode

OverCode was designed for thousands of correct solutions to introductory Python programming problems. Many residential and online exercises now afford automatic, immediate feedback about the correctness of a solution. Under these conditions, nearly all the final solutions are correct, but it is clear from reading through them that some students have better command of programming than others, just based on compositional quality. As a result, OverCode was designed to reveal to the teacher what their students' solutions actually look like, modulo white space and comments. These solutions are rendered for the teacher using most common variable names and statement orders.

However, as programming ~~exercises~~ problems escalate from the simple to the complex, OverCode's utility breaks down. It can still create ~~canonicalized cluster-representing~~ platonic solutions and display them in a way that helps teachers spot contrasts, but the clusters themselves become smaller and smaller. As programming ~~exercises~~ problems require more complex solutions, there is also an increasing need to go beyond representing what students

actually wrote and toward human-interpretable representations of larger clusters. There are two promising complementary avenues for implementing this next step.

The first avenue is more bottom-up clustering. ~~Just as the variation in variable names was hidden by recognizing that variables across programs were semantically equivalentand could be replaced with their most popular student-given name, it may be possible to do the same with respect to semantically-equivalent subexpressions. There two promising ways to determine the semantic equivalence of subexpressions or lines of code in large corpuses of student solutions: (1) semantic equivalence based on the rules of the programming language (2) probablistic semantic equivalence based on the dynamic~~ A critical part of forming OverCode stacks is identifying which variables are semantically equivalent, based on their behavior during execution on ~~test cases, as defined in Nguyen et al.~~the same test cases. Overcode then renames variables to their most common name across all solutions. This normalization process could be extended to subexpressions and code blocks as well, using semantics-preserving transformations, e.g., compiler optimizations, and probablistic semantic equalivalence (PSE) [**?**]. OverCode could~~feature an additional button which collapses these detected equivalences, showing the most popular choice in the solution representing the new, larger clusters~~, in the future, replace semantically equivalent or PSE subexpressions with their most common form across all solutions. Alternatively, this process could be controlled by the user, who decides the degree to which they would like to hide less common semantically equivalent subexpressions. An interface ~~, perhaps like the teacher interface in Foobaz , can~~ like Foobaz could allow teachers to view and compose feedback ~~that dimension of variation, if they wish~~on the distribution of semantically equivalent subexpressions, just as Foobaz allowed teachers to do for variable names.

The second promising avenue for creating larger clusters is continuing to model the corpus of solutions to the same programming ~~exercise~~ problem as samples from an underlying distribution of solutions shaped by ~~the teacher 's~~ teacher explanations, student ~~'s~~ prior knowledge, common novice misconceptions, and the constraints of the language itself. Depending on the model chosen, larger clusters could be groups of whole solutions or groups of sub-components the co-occur within many solutions. Iterating on solution representation,

model choice, interface, and interaction will hopefully allow teachers to understand their students' design choices, give style feedback, and assign grades with confidence.

Even simple additions to the OverCode interface could approximate what more sophisticated statistical methods would do. For example, LDA may pull out distributions of commonly co-occuring variable behaviors ~~(as described in the chapter on OverCode extensions),~~ but the current OverCode interface could be modified to suggest and filter by common sets of strictly co-occurring variable behaviors. This approximation is less robust in the face of noise, but it is at least easier for the ~~user~~ teacher to interpret using the language of filtering.

When visualizing the resulting clusters, it is important to keep in mind the ease with which ~~users~~ teachers can interpret the results. Simple modifications to the OverCode interface can go a long way, such as highlighting the sub-expressions or ~~canonicalized~~ normalized variable names within lines of code, rather than the entire lines, that separate one cluster from another. Another modification would increase the interpretability of the ~~canonicalized~~ normalized variables. Since the variable behavior is used to ~~canonicalize~~ normalize variable names, the values of variables and sub-expressions during execution on a test case could be displayed just beneath or alongside the code, as demonstrated in ~~an experimental Python notebook [?]~~Kevin Kwok's (unpublished) Python notebook demo, in addition to the way it is currently shown, as a legend mapping each variable name to its behavior beneath a tab that many user study subjects did not consult.

When the program tracing, renaming, or reformatting scripts generate an error while processing a solution, the solution is removed from the pipeline. This percentage of correct edX solutions that did not make it through the pipeline was less than five percent of solutions were excluded from each problem, but that can be reduced further by adding support for handling more special cases and language constructs to these library functions. As solutions get more complex, it will also be necessary to expand the OverCode pipeline to more gracefully handle user-created objects and helper functions. Complementary efforts, like [**?**], also cannot yet handle more than a single function. This remains a difficult but important hurdle to expanding beyond introductory Python programming courses. As part of this

expansion, it may be helpful to adopt the `observe` construct introduced by Gulwani et al. [**?**] and support teacher annotation of solutions with the observe construct within the OverCode user interface. By soliciting human annotation of important variable values, two variables would not need to behave identically in order to be considered by OverCode as semantically the same.

