Automated Data-Driven Hint Generation for Learning Programming

Kelly Rivers
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
krivers@cs.cmu.edu

CMU-HCII-17-103
July 13th, 2017

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Committee:
Ken Koedinger, Carnegie Mellon University, Chair
Brad Myers, Carnegie Mellon University
Vincent Aleven, Carnegie Mellon University
Sharon Carver, Carnegie Mellon University
Tiffany Barnes, North Carolina State University

This work is supported in part by the Program in Interdisciplinary Education Research (PIER), a training grant to Carnegie Mellon University funded by the Institute of Education Sciences (R305B090023).

Keywords

data-driven tutoring, programming tutors, canonicalization, path construction, hint representation, hint evaluation, self-improving tutoring system

Abstract

Feedback is an essential component of the learning process, but in fields like computer science, which have rapidly increasing class sizes, it can be difficult to provide feedback to students at scale. Intelligent tutoring systems can provide personalized feedback to students automatically, but they can take large amounts of time and expert knowledge to build, especially when determining how to give students hints. Data-driven approaches can be used to provide personalized next-step hints automatically and at scale, by mining previous students' solutions.

I have created ITAP, the Intelligent Teaching Assistant for Programming, which automatically generates next-step hints for students in basic Python programming assignments. ITAP is composed of three stages: canonicalization, where a student's code is transformed to an abstracted representation; path construction, where the closest correct state is identified and a series of edits to that goal state are generated; and reification, where the edits are transformed back into the student's original context. With these techniques, ITAP can generate next-step hints for 100% of student submissions, and can even chain these hints together to generate a worked example. Initial analysis showed that hints could be used in practice problems in a real classroom environment, but also demonstrated that students' relationships with hints and help-seeking were complex and required deeper investigation.

In my thesis work, I surveyed and interviewed students about their experience with help-seeking and using feedback, and found that students wanted more detail in hints than was initially provided. To determine how hints should be structured, I ran a usability study with programmers at varying levels of knowledge, where I found that more novice students needed much higher levels of content and detail in hints than was traditionally given. I also found that examples were commonly used in the learning process, and could serve an integral role in the feedback provision process. I then ran a randomized control trial experiment to determine the effect of next-step hints on learning and time-on-task in a practice session, and found that having hints available resulted in students spending 13.7% less time during practice while achieving the same learning results as the control group. Finally, I used the data collected during these experiments to measure ITAP's performance over time, and found that generated hints improved as data was added to the system.

My dissertation has contributed to the fields of computer science education, learning science, human-computer interaction, and data-driven tutoring. In computer science education, I have created ITAP, which can serve as a practice resource for future programming students during learning. In the learning sciences, I have replicated the expertise reversal effect by finding that more expert programmers want less detail in hints than novice programmers; this finding is important as it implies that programming teachers may provide novices with less assistance than they need. I have contributed to the literature on human-computer interaction by identifying multiple possible representations of hint messages, and analyzing how users react to and learn from these different formats during program debugging. Finally, I have contributed to the new field of data-driven tutoring by establishing that it is possible to always provide students with next-step hints, even without a starting dataset beyond the instructor's solution, and by demonstrating that those hints can be improved automatically over time.

Acknowledgments

So many people have supported me in this quest for a PhD that I am sure I will end up missing a few. Nevertheless, I’m so grateful to all of you.

Erik, thank you for being my stalwart supporter and best friend these past six years. Without you, my ideas would be only half-baked and my statistics would be abominable. I love you so much, and I’m so proud to be graduating alongside you.

Mom and Dad, thank you for teaching me to love learning and encouraging me all along the way. You, John, and Emily have been such a wonderful and loving family, and I owe so much to you. Stan, Jodi, Sonja, Greg, and of course Durinn: thank you for welcoming me into your family so warmly! I’m so happy to continue growing with all of you.

Ken, thank you for teaching me how to be a researcher. I’ve learned a great deal from you these past six years about how to shape ideas and test them, and how to keep going even when things fall apart. Thank you for everything.

To my committee, Sharon, Tiffany, Brad, and Vincent: thank you for all the advice, feedback, and recommendations. My thesis work was largely shaped by your wise words. Thanks as well for making time for me in your very busy schedules, especially when we had to schedule the proposal twice!

Huge thanks to Aayush Mudgal for spending a summer working on better syntax hint generation, and to Erik Pintar for modifications on the Cloudcoder interface. I hope the experience was beneficial for both of you. Thank you to David Kosbie, Dilsun Kaynar, and Margaret Reid-Miller for letting me run studies in your classes. Without you, I would never have been able to track down so many students! Thanks also to Jaime Spacco and David Hovemeyer for helping me modify and set up my Cloudcoder instance, and to Norman Bier, Raphael Gachuhi, and Martin van Velsen for getting ITAP into OLI. I appreciate your patience with my many mistakes.

And of course, thanks to all of the amazing people who have welcomed me into their communities this past year. Jenny and Samantha, you have been such wonderful officemates. See, we can ALL finish this whole thesis thing! Dave, Vi, Chris, Caitlin, Ryan, Rony, Nesra, Derek, Eliane, Iris, Steven: you guys are a fantastic learning science cohort. Thanks for helping me improve my research, and for opening my eyes to the many varieties of educational research. Sauvik, Anna, Jeff, Kerry, Queenie, and everyone from HCII: thank you for being the most phenomenal department. I think I might finally have some understanding of design thanks to all of you. Finally, thanks to Dan, Robert, and Julia for being the best co-puzzlers, and everyone from the SCS musical group for putting on such silly shows. You have all made this PhD process so much more fun than I ever could have anticipated.

Contents

1. Introduction: How Can We Support Students?

It is not particularly controversial to state that feedback is an essential component of the learning process. After all, without feedback a student can never learn from their mistakes, and every student runs the risk of encountering misconceptions and sources of confusion during their learning process. But there *are* questions about how feedback can be produced, what it should say, and when it should be provided.

Feedback

Feedback can occur in a wide variety of forms, and plays a role in many different aspects of the learning process, from motivation to content knowledge (Hattie & Timperley, 2007). I am primarily interested in the feedback given directly to students to modify thinking or behavior in order to improve learning, otherwise known as formative feedback. Determining what the content of feedback should be, when it should be provided during learning, and how much to provide can be complicated, due to many factors that affect learning (Koedinger & Aleven, 2007); however, many of these questions have already been addressed in studies over the years (Shute, 2008).

The question of *how* to provide feedback in domains with open-ended problem solving is especially difficult, as it is harder to define correctness and incorrectness when the range of possible strategies grows large. I'm particularly interested in determining how to provide feedback in the domain of programming, for a variety of reasons. First, programming is open-ended yet still easy to measure; there can be dozens of different correct solutions to a simple programming problem, and hundreds of paths towards those solutions, yet they can all be structurally represented and compared, as they are all constrained by the syntax of the programming language. Second, the domain of programming provides more insight into the problem-solving process than many other domains, as it is naturally done on computers, which makes it possible to record every step a student takes. And finally, there is great need for feedback during the process of learning how to program. Computer science is rapidly growing as a core component of general education (Computing Research Association, 2017), and computational thinking is viewed as a necessary skill for the modern world (Wing, 2006), yet students in introductory programming classes struggle and drop out at high rates (Watson & Li, 2014). Many students who drop out claim that they can't catch up in the course after falling behind, due to lack of time or inadequate study resources (Petersen et al, 2016); therefore, there is hope for retaining students if we can identify and assist struggling students early.

Since computer science as a field has rapidly growing class sizes (Computing Research Association, 2017), and since classroom feedback is most useful when given immediately (Kulik & Kulik, 1988), I am interested in determining how to provide feedback to students *automatically*. This challenge is addressed by the field of intelligent tutoring systems (ITSs) (Corbett, Koedinger, & Anderson, 1997). ITSs provide students with practice problems that provide feedback automatically as the student works, to personalize and maximize learning, with a goal of providing the same level of instructional assistance as a human tutor (Bloom, 1984). These tutoring systems traditionally provide feedback to students in the form of correctness feedback, error-specific feedback, and next-step hints (VanLehn, 2006).

Correctness feedback simply tells a student whether their current solution is correct or incorrect; this kind of feedback is already available to students via automatic assessment, where test cases are used to measure whether a solution performs as expected (Douce, Livingstone, & Orwell, 2005). Error-specific feedback tells the students what might be wrong with an incorrect piece of their solution; this assistance can be provided by automatic assessment as well, by carefully designing test cases so that each corresponds to a common error and connecting teacher-created feedback messages to each test case. This is actively done in many constraint-based tutors (Mitrovic & Ohlsson, 1999), and was used in the LISP tutor, one of the first intelligent tutoring systems, as well (Corbett & Anderson, 2001). Finally, next-step hints tell students what to do next in order to make progress or fix their work. Personalized hints are much rarer in automated programming instruction (Keuning, Jeuring, & Heeren, 2016), as it is far easier to see that a program is broken than it is to fix said error. Therefore, I focus primarily on the problem of how to provide next-step hints to programming students in this work.

Next-Step Hints

In human tutoring, hinting is commonly used by tutors as a method to help students remember information they already know or make new inferences on how to solve a problem (Hume et al, 1996). In computer-based tutoring, hints are either provided on-demand or are provided by the tutor based on specific events (such as a student getting a step wrong), with on-demand hints sometimes resulting in better learning (Razzaq & Heffernan, 2010). Student use of hints is varied, ranging from correcting errors or comparing their approach to the teacher's (Cummins et al, 2016) to gaming the system by using hints and feedback with no real attempt at learning (Baker et al, 2004).

One way to provide hints is to tell students what they should do next (next-step hints). Traditionally, next-step hints are provided by the tutor authors when the tutor is first built, by tagging each possible state that a student might reach with a specific hint message, or by generating production rules which can be matched to student work. This approach is feasible in domains where the number of possible states is small (like fraction addition), but it is nearly impossible to do in open-ended problem-solving domains, especially when students are working in an unstructured environment, as they do while coding. Even for the simplest problems, it still takes large amounts of expert time to construct an ITS, which makes the systems difficult to generate at scale (Folsom-Kovarik, Schatz, & Nicholson, 2010).

One alternative to expert authoring is to use a *data-driven* approach instead. This approach was first used in the Hint Factory, a hint-generation method used in a logic proof tutoring system (Barnes & Stamper, 2008). Instead of trying to hand-author hints for any possible proof state, this system gathered data on how previous students using a non-hinted version of the system solved problems, then used the successful solution paths of those students to generate hints for newcomers by telling them what other students in a similar state had done in the past. The Hint Factory was able to generate hints for 91% of student requests and 48% of all observed states (Barnes et al, 2008). This approach is a more feasible way to generate hints for programming problems, but it still has limitations. The space of possible programming states is even larger than the space of possible proofs, and the open-ended text-entry of coding provides more natural variation than a drag-and-drop proof construction system might experience. In other words, the chance that a past student will have written the exact same program as a current struggling student is rather low.

