Chapter 1

Discussion

The systems in this thesis give teachers more awareness about the content generated by students in large programming classes and enable styles of teaching that are usually only feasible in smaller classes, such as discussions of variation and style that are directly driven by what the students have already written. These systems also scale up automated compare-contrast, self-explanation, and formative assessment-style exercises whose content is generated by students and curated by teachers. The current state of the art in theories of how humans learn predict that these supported interactions between teachers and students will enhance learning.

1.1 Design Decisions and Choices

This thesis work began with the vision that a teacher would someday be able to look at a display summarizing hundreds or thousands of student solutions to the same problem and immediately see—and comment on—good and bad design decisions that students made. The number of possible distinct student solutions grows rapidly with the number of design decisions and design choices students can make.

The solution space could be imagined as an n-dimensional space, where each dimension corresponds to a design decision, e.g., whether to solve a problem iteratively or recursively. The design choices students make could be thought of as discrete points along each dimension in that solution space. While the number of distinct combinations of design choices students choose can be large, the number of dimensions in this space, ~~i.e., the~~ which represent underlying design decisions, grows more slowly.

However, the structure of this hypothetical solution space ignores how design choices affect each other. Some design decisions are mutually exclusive, e.g., looping over a particular array with `for` or `while`, some decisions are correlated with one another, and some decisions are completely independent. A tree-like description of the solution space can capture some of these relationships between decisions. If a node represents a design choice, e.g., to loop over an array, there may be two or more choices, e.g., a `for` or a `while` loop, that can be represented as child nodes. The choice to use a `for` loop poses an additional design design, e.g., whether to use `range` or `xrange`: `for i in [x]range(input)`. The average depth of the tree would correspond to the average number of design decisions the students faced and the average branching factor would correspond to the average number of distinct choices students in the corpus make at each decision point.

The curse of dimensionality predicts that, as we add more and more dimensions to the solution space, the density of solutions will decrease and the likelihood that any two solutions occupy the same location in that space will go down. The regularity of code discussed in Related Work should help ameliorate the curse of dimensionality, but only partly. In other domains, it is often necessary to collect more data as the dimensionality of the space increases. However, given that the solutions are generated by students, the number of correct solutions are more likely to go down rather than up as the complexity of solutions, and the associated dimensionality of the solution space, increases.

This vision of the solution space inspired and is shaped by the concrete systems built and described in this thesis. In each system, the dimensions of solution variety have been tamed and orderd by choosing what features to ignore and what features to index by. For example, OverCode ignores white space, comments, variable names, and statement orders. It indexes by normalized lines. In order to assign a variable name quiz, Foobaz looks at the behavior of the variables in the student solution, ignoring everything else about it. Dear Beta and GroverCode forget what values cause a solution to fail a test case, preserving only that the test case has failed. Dear Gamma ignores circuit topology and indexes by the number of transistors.

1.2 Capturing the Underlying Distribution

There will be some design decisions within each solution that are rare and some that are common, some that exemplify good programming practices and others that do not. These decisions might create inefficient solutions, reveal a student's fundamental misunderstanding, or use a feature of the language in a creative way. The distribution of solutions along these dimensions of variation may reflect student prior knowledge, teacher explanations, and common misconceptions.

Computer scientists are often looking for better ways to find needles in haystacks and computationally-minded social scientists are trying to characterize the haystack [**?**]. One could think of Codex [**?**] as a mechanism for finding needles ~~, i.e.,~~ in haystacks because it helps find particular code fragments ~~, in haystacks, i.e.,~~ in large collections of open source code. One could also think of Webzeitgeist [**?**] as a mechanism for finding needles ~~, i.e.,~~ in haystacks because it helps find web design choices ~~, in haystacks, i.e.,~~ in large collections of web pages. OverCode, Foobaz, Dear Beta, Dear Gamma, and GroverCode are trying to faithfully represent the haystack ~~, i.e., the distribution of student solutions and bugs, while also supporting~~ and support needle-finding.

OverCode characterizes the haystack by identifying and displaying the relative popularity of different correct solutions. It also provides filtering and cluster-merging tools that help teachers find needles, e.g., interesting and ~~unusal~~ unusual solutions. Similarly, GroverCode helps teachers understand the haystack of correct and incorrect solutions by displaying the distribution of solutions as a function of their correctness on test cases.

Foobaz characterizes the haystack of variable naming choices by diplaying the relative popularity of different names in each OverCode stack of solutions and across the entire dataset of solutions. The feature of sorting by popularity allows teachers to see the common names as well as the ~~needles, i.e.,~~ uncommon names that ~~might be~~ are excellent or terrible needles for teachers to find.