OverCode could also be integrated with the autograder that tests student solutions for input-output correctness. The execution could be performed once in such a way that it serves both systems, since both OverCode and many autograders require actually executing the code on specified test cases. If integrated into the autograder, teachers could give ~~'power feedback '~~ feedback by writing line- or stack-specific feedback to be sent to students alongside the input-output test results. The OverCode interface may also provide benefit to students, in addition to teaching staff, after students have solved the problem on their own. Cody is a standing example of the value of this kind of design/style feedback. However, the current interface may need to be adjusted for use by non-expert students, instead of teachers.

Simple interface enhancements would give teachers, and potentially students, more freedom to explore. For example, many user study subjects requested the ability to promote any cluster to be the reference cluster. A student might want to see their solution as the reference, at least as a starting point. Similar to ~~GroverCode's~~ the GroverCode sorting mechanism, the OverCode interface could support sorting clusters by their similarity to the reference as well as by cluster size.

1.2.2 GroverCode

~~GroverCode's canonicalization~~ The normalization of incorrect solutions is based on heuristics about co-occurrences of syntax and variable behavior in both correct and incorrect solutions. This was an attempt to see how far the OverCode features could go toward ~~canonicalizing~~ normalizing incorrect solutions in addition to correct solutions, without using any additional technology. ~~However, there is pre-existing technology, i.e.,~~

AutoGrader [**?**] ~~, that could~~ could also be added to the pipeline.

The AutoGrader understands a language for expressing corrections in introductory Python programs. The authors created a list of such possible corrections~~, and if applying~~ . If a small subset of those corrections changes a solution's input-output behavior to match a reference solution, ~~the Autograder~~ Autograder has diagnosed the bug(s) and can automatically correct the solution. It does not take into account which sets of corrections are more common, unlike the design philosophy the defines OverCode. However, the corrections that are applied are guaranteed to convert the incorrect solutions into correct ones. The correct solutions can move through the OverCode pipeline to be ~~canonicalized~~ normalized and clustered without heuristics.

If the AutoGrader is added to the pipeline, it will likely move some but not all solutions from the incorrect to the correct category. The corrections that have been automatically applied could be displayed alongside the corrected code in the GroverCode interface, for transparency. The GroverCode process for analyzing and displaying solutions that are still incorrect after being analyzed by the AutoGrader would be unchanged. If the language for expressing corrections is simple and expressive enough that it can be used by graders to write new rules while they are grading, the combined AutoGrader-OverCode pipeline could be rerun to correct any other yet ungraded solutions that are not already automatically corrected.

~~There are also a~~ A variety of interface updates ~~that have also~~ have been requested by the teachers who used GroverCode during live deployments~~, such as~~ . For example, some teachers requested the ability to dynamically add new tests to the test suite ~~. These requests have been catalogued and added the the project as feature requests.~~ and to edit student solutions in the interface without switching to an external editor.

1.2.3 Foobaz

The approach taken in designing Foobaz may be generalizable to other aspects of programming style. Just as OverCode establishes the equivalency of variables based on their behavior during test cases, one could establish the behavior equivalency of larger or more abstract entities, such as student-written lines of code, sets of lines of code, or entire functions. We consider this a class of problems that Foobaz, and similar systems built after it, can tackle. Future iterations of Foobaz-inspired interfaces could also include additional constraints and affordances to encourage teachers to leave more explanations for their assessments, accompanied by better support for reusing common comments. Teachers could also be given the option to augment or overwrite existing labels, e.g., "too short," to match their own preferences.

1.2.4 Targeted Learnersourcing

~~The primarily difficulty of iterating on the two workflows,~~ Dear Beta and Dear Gamma ~~, that are collectively referred to in this thesis as ClassOverflow, is that they are~~ were most successful when ~~they are~~ directly integrated into ~~teacher' s own systems and endorsed or~~ teachers' existing mandatory assignments or actively promoted by staff. ~~Sometimes the support is there, and sometimes it is not. For classes with sufficiently supportive teachersin the future, a 'prompting'~~ In future classes with supportive teachers, a module can be added to the Dear Beta workflow ~~which~~ that identifyies when a student is in a good place to supply a hint and prompts them to do so. This would be an improvement over the current stand-alone website that students are explicitly directed to when they need help, but not when they can give help. Future work on Dear Gamma should include improving the metric for optimality and the rendering of solutions such that all the differences between more and less optimal solutions are visible to reflecting, hint-writing students. One potential future incarnation of the Dear Beta ~~+~~ and Dear Gamma workflow is a crowd-sourced intelligent tutor that helps students get to a correct solution and then optimize that solution, completely driven by student-written hints.