In this thesis, I present a new approach towards data-driven tutoring in ITAP, the Intelligent Teaching Assistant for Programming (Rivers & Koedinger, 2015). This system uses student data to identify a personalized goal state for any given submission, with which it can generate next-step hints. These are bottom-out hints that tell the student exactly how to change their code to get closer to a correct solution. I have evaluated this system in several contexts and have learned much about what students want and need in a hint message. I have also found that hints may help students learn more quickly, which could have positive effects on the student experience.

In this document, I will first describe the technical implementation of the ITAP algorithm and demonstrate its capability with technical evaluations, including an evaluation of whether the algorithm can improve itself by collecting student data over time. I will then discuss the experimental evaluations run on the system. First, I describe several classroom evaluations on student use of hints; next, a usability study on the content and structure of hint messages; and ultimately, an experimental evaluation of the effect of hints on learning. Finally, I discuss the implications and contributions of this research, which include the ITAP system itself in addition to theories on how students perceive and use hints and examples, and I describe plans for future work.

2. ITAP: The Intelligent Teaching Assistant for Programming

This chapter focuses on the major components of ITAP, describing how they work theoretically and algorithmically. This process depends on the concept of a *solution space*, which I define as a graph of code states submitted by previous students where the edges between states represent edits that transform one state into another. I describe the system with three main categories: *canonicalization*, which is the process for creating an abstract representation of student code; *path construction*, the process for determining what steps must be taken to go from an incorrect state to a corrected version; and *reification*, the process of undoing canonicalizations in order to represent hints in the student's original context. Finally, I describe how the whole process connects to create *hint generation,* by canonicalizing an incorrect state, identifying the best path to a solution, and then generating a hint in the student's original context.

I also provide technical evaluations of canonicalization and the whole hint generation process, to determine whether they are performing as would be expected on real student code, and I test whether the ITAP algorithm improves as it collects more data. Much of the work presented in this chapter was originally presented in Rivers & Koedinger (2015). I have made all of the code described in this section open source; it can be found at https://github.com/krivers/ITAP-django

Related Work: Data-Driven Tutoring

Data-driven tutoring and automatic hint generation have grown tremendously in recent years, possibly due to the surge in popularity of MOOCs in previous years (Pappano, 2012), which led to a more urgent need for hint mechanisms which would work at scale. However, at the initial point of publication of this work (Rivers & Koedinger, 2013), the field was still quite new and had several open questions. In particular, this early work led to what is arguably the first approach that could generate hints automatically 100% of the time (Rivers & Koedinger, 2014).

Methodologically, this work draws primarily on the concept of the Hint Factory (Barnes & Stamper, 2008), as was described earlier. In this section I describe the state of the field of data-driven tutoring, both prior to and after my entry into this research area (Rivers & Koedinger, 2012). I restrict my review to systems which are at least partially automated, at least partially use student-generated data, and generate feedback which is hint-like (that is, feedback which pertains in some way to what the student should do next). It is worth noting that a range of non-data-driven techniques can be used to generate feedback and hints for programming problems automatically. Many of them are described by Le (2016), including plan libraries, program transformation, constraint-based models, strategy-based models, and machine learning.

I focus here primarily on four questions. First, how do different approaches use student data? Second, how do these approaches decide what should be done next? Third, what kind of feedback is generated? And finally, how does this feedback affect students?

Use of Student Data

Initial research into data-driven tutoring used student submission data as it was presented, with no modifications. This was first attempted with a bootstrapping approach, where previously-collected student interaction log data was used to generate a skeletal model of a tutor for a collaborative software tool (McLaren et al, 2004). Further study into interaction log data for an equation-solving tutor found that individual student states could be matched to each other in order to reduce the number of possible paths, confirming that this approach could be feasible for tutor generation (Lin, Chou, & Chan, 2008). This led to the creation of the Hint Factory approach, which logged student submissions, turned them into chains of steps, and then collapsed identical steps across students together (Barnes & Stamper, 2008).

All of these initial approaches were attempted in domains where the number of possible submissions was restricted; therefore, to generate hints in more open-ended domains (such as programming), alternate solution representations needed to be used. Many different representations have been proposed over time. Some approaches use simplified versions of the programs to make comparison between states easier. These include my method of canonicalization (Rivers & Koedinger, 2013), which will be described in more detail later, and Singh's abstraction of submissions as program sketches (Singh, Gulwani, & Solar-Lezama, 2013), which can leave 'holes' in programs to be fixed. These approaches can work quite well for some problems, but may not be applicable to open-ended problems where solutions do not converge (Price & Barnes, 2015). Alternatively, program ASTs can be broken down into subcomponents, which can then be compared directly (Price, Dong, & Barnes, 2016).

Other approaches use some control flow representation to represent the main function of a student's submission without being overwhelmed by syntactic variation. These include variable linkage graphs, which separate out statements that change specific variables (Jin et al, 2012) and more traditional control flow graphs, which demonstrate how conditionals and loops interact in the program (Phothilimthana & Sridhara, 2017; Wang et al, 2017; Marin et al, 2017). This approach can substantially reduce the number of unique states, but also removes potentially useful semantic variation from the state.

Some approaches represent programs with their output instead of their code. This has been used to cluster submissions based on compiler errors (Hartmann et al, 2010), and to generate hints based on graphical program output (Peddycord III, Hicks, & Barnes, 2014). Using output can greatly reduce the number of unique states, but also removes almost all semantic information from the student state, which can make bottom-out hint generation difficult.

A few other approaches have used student-generated edits instead of student-generated submissions. These approaches all use program synthesis to generate hints, and therefore need a knowledge model to identify possible edits to apply to programs. In some cases, this model is derived based on edits seen from previous students, either through use of textual line edits (Lazar & Bratko, 2014) or program edits (Rolim et al, 2017; Head et al, 2017). Alternatively, commonly-appearing subexpressions across student submissions can be mined to produce similar knowledge models (Perelman, Gulwani, & Grossman, 2014; Marin et al, 2017). These approaches all have the benefit of supporting rapid generation of knowledge models for many individual problems.

Finally, new work is breaking down student code even further by attempting to learn recurrent neural networks for token sequences in student submissions (Bhatia & Singh, 2016). This approach has strong potential for correcting syntax errors, but is still limited in the types of errors it can fix.

My work primarily differs from the other approaches shown here as it was one of the first to use solution abstraction (via canonicalization) to represent student submission states. Of course, student data can be represented at many different levels of detail and abstraction, and the chosen representation depends mostly on how hint generation will be performed. This depends at least partially on generating possible next steps, which I will discuss next.

Choosing Next Steps

The essential content of a next-step hint depends on knowing what the next step a student takes should be. It is worth noting that the types of next steps students take naturally can vary greatly in purpose; students can move between subgoals, make corrections or errors, or rethink prior decisions (Hicks et al, 2015). Many different approaches for identifying the best next step have been developed, but most fall into four categories: identification of a previously used step (the Hint Factory approach), comparison of the student state to a correct state (my path construction approach), repair of a program with a knowledge model (the program synthesis approach), and human annotation of abstracted states (the crowdsourcing approach). I will discuss each of these categories in this section.

The first approach was proposed in the Hint Factory, which posited that previously observed student submission chains could be combined into a graph of all possible learning paths. This graph would then be turned into a Markov Decision Process to assign weights to edges; then, when a new student needed a hint, they could be found within the graph and told to move towards the best edge leading out of the state (Barnes & Stamper, 2008). This approach has been adopted in several domains outside of the original logic tutor, including a tutor for linked lists (Fossati et al, 2009), a tutor for basic programming problems using linkage graphs (Jin et al, 2012), an educational game teaching basic programming (Peddycord III, Hicks, & Barnes, 2014), and a SQL tutor (Lavbic, Matek, & Zrnec, 2016). The approach works very well in domains with restricted solution spaces, but has difficulty covering all possible steps in domains where student submissions do not map together quite so well; previous work has reported that even the more-restricted solution spaces can only generate hints around 85% of the time (Barnes & Stamper, 2008).

The second approach, comparing a student state to a goal state, has been used in many different systems. How the goal state is chosen varies over different implementations. Most implementations compare states to correct solutions directly, often using AST tree edit distance. This can be done to provide the goal itself as the next step (Ade-Ibijola, Ewert, & Sanders, 2015; Freeman, Watson, & Denny, 2016), as would be done with clustering; this kind of next-step allows students to see all possible differences, but in doing so gives much information away. Alternatively, the edit between the student's state and the goal can be broken up to provide the student with intermediate steps. This was first proposed in my work on path construction (Rivers & Koedinger, 2014), which will be described in more detail later; alternative and improved algorithms for next-step selection have been analyzed as well (Piech et al, 2015; Zimmerman & Rupakheti, 2015; Price, Dong, & Barnes, 2016). The approach has also been used to build chains of edits for style hints (Choudhury, Yin, & Fox, 2016), and in matching states with control flow information (Wang et al, 2017). The path construction approach makes it possible to provide smaller edits to students at any possible state, allowing full hint coverage, but cannot guarantee that those edits are always optimal, as automatically-generated intermediate states may never have been seen before.

The third approach, program synthesis-supported repair, relies on the concept of an error/knowledge model to identify possible fixes which can be applied to incorrect student states. Various algorithms have been developed which attempt to find the minimal set of fixes needed for a given incorrect program. This approach was originally attempted with the SKETCH system, which used an instructor-built expert model to fix mistakes in student code (Singh, Gulwani, & Solar-Lezama, 2013; D'Antoni, Samanta, & Singh, 2016; Phothilimthana & Sridhara, 2017). Additional approaches attempted to apply program synthesis using text-based approaches (Lazar & Bratko, 2014) and test-driven approaches (Perelman, Gulwani, & Grossman, 2014).

Recently, new program synthesis systems have been designed specifically to fix buggy code without requiring expert-built error models; these include Refazer (Rolim et al, 2017; Head et al, 2017) and SIMPL (So & Oh, 2017). And finally, some approaches have combined program repair with pattern matching to identify optimal fixes between incorrect and correct states (Gulwani, Radicek, & Zuleger, 2016; Marin et al, 2017). The program synthesis approach has many strengths, but it can also be somewhat slow in hint generation as it is constructing edits instead of comparing states, and hint generation cannot always be guaranteed, with hint coverage ranging from 70-90% across systems (Lazar & Bratko, 2014; Rolim et al, 2017; Gulwani, Radicek, & Zuleger, 2016).

Finally, some approaches use human support to identify the best next-step for a given student state. Often this human support is providing human annotation to data that has been observed before. This was the case in the original work my McLaren et al (2004); the skeletal model used by the bootstrapping approach had no information about correctness, and thus needed to be annotated by a tutor author. A similar approach is taken in clustering approaches, where experts were asked to identify correct states in given clusters, which can then be compared directly to other student submissions (Gross et al, 2012; Kaleeswaran et al, 2016).