The learnersourcing workflows helped students, in addition to teachers, understand the distribution of solutions and bugs. Dear Beta reveals some information about the distribution of failed test cases. The variety of student debugging hints for each failed test case reveals some information about how many different possible problems might cause each failed test case. In Dear Gamma, the optimality of circuit solutions was quantified as the number of transistors. The haystack was characterized by visualizing the distribution of solutions from least to most optimal. This distribution was examined by teachers and given to students after they completed their own solution. The distribution helped teachers identify the most optimal solution, which ~~represented a needle, i.e., a~~ made up a very small proportion of all solutions. In other words, the edge of the solution distribution contained a valuable needle.

1.3 Helping Teachers Generate Solution-Driven Content

The future generations of tools like OverCode, Foobaz, Dear Beta and Dear Gamma might allow teachers to continuously mine the solutions of ~~their~~ thousands of current and previous students, generating days worth of lessons just by extracting themes from both the innovative and problematic design decisions found in the growing pile of student solutions. This would be an opportunity to discuss and reflect on the various trade-offs that exist in each different solution or optimization.

Turning teacher actions and annotations into personalized feedback for students need not be restricted to variable names. For example, when a teacher writes a rewrite rule in OverCode, OverCode could treat that as an implicit annotation that one sytactic form is semantically quivalent and stylistically superior to another. Like ~~Foobaz's~~ variable name quizzes in Foobaz, the teacher ~~'s implicit or explicitcommentary~~ commentary, both implicit and explicit, could be sent to current and future students as a personalized quiz or just a helpful suggestion. Ideally, all the work that a teacher does in an interface to make sense of student solutions could be automatically translated into solution-driven discussion content and rich feedback.

These solution-driven lessons and feedback would be reuseable as long as the problem specifications do not change. This aligns well with the lifecycle of many courses that are eventually offered at scale. First, the material is iteratively developed and debugged in residential, face-to-face teaching environments. When the course is stable, it can be put up online for a larger number of remote students to engage with. Finally, the course can be archived in such a way that any new student can move through the material at their own pace, but there is no guarantee of staff or a cohort of fellow students moving through the material synchronously. Tools like OverCode, Foobaz, Dear Beta, and Dear Gamma accumulate solutions, commentary, and hints ~~in such a way~~ so that, after enough students and teachers have used them for a given problem, new students can benefit from them without synchronous peers or teachers.

1.4 Writing Solutions Well

The focus on introductory Python programming courses is often just correctness. The MIT EECS introductory Python programming course whose staff used GroverCode makes it a policy not to penalize students on how their solution is written. This means that they sometimes have to hand out full marks to a solution that makes them groan.

This policy may exist for several reasons. The first is the clarity of the policy: if it is correct, it gets full credit. Second is the clarity of the evaluation: if it passes the suite of test cases, it is correct. Third is the apparent appropriateness of the objective for novices: it may be too hard for novices to write a solution well in addition to achieving correctness.

However, it can also be very hard to achieve a correct solution if it is not written well. Something as simple as poor variable names, such as giving an array index a name that suggests it holds the value of the array at that index, can cause students to produce incorrect code.

Just as there is no silver bullet for writing prose well, there is no silver bullet for writing solutions well. Solutions, prose, products, buildings etc., are all designed, and each community has its own ways to help students make good design decisions. For example, in *The Sense of Style*, Steven Pinker suggests picking apart examples of good writing to understand what makes them good [**?**]. The software industry uses one-on-one peer review, often called code review, to find bugs and judge design decisions. ~~This may also serve~~ It serves as an informal teaching opportunity for software engineers. Some communities create design studios, where students give and receive constructive criticism from peers. Designed objects are examined individually and also as a group, emulating the conditions for learning from variation espoused by the theories of learning reviewed in the related work.

Written rubrics can serve as a complement to peer review and design studio practices. For example, some software companies compose and maintain ~~prescriptive~~ style guides against which new code is carefully compared. However, these guides may be more prescriptive than descriptive. In other words, they are not necessarily built on data about how software engineers actually design and write their code.

Design studios and peer reviews are not yet a standard practice in programming education. However, peer review practices are now starting to be used in large online courses to make up for small teacher-to-student ratios. ~~Some~~ At least one residential software engineering ~~courses, e.g., MIT's~~ course, 6.005 ~~, also~~ at MIT, has set up infrastructure for peer review. Peer review forces students to engage with some sampling of other student solutions.