Other approaches identify common incorrect states and, instead of identifying the next state, ask experts to annotate the incorrect state with feedback. This approach has been used to attach feedback to common incorrect subexpressions (Marin et al, 2017), and has also been used with crowdsourcing to provide feedback on a variety of incorrect states (Hartmann et al, 2010; Glassman et al, 2016). All of these human-based approaches have the benefit of containing feedback that is checked by a real human, but this benefit is also a downside; adding a human into the loop delays feedback, and make the system not fully automated as well.

A few additional approaches for next-step generation have been proposed which do not fall into the previous four categories. These include using spectrum-based fault localization to identify locations in the code that need to be fixed (Edmison, Edwards, & Perez-Quinones, 2017), using recurrent neural networks to predict which tokens need to be changed in syntactically invalid code (Bhatia & Singh, 2016), and building a knowledge base to predict any possible next step a student might need to take (Paquette et al, 2012; D'Antoni et al, 2015).

Altogether, I would conclude that none of these four approaches is better than the others; they all have strengths and weakness, and are best used in different contexts. The Hint Factory approach is most suitable to domains with small solution spaces, and a combination of the other three approaches can support most need cases in other domains. Human-generated feedback can provide insightful comments on common error states, program synthesis can be used to quickly fix small errors in the code, and solution-comparison can be used to support the remaining incorrect states, which cover a long tail of possible submissions.

My own approach was, as far as I can tell, the first fully-automated solution comparison approach in data-driven hint generation. It also provided one of the earliest demonstrations of how corrections could be broken down into multiple steps, instead of giving all corrections to the student at once. Of course, knowing how to choose a next step is only half the battle; once that information has been found, a system still needs to generate a hint for students to see.

Automatically-Generated Feedback Types

The systems I have described in the previous section generate many different types of hints. These range in content (location of bug, edit to fix bug, or entire solution) and form (direct edits, text messages, or visual cues). In this section I describe how these different hints can be categorized as the point, teach, and bottom-out hints used in traditional intelligent tutoring systems (VanLehn, 2006).

First, a few systems generate hints that can tell a student where errors are, but not how to fix them. These are similar to point hints, as they show a student where to look without giving the answer away. If the system knows where the change needs to occur, it can give the student the location by pointing at the line number where a change needs to be made to fix the program (Perelman, Gulwani, & Grossman, 2014). Alternatively, this information can be represented visually with a heatmap that highlights lines of code likely to be incorrect (Edmison, Edwards, & Perez-Quinones, 2017).

If the specific location is not known, the system could instead demonstrate how the program's output is different from what is expected by showing the student what the graphical output of a program should be at the next step (Peddycord III, Hicks, & Barnes, 2014). Finally, Hint Factory systems can sometimes use information about the path students are traversing to provide Hazard hints, which tell a student whether the steps they're currently taking are unproductive (Fossati et al, 2009; Eagle & Barnes, 2014). These hint types may nudge the student towards the right direction, but they require that the student do their own debugging; therefore, if a student cannot find the answer on their own, more assistance is needed.

The second hint level, 'teach' hints, gives the student high-level instruction about how to solve the problem without giving the solution away. These high-level hints are difficult to generate automatically, since current data-driven systems only know how to fix a program, and not why making a certain change fixes it. Therefore, most currently available teach hints are all generated using some kind of expert annotation (Singh, Gulwani, & Solar-Lezama, 2013; Marin et al, 2017; Head et al, 2017) or crowdsourcing (Hartmann et al, 2010). One exception can be found in the Hint Factory, which can provide subgoal hints to students with no annotation necessary. High-level hint generation works in this context because showing a student an intermediate step in the solution does not immediately solve the problem; students still must find logical rules to derive the intermediate stage themselves (Eagle & Barnes, 2014). It is not clear how this would be accomplished in programming, where code is both the state and the solution. Teach hints can be highly valuable, but require time and training for experts to generate.

Finally, the vast majority of data-driven hints fall into the bottom-out hint category, as they directly provide a student with the edit or solution they need to solve the problem. This can be done with the Hint Factory approach by demonstrating which rule needs to be applied (Barnes & Stamper, 2008) or which line needs to be changed (Jin et al, 2012). It can also be done with path construction (Rivers & Koedinger, 2015; Zimmerman & Rupakheti, 2015; Lavbic, Matek, & Zrnec, 2016; Price, Dong, & Barnes, 2016; Wang et al, 2017), program synthesis (Lazar & Bratko, 2014; D'Antoni, Samanta, & Singh, 2016; Gulwani, Radicek, & Zuleger, 2016; Rolim et al, 2017; So & Oh, 2017; Head et al, 2017), and other methods (Bhatia & Singh, 2016) to demonstrate which edits need to be applied to fix the program. These edits can be represented as text messages or highlighting in the IDE.

Some approaches show hints by comparing the student's solution to the goal state side-by-side, while highlighting differences (Gross et al, 2012; Ade-Ibijola, Ewert, & Sanders, 2015; Kaleeswaran et al, 2016). This can benefit students by supporting comparing and contrasting, but also makes it easier for students to apply hints directly without reading them thoroughly first. Finally, one approach generated somewhat higher-level edit hints by comparing the control flow of the two solutions and stating where the two flows differed (Phothilimthana & Sridhara, 2017). All of these approaches have the advantage of being able to support even the most confused of students, though they may also remove possible opportunities for debugging.

Finally, a few projects have explored the combination of multiple types of hints. Gross et al (2014) compared four different content types which could be used to generate bottom-out hints. Glassman et al (2016) explicitly labeled user-generated hints as point, teach, or bottom-out, and compared how these different types performed. Suzuki et al (2017) identified different hint types used in natural language and demonstrated how to recreate them automatically. It is likely that an optimal hint provision system will need to provide hints at multiple levels, to support students at different stages of learning.

My research does not attempt to define entirely new feedback types; instead, I have attempted to modify the content of ITAP's hints to provide different levels of instruction, as will be described in later sections. ITAP's original hints were modeled after the Hint Factory hints in that they told the student directly what to do. One would assume that these hints would help students learn, but to determine if that is the case, the learning process needs to be studied directly.

Effects of Automatic Feedback on Students

Now that I have described the data-driven tutoring systems that already exist, one main question remains: how do these systems perform with real students? Much research still needs to be conducted to determine how data-driven hints impact learning, but some studies have found early results.

First: what do these studies tell us about what kinds of hints should be generated? Some prior work suggests that smaller amounts of content (steps, not solutions) may be more appropriate for the student learning process (Gross et al, 2014). The same research suggests that those steps should come from expert solutions, which may be more beneficial to novices than other novice solutions. Additionally, the systems which are choosing these examples may need careful development to reach optimality. Out of two studies which asked experts to rate the optimality of chosen examples, one found that less than half of the examples were optimal (Gross et al, 2012), while another found only 20% non-optimal cases (Head et al, 2017).

Second: do automatically-generated hints improve the student learning experience? Previous work indicates that this is true. Prior studies have shown that having access to automatic hints results in spending less time on problems (Eagle & Barnes, 2013; D'Antoni et al, 2015), though this is potentially refuted by (Lavbic, Matek, & Zrnec, 2016). Additionally, students may require fewer submissions to solve problems with hints (Phothilimthana & Sridhara, 2017), and students complete more practice problems with hints (Barnes et al, 2008; D'Antoni et al, 2015). Therefore, these systems have promise.

However, efficiency is not the same as improved learning. Few studies have been conducted which directly measure learning; the only definite result reported thus far showed that in a style tutor, students with style hints improved dramatically more than students without (Choudhury, Yin, & Fox, 2016). However, multiple studies have shown that students will use the hints they are given and consider them beneficial (Hartmann et al, 2010; Price, Dong, &

Lipovac, 2017; Phothilimthana & Sridhara, 2017), so time may still demonstrate whether data-driven hints can directly impact the learning process.

The studies I report on later in this thesis have supported the previous results that automatically-generated hints help students spend less time with equal learning. However, they have potentially countered the results on hint content by demonstrating that higher levels of content may be better for improving student performance. Further research will be needed to substantiate these claims.

Canonicalization

In the first part of the hint generation process, the system needs to represent student data for optimal information processing. It is possible to compare student states in their original text, but this results in great amounts of non-meaningful variation, especially when ITAP compares across different students (who will likely have different styles). To remove some of this variation, I have created a suite of *semantics-preserving program transformations*. These transformations normalize the syntactic structure of the program (the code itself) while preserving semantic meaning (what the code does). If ITAP can apply an optimal set of transformations, any resulting differences between states should be semantic in nature, rather than syntactic. In this section, I describe how this process works, and explain the implementation of the canonicalizations used. The work described in this chapter was originally published in (Rivers & Koedinger 2012), though I have updated the canonicalizations since then; the most recent approach is described here.

Related Work: Data Representation

When considering how to represent data in tutoring systems, the tutor has to determine what content is necessary when evaluating and progressing from the student’s current state. This data can be abstracted at varying levels, where each level focuses on a different type of information which can be used to provide context on the student’s decisions.

One approach to data-driven tutoring relies on the original actions made by students, and generates hints only based on those actions seen before. These can be represented by interaction networks (Eagle, Johnson, & Barnes, 2012), which can efficiently model student interactions as a complex weighted network, where tutoring information is encoded into nodes and edges. In this model, both the student states and the student actions are necessary to represent the work that has been done. A similar type of model tracks student actions during the programming process (Carter, Hundhausen, & Adesope, 2015), though it does not analyze student code at the semantic level.

Other approaches view states and actions as independent, and use either the state or the action as a representation. Actions can be represented as text edits (Lazar & Bratko, 2014) or as AST edits (Rolim et al, 2017). In both cases referenced here, the actions can be used outside of their original contexts to provide feedback in new situations, for code states that have not been seen before.

I view the question of data representation more as a problem of representing the state than the action. A program's state can be represented by a set of features (Gross et al, 2012; Sudol, Rivers, & Harris, 2012), the output of the program (Peddycord III, Hicks, & Barnes, 2014), the execution of the program based on API calls or state changes (Piech et al, 2012; Paaßen, Jensen, & Hammer, 2016), or the actual code of the program. Furthermore, all these representation levels can be abstracted to various degrees. I am primarily interested in providing semantic hints about what a student should do next, so I do not use feature vectors, output, or execution; all of these would contain no information about how to actually change the program's code. Instead, I focus on varying levels of abstraction of a code state.

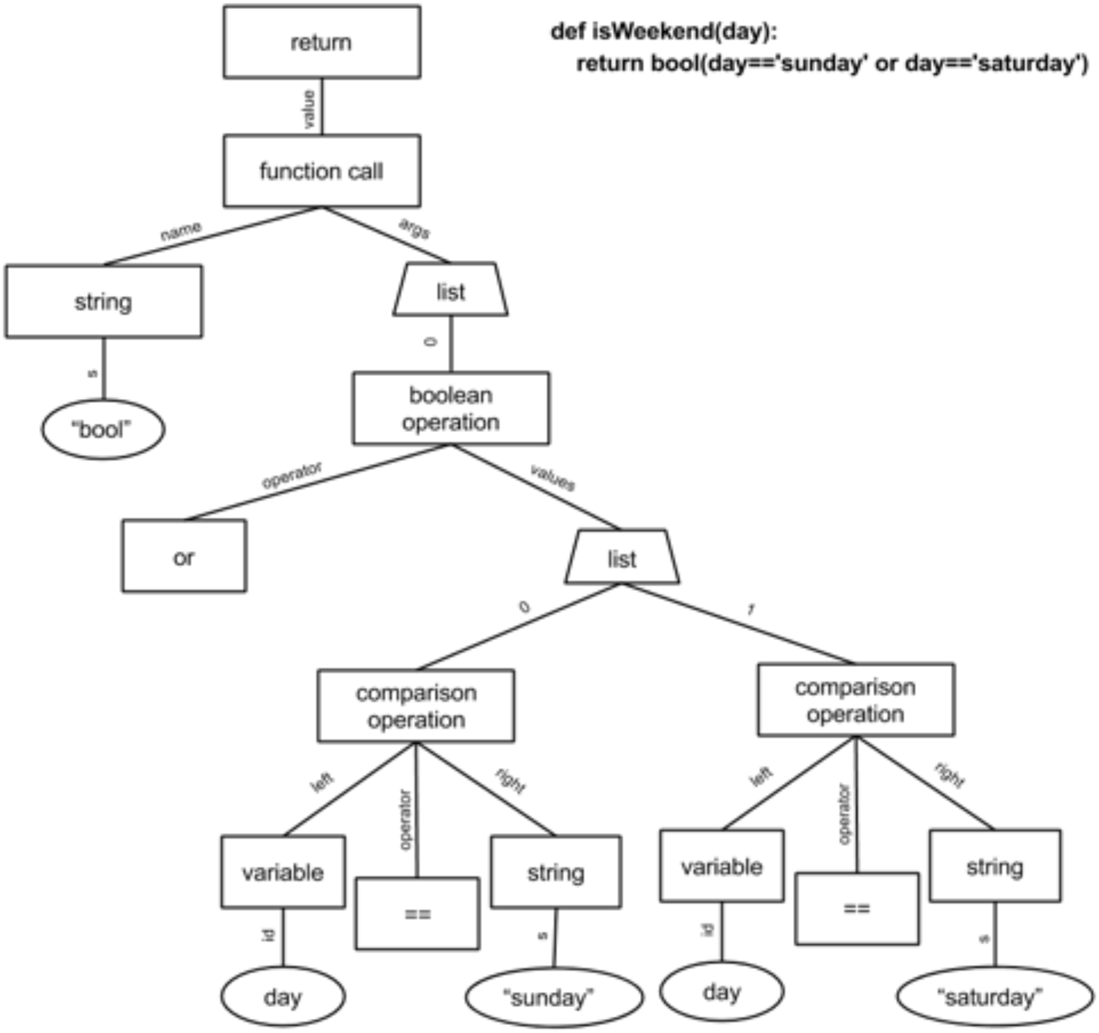
A program's code can be represented as text, a set of parsed tokens, an abstract syntax tree, or even bytecode. Programs can also be represented based on their control flow (Wang et al, 2017), data flow (Jin et al, 2012), or a combination of the two (Suarez & Sison, 2008; Srikant & Aggaral, 2013). Additional work has represented specific subsets of programs, to break larger programs up into pieces (Nguyen et al, 2014; Price, Dong, & Barnes, 2016). Since I want to be able to map back from the representation to the original text, I choose a level of representation which is highly abstracted while still bearing a full connection to the text: the abstract syntax tree.

Abstract syntax trees retain information that is purely syntactic, which can be modified using program transformations without harming the semantic meaning of the code. This canonicalization approach has been used before for educational purposes. Xu and Chee showed that program transformations could be used to facilitate comparing student programs to a teacher solution to provide difference feedback, and described a set of transformations they personally used (Xu & Chee, 2003). Further work showed that these transformations could also be used to supplement grading (Wang et al, 2007; Li et al, 2007). More recent work has used canonicalizations to support clustering code for the purposes of grading and feedback provision (Glassman et al, 2014), and has proposed control-flow abstract syntax trees, which reduce ASTs to only include control flow information, as an even more abstract representation for analyzing student programs (Hovemeyer et al, 2016).

Additionally, canonicalization has been used in intelligent tutoring systems for programming before. Early programming ITSs were often based on functional languages, where the accepted responses for questions were more constrained, making it easier to model the set of all possible solutions. Several researchers used program transformations to normalize these solutions or extend the set of possible solutions, to better support matching to student code. This has been done in the LAURA automatic debugging system for LISP programs (Adam & Laurent, 1980), in the Prolog Tutor (Looi, 1991; Gegg-Harrison, 1992), and in Ask-Elle, a tutor for Haskell (Jeuring et al, 2014). However, the technique has not been used in non-functional programming tutors, to the best of my knowledge, and I am not aware of any previous approaches which paired canonicalization with reification, to undo the changes made to student code.

Categories of Canonicalization

To run the canonicalization process with any given piece of code, it must first be transformed into an Abstract Syntax Tree (AST). An AST is an intermediate representation of a program created by the compiler during the conversion of a program from text into binary code; an example is shown in Figure 1. This format allows algorithms to directly modify the structure of a program, as it is represented by the tree. Any program which can be parsed by a compiler can be turned into an AST (as, indeed, compilers transform programs into ASTs internally during the process of creating executable code); to generate hints for non-parseable code, I use a different process, which will be described later.



```
def isWeekend(day):
    return bool(day=='sunday' or day=='saturday')
```

Figure 1: An example of a program code state and the corresponding AST.

Once ITAP has the AST (which can be generated simply using a built-in Python library), it runs each of the canonicalizations in turn, one by one. If the resulting AST has changed from its original state, it runs the set again, and continues doing so until the AST does not change anymore; once the state remains the same, it cuts off the process. This process should always terminate, as no two canonicalizations should ever reverse each other's changes.

Most of the canonicalizing functions described below only require the AST to run, but a few perform better when given additional information. For example, if the author knows the intended argument types for a function, providing that information will make it easier to understand the types the program can expect to encounter. Additionally, providing a set of 'given names' (like names of instructor-provided or -required functions) means that ITAP will not accidentally anonymize names that are required by the program.

Anonymization

First, I describe the process of *variable anonymization*. This is kept separate from the other categories because it can be applied outside of the canonicalization process to create anonymized but otherwise unchanged states. This is useful during hint generation, for reasons that will be discussed later.

To anonymize the variable names of a piece of code, ITAP first identifies any names which should remain unchanged in the program. This includes the main function's name, any instructor functions that will be standardized across submissions, imported module names, and built-in function names (like `sum` and `max`). It creates a dictionary which maps these names to themselves, to establish that they should remain unchanged. It then goes backwards through the main body of the code (which contains functions, imports, and global assignments), identifying all global names and mapping them to anonymized global ids. Helper functions are anonymized to `helper<num>`, where `<num>` goes up by one for each new helper function seen, while global variables are represented by `g<num>`, with `<num>` again increasing with each new variable.

Once all the global values have been identified, ITAP gathers the local scope of each individual function. Here, function parameters are mapped to `p<num>_<functionname>` and local variables are mapped to `v<num>_<functionname>`. Each local variable is followed by the parent function's name to ensure that variables do not clash across helper functions during code matching later.

Once the full scope has been put into the dictionary, ITAP traverses the functions again, this time changing any encountered variable to its new name. Original names are stored in the ast.Name object's metadata, under the tag `originalId`. At this point, it also identifies variables that do not appear in the dictionary. These are random variables, as they have not had a value assigned to them (and will therefore likely crash the code when reached). It maps these variables to `r<num>_<functionname>`, and gives them an additional metadata tag, `randomVar`. Finally, when it encounters a variable with a given name (where the variable's name is mapped to itself in the dictionary), it is given the metadata tag `dontChangeName`, to track that this name should never be anonymized.

At this point, the function should be fully anonymized. An example of a function pre- and post- anonymization is shown in Figure 2 (with the name `find_the_circle` replaced with `ftc` for readability). Note that most variables are mapped to `v<num>` or `p<num>`, but in the helper function, `lst` is mapped to `r0`, as it is not defined in the local or global scope.

```
def ftc(lst):
    for i in range(len(lst)):
        for j in range(len(lst[i])):
            if lst[i][j] = 'o':
                return helper(i, j)

def helper(i, j):
    y = len(lst) - 1 - i
    x = j
    return [x, y]
```

```
def ftc(p0_ftc):
    for v0_ftc in range(len(p0_ftc)):
        for v1_ftc in range(len(p0_ftc[v0_ftc])):
            if p0_ftc[v0_ftc][v1_ftc] = 'o':
                return helper_g0(v0_ftc, v1_ftc)

def helper_g0(p0_helper_g0, p1_helper_g0):
    v0_helper_g0 = len(r0_helper_g0) - 1 - p0_helper_g0
    v1_helper_g0 = p1_helper_g0
    return [v1_helper_g0, v0_helper_g0]
```

Figure 2: A piece of code before and after anonymization takes place.

Simplification

Several of the canonicalization functions are *simplifying*; they remove excess code and generally streamline programs. This allows ITAP to map together both verbose and succinct implementations that are semantically identical.

First, there are a few small simplifications which are performed primarily to simplify the AST itself. These are also used to simplify the implementations of following canonicalizations. For example, in Python, it is possible to perform multiple assignments on a single line. ITAP separates out the multiple assignments into independent lines, as is shown in Figure 3. This will be evaluated in the same way, but is also more likely to coincide with other programs it has already seen.

`a = b = c`	`b = c`
	`a = b`

Figure 3: An example of multi-assignment simplification

Additionally, assignment can be done on separate items on the same line using tuples or lists. ITAP separates out the multiple assignments here as well, when possible; see Figure 4.

`(a, b) = (c, d)`	`a = c`
	`b = d`

Figure 4: An example of tuple assignment simplification

Augmented assignments are also used in Python, which allow variables to be updated using shorthand. ITAP separates these out by displaying the full operation; see Figure 5.

`a += b`	`a = a + b`

Figure 5: An example of augmented assignment simplification

Finally, Python allows *multi-comparisons*, where multiple comparisons are joined in the same operation. ITAP separates these out into individual comparisons joined by `and` operators, to achieve the same semantic effect; this is shown in Figure 6.

`a < b <= c`	`(a < b) and (b <= c)`

Figure 6: An example of multi-comparison simplification

The above simplifications are used to simplify the built-in Python AST format, but ITAP also adds additional information to the AST by propagating expected type values throughout the code. It does this because Python is weakly typed, which means that variables can change types or not be assigned to the expected type without causing a compiler error. A side effect of this is that the AST does not, on its own, know the types of its various variables. This makes it difficult to tell whether given pieces of code might crash, and if ITAP does not know whether a piece of code might crash, it does not know if it will be safe to move or delete that code later.

If the problem statement guarantees that input for a given function will always take on specific types (as many problems do), ITAP can assign those types as metadata to the function's parameters and propagate it through the rest of the code. It does exactly that, giving variables a `type` tag which maps to the guaranteed type of the variable (if it can be determined), and otherwise mapping to `None`. I also define a function `eventual_type`, which can determine the eventual type of a Python expression if enough information is provided. Using these functions, ITAP can propagate metadata throughout all of the code about types for later use.