The work in this thesis takes this idea farther. Rather than hope that the randomly assigned peer review experiences provides students with a sufficient variety of examples, the work in this thesis attempts to pull out the design dimensions as well as concrete examples along those dimensions and, when possible, ask students to engage with them in a targeted, personalized way.

This may be helpful even at the level of introductory programming. For example, according to variation theory, a student will understand the concept of a loop in a more generalized, robust way if they have seen all the different ways in which their peers have written that loop. The teacher can quickly and easily provide an expert perspective by commenting on the popular and rare choices. Students can be overwhelmed by choices. For example, they might ask themselves, *Should I use* `range` *or* `xrange`*? Does it matter?* With concrete examples, teachers can help students identify what matters and what does not.

1.5 Clustering and Visualizing Solution Variation

The OverCode pipeline ~~performs~~ does unsupervised clustering. There are distinct failure modes when using clustering in the context of teachers reviewing solutions. Two are most relevant in this work. First, the representation of the solution and the measure of similiarity between solutions can ignore, hide, or otherwise fail to capture what the teacher cares about. Second, when (1) the teacher cannot easily decide what a cluster means to them based on its members and (2) the clustering algorithm cannot explicitly communicate why the cluster exists, the teacher may not trust the clustering and may not feel comfortable using it for propagating feedback and grades back to students. This is exacerbated when the teacher discovers a member of a cluster that seems not to belong.

The OverCode clustering pipeline attempts to escape the first clustering failure mode: erasing distinctions that the teacher may care about. For reasons discussed in detail in the OverCode chapter, OverCode is 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 the most common variable names and statement orders. To stay true to what students actually wrote, this process preserves syntactic differences. For example, when iteratively exponentiating `base` to an exponent, there are multiple ways to multiply an accummulating variable `result` by `base` and save the product back into `result`, such as `result *= base` and `result = result * base`. If the teacher just gave a lecture on common forms of syntactic sugar, OverCode will be sensitive to whether or not students use it. OverCode would allow teachers to observe whether students absorbed the lesson on syntactic sugar based on the way they write their subsequent solutions. The fact that even small differences in syntax creates separate clusters within the OverCode pipeline nearly ensures that any syntactic choices a teacher is interested in has been preserved and can be filtered for in the interface.

The second clustering failure mode—producing clusters that the teacher does not trust—is avoided in several complementary ways. First, there is a clear interpretation of what an OverCode cluster can and cannot include, based on its platonic solution. Specifically, all solutions in the cluster have the same set of lines as the platonic solution after normalizing variable names, standardizing formatting, and removing comments. Second, the differences between clusters is made clear by highlighting which lines make the non-reference clusters different from the reference cluster. Third, the OverCode filtering features and rewrite rules help teachers change their view of solutions into one that preserves the differences they are explicitly interested in and ignores those they are not interested in. Note that filtering by semantic choices may only be possible when the work on Bayesian modeling of these solutions becomes more mature.

In general, the interfaces in this thesis cluster complex objects and visualize those clusters so that there is little guesswork about what is included and excluded in a group and what the boundary between groups is. There are no outliers that are grouped with something else without explanation. Rather than losing faith in the clustering process, outliers can be used as the teacher sees fit to spur ~~improvements either to the software infrastructure of the course , e.g., the input-output testing harness, or the examples students are asked~~ course improvements, including autograder updates and new examples for students to engage with.

1.6 Language Agnosticism

It is not surprising that the evolving values of variables would carry significant semantic meaning in code written by students at the introductory level in languages like Python and Matlab, especially if the style of programming is procedural. This thesis confirms that within the context of introductory procedural Python programs. According to Taherkhani et al. [?], variables carry useful semantic meaning in object-oriented and functional programming styles as well. Gulwani et al. [?] and [?] have also authored semantic analyses ~~, i.e., of the~~ of programming languages that include variable behavior. The languages were C++ and Pascal~~programming languages, with a focus on variable behavior.~~ , respectively. This suggests that OverCode and Foobaz could also be applied to other programming languages. For a language to be displayable in OverCode, one would need (1) a logger of variable values inside tested functions, (2) a variable renamer and (3) a formatting standardization script.

1.7 Limitations

The thesis presents a series of case studies about how to present the variety of programming solutions in a human-interpretable way and make use of it in pedagogically valuable, scalable ways. The methods described only work in a particular domain of solutions: executable programs that solve the same programming problem. This excludes natural language, for example. These case studies embody the design principles espoused, but there is no validated unifying recipe by which a corpus of student solutions can be processed and used. Each corpus and set of teacher values were considered together in order to engineer a representation of solutions–both in the pipeline and in the user interface–that would empower teachers and benefit students. There are many specific technical limitations of the approaches described in the previous chapters.