In addition to the simplifications described above, ITAP runs four traditional compiler optimizations on student code: function inlining (Chang & Hwu, 1989), constant folding (Wegman & Zadeck, 1991), copy propagation (Aho, Sethi, & Ullman, 1986), and dead code elimination (Kennedy, 1979). Implementations for these functions can be found elsewhere, so I only briefly describe how each function works.

Function inlining is used to take the body of a helper function called by the main function and replace the original call to the helper function with that code. This can only be done in certain restrained circumstances (often when the control flow of the helper function is very simple), but it can be useful for the purpose of removing unnecessary helper functions written by students still trying to understand function decomposition. An example of function inlining is shown in Figure 7.

```
def helper_g1(p0_helper_g1, p1_helper_g1):
    return ((p0_helper_g1 // 10) * (p1_helper_g1 - 1))

def helper_g0(p0_helper_g0):
    v0_helper_g0 = p0_helper_g0
    v1_helper_g0 = 0
    while (v0_helper_g0 > 0):
        v0_helper_g0 = (v0_helper_g0 // 10)
        v1_helper_g0 += 1
    return v0_helper_g0

def sum_of_odd_digits(p0_sum_of_odd_digits):
    sum = 0
    if ((helper_g0(p0_sum_of_odd_digits) % 2) == 0):
        for v0_sum_of_odd_digits in range(1, helper_g0(p0_sum_of_odd_digits), 2):
            sum += helper_g1(p0_sum_of_odd_digits, v0_sum_of_odd_digits)
    else:
        for v0_sum_of_odd_digits in range(1, (helper_g0 + 1), 2):
            sum += helper_g1(p0_sum_of_odd_digits, v0_sum_of_odd_digits)
    return sum
```

```
def helper_g0(p0_helper_g0):
    v0_helper_g0 = p0_helper_g0
    v1_helper_g0 = 0
    while (v0_helper_g0 > 0):
        v0_helper_g0 = (v0_helper_g0 // 10)
        v1_helper_g0 += 1
    return v0_helper_g0

def sum_of_odd_digits(p0_sum_of_odd_digits):
    sum = 0
    if ((helper_g0(p0_sum_of_odd_digits) % 2) == 0):
        for v0_sum_of_odd_digits in range(1, helper_g0(p0_sum_of_odd_digits), 2):
            sum += ((p0_sum_of_odd_digits // 10) * (v0_sum_of_odd_digits - 1))
    else:
        for v0_sum_of_odd_digits in range(1, (helper_g0 + 1), 2):
            sum += ((p0_sum_of_odd_digits // 10) * (v0_sum_of_odd_digits - 1))
    return sum
```

Figure 7: A piece of code before and after function inlining has been performed. Note that `helper_g1` is only a single line, and thus can easily be inlined.

Constant folding is used by compilers to calculate constant operations in the code at compile-time instead of waiting to determine them at runtime. ITAP identifies operations that can be performed ahead of time and simplifies the code to show the resulting value. This is not restricted only to operations on constant values; for example, given the operation `(x + 0)`, if it knows `x` is a number, it can simplify the expression to `(x)`. In some occasions, these operations are used to adjust the type of an expression; for example, the expression `(1.0 * x)`, when `x` is a number, is meant to turn `x` into a float. To normalize code, it transforms expressions such as this into the direct type-cast instead (`float(x)`). An example of constant folding is shown in Figure 8.

```
def has_extra_fee(p0_has_extra_fee, p1_has_extra_fee):
    if ((p0_has_extra_fee > 5) or (p1_has_extra_fee == p1_has_extra_fee)):
        return False
```

```
def has_extra_fee(p0_has_extra_fee, p1_has_extra_fee):
    if ((p0_has_extra_fee > 5) or True):
        return False
```

Figure 8: A piece of code before and after constant folding has been performed. Note that obviously `p1 == p1`, so that operation can be replaced with `True`.

Copy Propagation is used to propagate variable values into the code, which can be paired with Dead Code Elimination to remove unnecessary variables. In copy propagation, ITAP keeps track of which variables are *live*; that is, which variables have not been changed since they were last assigned. If a variable is live when it is used in an expression, ITAP copies its value into the variable's place. Dead code elimination checks for *dead code* (code which is not used or will never be reached). This includes any code that occurs after a return statement, variables that are never used, and expressions that are run by themselves without changing the state. An example of both copy propagation and dead code elimination is shown in Figure 9.

```
import math
def convert_to_degrees(p0_convert_to_degrees):
    degrees = ((p0_convert_to_degrees * 180) / 3.14)
    return degrees
```

```
import math
def convert_to_degrees(p0_convert_to_degrees):
    degrees = ((p0_convert_to_degrees * 180) / 3.14)
    return ((p0_convert_to_degrees * 180) / 3.14)
```

```
import math
def convert_to_degrees(p0_convert_to_degrees):
    return ((p0_convert_to_degrees * 180) / 3.14)
```

Figure 9: The code shown at the top is run through copy propagation to get the second state (middle), where the value of `degrees` has been copied into the return statement. ITAP then runs through dead code elimination to get the third state (bottom), where the initial variable assignment is removed.

Ordering

The second set of canonicalizing functions I describe are *ordering functions*. These functions impose a strict ordering on Python expressions so that ITAP can standardize the ordering of commutative (reorderable) operations in different pieces of code. This makes it much easier for it to compare functions across multiple students, and usually improves the accuracy of hint generation later.

The ordering function checks for non-AST values (such as strings and numbers) first, putting them after ASTs in ordering (so that an expression like `(1 + x)` will be reordered to `(x + 1)`). It then orders among AST nodes by imposing a strict ordering on node types; statements come before expressions, which come before operations, and all the individual types among these are ordered as well. Within identical node types, ordering is done first based on the depth of the tree (so that larger expressions go later); if the depths are the same, it compares the attributes of the two trees instead.

When canonicalizing, ITAP imposes this ordering on Boolean operations, binary operations, comparisons, and conditionals. In this section I will demonstrate the effect this ordering has on code for each of these different types.

First, consider basic commutative operations. Binary operations take two inputs and evaluate them based on some (often-mathematical) operator. When this operation is addition, multiplication, or a bitop, and when ITAP knows the inputs are numbers, it can reorder the inputs based on the ordering from before without causing any changes. The same is true of Boolean operations, as long as all of the inputs are Boolean-typed and none of them can crash; ITAP also tries to combine adjacent Boolean operations that have similar components. For comparisons, it can reorder the inputs for `==` or `!=` operations, and it also changes `>` and `>=` operations to `<` and `<=`, just to keep everything consistent. Boolean and comparison reordering is demonstrated in Figure 10.

```
def no_positive_even(p0_no_positive_even):
    for v0_no_positive_even in range(len(p0_no_positive_even)):
        if ((v0_no_positive_even > 0) and ((v0_no_positive_even % 2) == 0)):
            return False
    return True
```

```
def no_positive_even(p0_no_positive_even):
    for v0_no_positive_even in range(len(p0_no_positive_even)):
        if (((v0_no_positive_even % 2) == 0) and (0 < v0_no_positive_even)):
            return False
    return True
```

Figure 10: Code that has had the ordering function imposed on it. The Boolean operation in the `if` statement has switched the operations (to put the longer expression on the left); the comparison operation has also switched direction, to become a `<` operation.

Additionally, ITAP tries to reorder some binary operations in order to reduce the number of negations, and it uses De Morgan's law to reduce the number of `not` operators in Boolean operations. For negations, it identifies expressions such as `x + (-y)` and changes them to `x - y`; for `not` operations, it uses De Morgan's law to turn expressions such as `(not (x == True))` into `(x != True)`. Examples of this are shown in Figures 11 and 12.

```
def nearest_bus_stop(p0_nearest_bus_stop):
    if ((p0_nearest_bus_stop % 8) > 4):
        return (p0_nearest_bus_stop + (8 - (p0_nearest_bus_stop % 8)))
    else:
        return (p0_nearest_bus_stop - (8 - (p0_nearest_bus_stop % 8)))
```

```
def nearest_bus_stop(p0_nearest_bus_stop):
    if ((p0_nearest_bus_stop % 8) > 4):
        return (p0_nearest_bus_stop + (8 - (p0_nearest_bus_stop % 8)))
    else:
        return (p0_nearest_bus_stop + ((p0_nearest_bus_stop % 8) - 8))
```

Figure 11: An example of negation reduction in student code. Note how the `else` statement's `return` value changes.

```
def is_prime(p0_is_prime):
    v0_is_prime = round((p0_is_prime ** 0.5))
    if (p0_is_prime <= 1):
        return False
    else:
        if (not (p0_is_prime == 2)):
            for v1_is_prime in range(3, ((p0_is_prime ** 0.5) + 1), 2):
                if ((p0_is_prime % v1_is_prime) == 0):
                    return False
        return True
```

```
def is_prime(p0_is_prime):
    v0_is_prime = round((p0_is_prime ** 0.5))
    if (p0_is_prime <= 1):
        return False
    else:
        if (p0_is_prime != 2):
            for v1_is_prime in range(3, ((p0_is_prime ** 0.5) + 1), 2):
                if ((p0_is_prime % v1_is_prime) == 0):
                    return False
        return True
```

Figure 12: An example of De Morgan's law applied to student code. Note how the `if` statement inside the `else` changes.

Apart from these operation reorderings, ITAP can also impose specific orderings on conditional statements. In fact, there are many ways in which ITAP can combine and order conditionals! When conditionals are connected in if-elif-else trees with disjoint tests, it can reorder those tests according to the ordering operation. It can also switch if-body and else-body orderings in cases where the else body is smaller than the if. Additionally, when connected conditional tests achieve the same result (such as returning a value), ITAP can combine those tests; an example is shown in Figure 13.

```
def can_make_breakfast(p0_can_make_breakfast, p1_can_make_breakfast):
    if (p0_can_make_breakfast < 11):
        if (not p1_can_make_breakfast):
            return True
    return False
```

```
def can_make_breakfast(p0_can_make_breakfast, p1_can_make_breakfast):
    if ((p0_can_make_breakfast < 11) and (not p1_can_make_breakfast)):
        return True
    return False
```

Figure 13: An example of the effect of conditional reordering on a piece of code. As both `if` statements must succeed to reach the `return True` statement, ITAP can combine them with an `and` operator.

ITAP can also recognize separate but disjoint if statements and connect them, which can help improve ordering using the methods listed above. This is done by determining whether the tests are disjoint, which can be done for many comparison operations. An example is shown in Figure 14.

```
def has_extra_fee(p0_has_extra_fee, p1_has_extra_fee):
    if (p0_has_extra_fee <= 5.0):
        return False
    if (p0_has_extra_fee > 5.0):
        return True
```

```
def has_extra_fee(p0_has_extra_fee, p1_has_extra_fee):
    if (p0_has_extra_fee <= 5.0):
        return False
    elif (p0_has_extra_fee > 5.0):
        return True
```

Figure 14: Another effect of conditional ordering on student code. Because `p0 > 5` is disjoint from `p0 <= 5`, the two conditionals can be combined into one conditional tree.

Domain-Specific

Finally, there are a set of canonicalizing functions which are not used for generic simplification or ordering, but instead target specific oddities of novice code which can be reduced to great effect. These functions were mostly created by examining student code to find places where inefficient code was being written, and cases where semantically identical code was not mapped to the same canonical states.

First, ITAP looks for cases where default values are used (in slices and ranges), even when they are not required. Both slices and ranges in Python take the form of `value[start:end:step]`, where start defaults to `0`, end defaults to `len(value)` (for slices), and step defaults to `1`. However, there are some cases where students include default values where they are not needed. In these cases, ITAP can remove the default values entirely; see Figure 15 for an example.

```
def first_and_last(p0_first_and_last):
    return (p0_first_and_last[0] + p0_first_and_last[:len(p0_first_and_last)])
```

```
def first_and_last(p0_first_and_last):
    return (p0_first_and_last[0] + p0_first_and_last[:])
```

Figure 15: An example of the simplification of default values. Note that `len(p0)` can be removed because it is already the default value for the slice.

Next, ITAP looks for cases where unnecessary type-casting is performed. These are simply cases where students try to cast a value to a type, when the value is already that type. Since the cast is unnecessary, it can be removed altogether. Of course, ITAP only removes these casts in conditions where it knows the type of the expression (using type metadata and the `eventual_type` function). An example is shown in Figure 16.

```
def over_nine_thousand(p0_over_nine_thousand):
    return bool((p0_over_nine_thousand > 9000))
```

```
def over_nine_thousand(p0_over_nine_thousand):
    return (p0_over_nine_thousand > 9000)
```

Figure 16: An example of the simplification of type casting. Because comparisons always evaluate to Boolean values, ITAP does not need to cast them.

Additionally, ITAP finds several cases where a student checks whether `boolean_value == True`, even though this is entirely unnecessary, as the result of this check is the same as the Boolean value itself. This is often used in conditional statements, perhaps because students do not fully understand how Boolean variables work, or perhaps because they think single values cannot be used as conditional tests. Either way, it can simplify these expressions by removing the `== True` part of the expression, as is shown in Figure 17. It does the same for `value == False`, turning it into `not value`.

```
def has_extra_fee(p0_has_extra_fee, p1_has_extra_fee):
    if ((p0_has_extra_fee < 5.0) and (p1_has_extra_fee == True)):
        return False
    if ((p0_has_extra_fee < 5.0) and (p1_has_extra_fee == False)):
        return True
```

```
def has_extra_fee(p0_has_extra_fee, p1_has_extra_fee):
    if ((p0_has_extra_fee < 5.0) and p1_has_extra_fee):
        return False
    if ((p0_has_extra_fee < 5.0) and (not p1_has_extra_fee)):
        return True
```

Figure 17: An example of code transformation via removal of `== True`.

Finally, ITAP can (again) simplify various unusual uses of conditionals. First, it can fix cases where students use redundant statements, by moving those statements outside of the conditionals. An example of this is shown in Figure 18, where the return statement is moved outside of the conditional as it appears in both branches. Second, it can collapse conditional statements that are entirely unnecessary (where both branches have identical bodies), by moving the branch body outside and deleting the conditional. If there is a chance that the conditional test will crash, ITAP moves it into an expression that occurs before the conditional's body.

```
import math
def nearest_bus_stop(p0_nearest_bus_stop):
    v0_nearest_bus_stop = (p0_nearest_bus_stop % 8)
    if (v0_nearest_bus_stop <= 4):
        v1_nearest_bus_stop = ((p0_nearest_bus_stop // 8) * 8)
        return v1_nearest_bus_stop
    else:
        v1_nearest_bus_stop = (((p0_nearest_bus_stop // 8) * 8) + 8)
        return v1_nearest_bus_stop
```

```
import math
def nearest_bus_stop(p0_nearest_bus_stop):
    v0_nearest_bus_stop = (p0_nearest_bus_stop % 8)
    if (v0_nearest_bus_stop <= 4):
        v1_nearest_bus_stop = ((p0_nearest_bus_stop // 8) * 8)
    else:
        v1_nearest_bus_stop = (((p0_nearest_bus_stop // 8) * 8) + 8)
    return v1_nearest_bus_stop
```

Figure 18: An example of code transformation via moving of redundant conditional statements. As the `return` statement appears in both branches, it can be moved outside the `if` statement entirely.

Altogether, these various transformations are used to generate canonical states. It is worth noting that these transformations were tailored to work specifically for Python ASTs; the compiler optimizations and ordering functions could be imposed on other language ASTs with reimplementation, but the domain-specific transformations may not be transferable. Still, transformations work across all problems, as they are associated with the whole language instead of specific concepts.

Evaluation of Canonicalization Reduction

To evaluate the canonicalizations, I must address their original purpose: do they adequately map together syntactically independent states that are semantically equivalent? To determine whether this is the case, I generate solution spaces with student states at different levels of state abstraction, to see if I can *reduce* the size of the solution spaces with higher abstraction (by mapping more program states together).

When running this technical evaluation (and the evaluations which follow), I use student code submissions collected from a series of studies described in the Chapters 3-5. I combined problem data across studies in cases where the problem statement did not change from one study to another, then identified problems from the resulting set which were attempted by at least ten students and had encountered at least 100 states (to ensure that reduction could be detected), where a state is defined as a program representation at some level of abstraction. This resulted in a set of 41 problems, where the smallest dataset had 122 states and the largest had 1065 (with an average of 404 states). The problems range in complexity as well, from 3 tokens in the simplest solution to 56 tokens in the most complex (with an average of 20 tokens). Full summary statistics for all the problems used in this and following technical evaluations are included in Appendix 2.

In this analysis, I do not use states which cannot be parsed by the compiler (as ITAP must have an AST to perform canonicalization); these syntactically incorrect states take up 14% of the space on average. I also ignore states that are immediate duplicates of prior submissions (which often happens when students get impatient and click Submit several times); this leaves 91 states in the smallest solution space and 697 in the largest (average of 263). Note: in the following analyses, the reduction reported for each stage of abstraction is computed based on the solution space size of the abstraction stage that came before it, so that the effects of each abstraction stage can be compared independently.

With the above-mentioned dataset, I investigate the size of the solution space for varying levels of abstraction: original submissions, unique text submissions, unique AST submissions, anonymized submissions, and canonicalized submissions. Unfortunately, I was not able to calculate an accurate reduction amount for unique text submissions or unique ASTs at this time, due to a bug in the logging software that replaced original code text with an AST-normalized version that obfuscated differences between text submissions; I will leave this for future work. However, I can still look for additional reduction contributed by anonymization and canonicalization, as well as reduction from original submissions to the unique AST version. I also realized that the IDE used in one of the studies provided a ‘see our solution’ feature to students after they had attempted to solve the problem at least once, which artificially inflated the appearances of the provided solution. To account for this, I remove those provided solutions from the data set.

Averaging over the 41 problems shows that reduction from original submissions to unique ASTs is quite high (35% fewer unique states), while reduction due to anonymization and canonicalization is rather lower (8% and 9% on the resulting solution spaces respectively), but each reduction is significant according to a paired t-test ($p < 0.001$; original means and standard deviations reported in Table 1). Overall, the average solution space was reduced to 55% its original size. I also found that unique AST reduction was strongly negatively correlated with token complexity ($r = -0.65$) and canonicalization reduction was moderately negatively correlated with token complexity ($r = -0.36$); in other words, more reduction was possible in programs that had less tokens.

Representation	Mean	Std. Dev
Original Submissions	238.93	127.94
Unique AST	157.63	105.52
Anonymized State	143.05	92.10
Canonicalized State	131.90	87.06

Table 1: The mean and standard deviation for number of unique states in each representation level across 41 problems. The high variance is due to significantly different submission rates across problems.

I also investigate the number of correct states per problem with varying levels of abstraction, as one would expect many more possible incorrect states in a solution space than correct states. The problems ranged from having 12 to 264 correct solutions recorded (average of 100). Averaging over the same 41 problems shows that reduction from original correct submissions to unique ASTs continues to be strong (48%), and that reduction due to anonymization and canonicalization has doubled (20% and 21% respectively). Overall, the average set of correct states was reduced to 34% its original size. All reductions were significant according to paired t-tests ($p < 0.01$; means and standard deviations reported in Table 2). AST reduction was strongly negatively correlated with complexity ($r = -0.66$), and canonicalization reduction was moderately negatively correlated with problem complexity ($r = -0.34$).

Representation	Mean	Std. Dev
Correct Submissions	76.00	47.85
Unique AST	33.68	21.81
Anonymized State	25.17	16.82
Canonicalized State	20.34	14.54

Table 2: The mean and standard deviation for number of unique correct states in each representation level across 41 problems. The high variance is due to significantly different submission rates across problems.

It is worth noting here that ITAP can apparently reach a stable number of correct states per problem, though the same cannot be said of the entire solution space. First, I note that the number of unique canonicalized states is highly correlated with the original number of submissions (r=0.87), which implies that more submissions lead to more unique states. However, the number of unique canonicalized states among only correct solutions is only moderately correlated with the total number of correct submissions (r=0.40). This point is illustrated in Figure 19. The first graph shows the number of unique states for each format, organized by total number of submissions; here it is obvious that the number of states tends to increase with the number of total submissions. The second graph shows the same data for only correct submissions; here, the number of more abstracted submissions does not appear to linearly increase with the number of correct submissions.

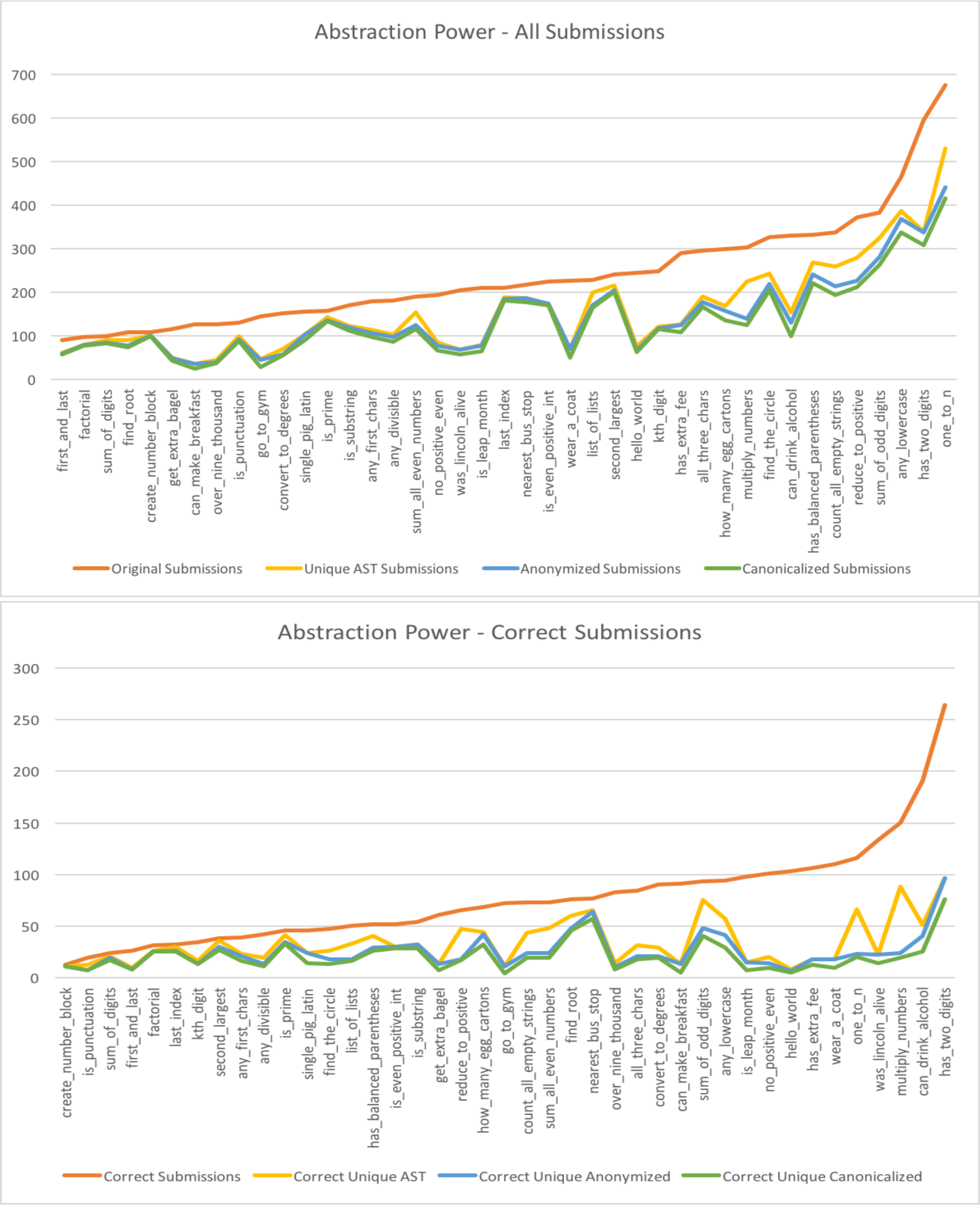


Figure 19: The abstraction power for the entire solution space and the set of correct states alone, organized by total number of submissions. Though the number of canonicalized states increases with the number of total submissions for the whole solution space, it remains static for the set of correct states.

Altogether, these results suggest that student solutions may naturally converge more than I initially expected, considering how much reduction could be achieved with unique AST reduction alone. This is mainly due to the simplicity of many of the problems in the dataset; with more complex problems, I would not expect such convergence, and the negative correlation between solution complexity and reduction supports this theory. I also found that most reduction occurred with correct states (as expected), and that ITAP appears to be able to identify a stable set of correct solutions despite increasing submission rates. To determine if this is the case, I will need to run studies with real teachers to see whether their clustering of solutions matches the clustering performed by ITAP; however, I leave this for future work.

Path Construction

In this section, I describe how the path construction algorithm can generate the content used in hints. The goal of path construction is to identify the closest correct/goal state (where a correct state is one which passes all the problem's designated test cases) to a submitted incorrect state, then to link the submitted state to that goal with a series of edits. The work described in this chapter was originally described in (Rivers & Koedinger, 2014), though I have modified the algorithm since that point; the more recent version of the algorithm is included here.

Algorithmic Description

Before I describe the path construction algorithm, I have to explain a few terms that will be used in the upcoming section. As most of path construction relies on being able to compare and modify ASTs, it is easiest to describe the functions that make this possible upfront.

AST Algorithms

First, I explain how the edits between ASTs are represented. When diffing AST `x` and AST `y`, the system identifies a set of *edit vectors* which can be applied to `x` to turn it into `y`. These edit vectors are comprised of several properties:

- **Type:** The type of the edit vector describes what kind of change is being made. Currently, ITAP can generate seven types:
    - *Change*: replaces one value with another
    - *Subset*: replaces a value with a subset of itself
    - *Superset*: replaces a value with a superset of itself
    - *Add*: inserts a new value into the specified location
    - *Delete*: removes a value from the specified location
    - *Swap*: swaps the positions of two values.
    - *Move*: moves the value at the indicated position to a new position
- **AST:** The AST which the edit vector should be applied to. When transforming `x` into `y`, this is `x`.
- **Path:** A list of nodes and indices that can be traversed through the starting AST to reach the location in the tree where the edit occurs.
- **Old and New Expressions:** The old and new values described above in the varying edit types. The old value is taken from `x`, while the new value is taken from `y`.

As edit vectors are intended to work only on the specified AST, ITAP often needs to modify them if it needs to change the starting AST they are applied to. This usually happens when several edits need to be applied consecutively, as earlier edits may change the position of the following ones. When this happens, it identifies the changes between the original and new ASTs and records them in the edit vector in a metadata map. This allows ITAP to apply the edit vectors to the updated locations in the new AST. Most often, this is done during the goal optimization stage, which will be described later.

Next, I explain how ITAP generates the edit vectors between two ASTs; that is, how it performs a diff operation on two ASTs. This can be broken down into two questions: how to compare two AST nodes, and how to compare two lists. Comparing AST nodes is simple; ITAP checks whether the node types are equal. If they are, it recursively generates edit vectors for the diffs of the fields of each item. If they are not equal, it checks whether the first AST occurs within the second, or vice versa; this can generate a Subset or Superset vector. If that is not the case, it creates a Change vector between the two nodes.

Comparing two lists is slightly harder, as ITAP aims to provide optimal matching (so that the number of edits required is minimized, which should make those edits sensible). To do this, it takes the two lists (`x` and `y`) and identifies the optimal item-matching between the two (that is, a mapping from the index in `x` to the index in `y`). Matching is done via the following steps:

1. First, match items that are exactly the same (prioritizing items on the same line)
2. Next, match items that are the same type (prioritizing lower distance between items according to the distance metric, then distance between indices)
3. Next, match items that occur in the same position
4. Finally, match the remaining items in order. Extra items in either list are marked.

Once ITAP has the mapping between the two lists, it identifies the set of edit vectors needed. First, it generates vectors based on position. The items marked as unmatched are turned into Delete vectors (if they came from `x`) or Add vectors (if they came from `y`). For the remaining item pairs, it generates two lists: one a set of indices in the normal order from `0` to `n` (`A`), the other the set of indices ordered so that they're paired with the first set (`B`). ITAP has to find a set of edit vectors that will turn `A` (the current ordering of lines in `x`) into `B` (the current ordering of lines in `y`). It does this with the following algorithm, which tries to find an optimal set of vectors from the outside in:

1. If `A[0] == B[0]`, run recursively on `A[1:]` and `B[1:]`
2. If `A[-1] == B[-1]`, run recursively on `A[:-1]` and `B[:-1]`
3. If `A[0] == B[-1]` and `A[-1] == B[0]`, generate a Swap vector between `A[0]` and `A[-1]`, and run recursively on `A[1:-1]` and `B[1:-1]`
4. If `A[0] == B[-1]`, generate a Move vector from `A[0]` to `A[-1]`, and run recursively on `A[1:]` and `B[:-1]`

5. If `A[-1] == B[0]`, generate a Move vector from `A[-1]` to `A[0]`, and run recursively on `A[:-1]` and `B[1:]`
6. Otherwise, identify the index `i` in `B` where A[0] occurs, generate a Move vector from `A[0]` to `A[i]`, and run recursively on `A[1:]` and `B[:i] + B[i+1:]`

An example of how this algorithm works is shown in Figure 20. This algorithm is optimal for lists up to size four; for larger lists optimality is not guaranteed, but such large lists are relatively rare in novice programs. Once all the positional vectors have been generated, ITAP compares the item-pairs to identify any further edit vectors that need to be generated between the ASTs.

	List A (old)	**List B (new)**	**Actions**
Initial	[0,1,2,3,4,5,6]	[0,6,3,4,2,5,1]	0 matches already
Step 1	[1,2,3,4,5,6]	[6,3,4,2,5,1]	Swap 1 and 6
Step 2	[2,3,4,5]	[3,4,2,5]	Move 2 behind 4
Step 3	[3,4,5]	[3,4,5]	Done!

Figure 20: An example of how position edit vectors are generated. Initially, 0 can be discarded as the positions already match; next, 1 and 6 can be swapped; finally, moving 2 results in a perfect match.

The paths for the edit vectors are constructed as the vectors are passed back up the recursive chain, by noting each parent node and list position that the chain passes through. With this, ITAP can recursively generate sets of edits between any two given ASTs.

Once ITAP knows how to diff two ASTs, it can use this procedure to calculate the distance between two trees (which will be useful when selecting optimal paths and goals). To do this, it divides the weight of the changes between the two trees by the base weight of the two trees. In this context, a tree's weight is calculated based on the number of nodes that appear in the tree. The base weight is simply the maximum of the two trees' weights; it uses this as the base because in the worst case, an edit will involve replacing one entire AST with the other. Thus, a distance of 0 means that two trees are identical, while a distance of 1 means that they are entirely different. When calculating the weight of the edits, ITAP adds each individual edit weight, and calculate the weights of different types as follows:

- **Change**: take the maximum of the old and new value weights
- **Subset**: weight of the new value - weight of the old value
- **Superset**: weight of the old value - weight of the new value
- **Add**: weight of the new value
- **Delete**: weight of the old value
- **Swap**: 2
- **Move**: 1

My goal with these weights is to emphasize how much of a change the student will need to make. Therefore, subset and supersets only count the code that is removed or added, as do add and delete values. Swap and Move vectors cost very little, as they only require that a student move lines of code, rather than write new code. Note: when ITAP calculates the weight of an AST, it is effectively counting the number of tokens that occur in it.

Finally, a quick note on measuring the correctness of code: ITAP relies on test cases to measure whether or not a program is correct. Therefore, certain types of problems (such as graphics problems, or open-ended tasks) cannot be used in path construction. ITAP only allows 0.1 seconds to run test cases on any given piece of code, due to the large number of states which need to be tested during path construction and the high rate of infinite loops. I have tested the difference between cutting off the test cases at 0.1 seconds vs. 1 second, and found very few cases where it changed the score.

Now I can finally describe the actual path construction algorithm. This takes place in four parts: choosing a goal state, optimizing that goal state, finding all valid edit combinations between the starting state and the goal, and choosing the optimal path through those combinations. I describe these parts in this section.

Goal State Generation

The first two steps involve choosing and optimizing the best possible goal for a given state. This goal will be the endpoint for the path construction algorithm, and should be as close to the student's intended goal as possible. Since ITAP cannot read the student's mind, it estimates what they're trying to do based on data instead.

First, ITAP compares the start state (the student's submission in some abstracted form) to each of the correct states it has gathered so far, to identify which state has the smallest distance. During these distance calculations it ignores variable name differences, for reasons described below. When two states tie on distance, it breaks the tie based on which state has been used more often (to encourage more common and therefore more sane solutions).

Once ITAP has found the best-matching goal state, it does function and variable matching between the two states, to ensure that the *semantic meaning* of functions and variables matches as closely as possible. To do this, it identifies all anonymized function and variable names in the two functions, and then generates all possible pairings between function/variable names across the two. This does not include functions/variables with names that have *not* been anonymized, as these names cannot be changed. It also does not include parameters, as they are constrained by position, and it does not allow variable matching across functions. It applies each mapping to the goal state in turn, and identifies which version of the goal state is closest to the starting state. An example of this process is shown in Figure 21. In this example, the goal state has three variables where the start state has only two. By trying all possible mappings between the two original variables and the three possible locations, the algorithm can find that the best mapping changes the first variable in the goal state to be the new one, while the second and third goal variables are mapped to the first and second state variables. This greatly reduces the number of edits needed.

<table>
<tr><th>Variable Map</th><th>Goal State</th></tr>
<tr><td><b>Start State</b></td><td><pre>def all_three_chars(p0):
    for v0 in range(len(p0)):
        for v1 in range(len(p0[v0])):
            if (len(p0[v0][v1]) != 3):
                return True
    return False</pre></td></tr>
<tr><td><b>Original Goal State</b>
v0 -> v1,
v1 -> n0,
add v0 later</td><td><pre>def all_three_chars(p0):
    v0 = 0
    for v1 in range(len(p0)):
        for n0 in range(len(p0[v1])):
            if (len(p0[v1]) != 3):
                return True
    return v0</pre></td></tr>
<tr><td>v0 -> n0,
v1 -> v1
add v0 later</td><td><pre>def all_three_chars(p0):
    v0 = 0
    for n0 in range(len(p0)):
        for v1 in range(len(p0[n0])):
            if (len(p0[n0]) != 3):
                return True
    return v0</pre></td></tr>
<tr><td>v0 -> v0,
v1 -> n0,
add v1 later</td><td><pre>def all_three_chars(p0):
    v1 = 0
    for v0 in range(len(p0)):
        for n0 in range(len(p0[v0])):
            if (len(p0[v0]) != 3):
                return True
    return v1</pre></td></tr>
<tr><td>v0 -> n0,
v1 -> v0,
add v1 later</td><td><pre>def all_three_chars(p0):
    v1 = 0
    for n0 in range(len(p0)):
        for v0 in range(len(p0[n0])):
            if (len(p0[n0]) != 3):
                return True
    return v1</pre></td></tr>
<tr><td><b>Best Goal State</b>
v0 -> v0,
v1 -> v1,
add n0 later</td><td><pre>def all_three_chars(p0):
    n0 = 0
    for v0 in range(len(p0)):
        for v1 in range(len(p0[v0])):
            if (len(p0[v0]) != 3):
                return True
    return n0</pre></td></tr>
<tr><td>v0 -> v1,
v1 -> v0,
add n0 later</td><td><pre>def all_three_chars(p0):
    n0 = 0
    for v1 in range(len(p0)):
        for v0 in range(len(p0[v1])):
            if (len(p0[v1]) != 3):
                return True
    return n0</pre></td></tr>
</table>

Figure 21: An example of how variable matching is used to identify an ideal matching for goal states. In this case, the best goal state (shown second from the bottom) is different from the original goal (shown second from the top).

Next, ITAP seeks to *optimize* the goal state by trimming down the number of edits between the start and goal state as much as it can. This focuses on only those edits which will actually help a student improve their code, instead of showcasing every single difference. ITAP's main approach towards optimization is to apply every possible subset of edits to see if any of them can generate a correct state; however, this approach is time-consuming, as it grows exponentially. Therefore, it expedites the optimization process by attempting fast optimizations first.

First, ITAP checks whether any individual edit vector can correct the starting program when applied on its own. If one or more can, it chooses the smallest edit and chooses the correct state that it generates as its goal state. If none of the individual edits can correct the program, it starts checking all subsets of the main set of edits that include *all but one* of the edits. Here, it is effectively checking whether any single edit is not required, and can be removed without penalty. When it finds an all-but-one set that results in a correct state, it removes the unneeded edit, then repeats the process with the remaining edit set. This process continues until there is no single edit which it can remove.

Once fast-optimization has been performed, ITAP checks how many edits remain in the edit set. If more than six exist, it abandons the optimization and path construction process, as it will take too long to find an optimal path. This is because in the worst case, it will need to test 2^7 = 128 code states, which will take 128*0.1s = 12.8 seconds at worst. I assume that no student wants to wait this long for an optimal hint, so instead ITAP chooses the current goal as the optimal goal, constructs a very simple path (that goes directly from the start state to the goal state with no intermediate stops), and ends the algorithm there. If there are less than seven edits, it performs the real optimization process, generating all possible combinations of edits and testing each one for correctness. If multiple correct states are found, it chooses the one which is closest to the starting state. This gives ITAP a truly optimized goal state which may be close to what the student is trying to do, though it is impossible to tell what a given student's intended solution truly is.

Next-Step Path Construction

Once ITAP has chosen an optimized goal state, it needs to determine how the student should progress from their current start state to that goal state. To do this, it must find all valid edit combinations (which represent intermediate steps between the start state and the goal), then generate a path through these steps; in other words, a path through the solution space.

Finding all valid edit combinations is simple: ITAP generates every possible subset of edits (which has often already been done during optimization), then checks each subset for whether it produces a next-step state which is *valid*. Valid states must obey three properties:

1. A valid state must be able to parse into an AST.
2. A valid state must be closer to the goal than the starting state.
3. A valid state must perform no worse than the starting state when tested.

The first requirement is almost always true (though there are a few exceptions: for example, multi-comparisons where the number of values does not match the number of operators). The second requirement is also almost always true, but the third varies across states. I recognize that there may be some cases where a student must fail more test cases in order to make progress, but unfortunately, students are unlikely to listen to hint messages which make their performance worse. In the worst case, ITAP can always tell students to jump straight to the goal state.

Once it has the set of all valid combinations, ITAP needs to generate a path through the edits. This path will effectively break up the full set of edits into subsets and put them into a certain optimal order. To decide how they'll be broken up and what order to put them in, ITAP looks for the most desirable next-step state out of all possible next states in the set. I define desirable states as having the following properties:

- **Seen Before**: a state that has been submitted by a student before has a better chance of being reasonable than a state entirely concocted by an algorithm.
- **Close to Current State**: when possible, I want the next-step state to require a small amount of change from the previous state
- **Good Performance**: I want the next state to do as well on test cases as possible, to encourage the student by demonstrating that they're making progress.

When weighing these properties, ITAP currently gives double weight to closeness and quadruple weight to having been seen before, to emphasize my priorities. However, I have not tested this formula carefully; it is entirely possible that a different formula would result in better intermediate states. I leave experimentation into this formula for future work.

Once the most desirable next-step state has been identified, ITAP reduces the set of possible combinations to include only those between the next-step state and the goal state. It then repeats the process until, eventually, it reaches the goal; this process should always terminate, as the set of edits to be applied is finite. This process gives ITAP a chain of edits which can be applied, one by one, to reach the goal state.

Reification

Once ITAP has generated a goal state and a path of edits from a starting state, it needs to make sure that that goal and that path are properly contextualized to the student's code. This is especially true when the student's code has been majorly changed during canonicalization, as parts of it might have been moved or modified, and those parts might have ended up in the recommended edits. Therefore, ITAP needs to undo the canonicalizations within the edits, to make them applicable to the student's code. I call this process reification, as it is bringing edits from the abstract solution space to the reality of the student's concrete code.

When reifying code, ITAP needs to update the old values, new values, and paths of the edit vectors, as these are all the components that directly relate to the code they're applied to. In most cases, this can be done simply by tracking global IDs. At the start of the canonicalization process, ITAP annotates each of the nodes of the starting AST with an ID number. During canonicalization, this ID is passed along whenever changes are made so that it always maps back to the semantically equivalent expression in the original AST. Then, during path construction, the old values in edit vectors are extracted from the canonicalized AST, so they retain their global ID. This means that during reification, ITAP needs only search for the old value's global ID in the original AST to identify where the semantically equivalent version of the expression occurred in the original code. This also lets it identify the reified path, as it can calculate it based on that position.

Of course, there are some cases where this does not work quite so easily. For example, some edit vectors change basic values (like strings or integers), which do not retain metadata. In these cases, ITAP can traverse the edit vector's path backwards to find the nearest parent node that has a global id; it can then find that parent's equivalent in the original code, and follow the path to find the old value. Furthermore, while this method works very well at reifying the old values and the paths, it cannot do the same for new values, as these have been taken from the goal states. Therefore, to make sure that the new values are sensible, it must keep track of any canonicalizations that affect the old values, to see if changes will ever need to be made to both.

Specific Canonicalizations

There are several canonicalizations that need to be treated specially during reification, to ensure that edits remain sensible. To keep track of these, ITAP applies metadata to affected AST nodes during canonicalization, which stores which canonicalizations have affected which nodes. In this section, I describe the special reifications that have been implemented.

First, ITAP needs to undo the anonymization of variable names in both old and new values. This is fairly simple; when doing anonymization, it adds metadata to variable nodes that keep track of the original name, and then in reification, it replaces the anonymized name with the original one. The only exception is new variables that have been added from the goal state. ITAP replaces the filler names on these with `new_var_<num>` (where `<num>` keeps track of how many variables have been added), to make it clear to the student that a new variable is being added.

Next, ITAP has to identify augmented assignments that were turned into normal assignments during simplification (`x += 1` to `x = x + 1`). This can cause a problem in reification when edits are made to the part of the code originally tied to the assignment; in this simple example, that would occur if ITAP tried to change the `x` or `+` on the assignment's right side. To address this problem, ITAP applies the non-reified edits to its canonicalized starting point to get the modified AST, then moves upward in both the new tree and the original tree to reach the node where the whole assignment was performed. This allows it to change the whole statement, as is shown in Figure 22. This method of moving up in the tree to reach an unchanged statement or expression will be repeated often in the following examples.

Time	Old Context	New Context	Edit

Pre	`Function:`
	`x = x + 1`
		`Function:`
		`x = y + 1`
			`Change x to y`
			`[Binary Operation Left Side <- Assign Value <- Function Body Line 0]`

Post	`Function:`
	`x += 1`
		`Function:`
		`x = y + 1`
			`Change AugAssign to Assign`
			`[Function Body Line 0]`

Figure 22: An example of augmented assignment pre- and post- reification. Since changing `x` to `y` in the left side of the value breaks the augmented assignment, the whole assignment statement is changed instead.

Another special case from the simplification canonicalizations is multi-comparison operations. When ITAP changes an expression like `(a < b < c)` to `(a < b and b < c)`, it must be careful about any edits that apply to the expression. For example, if an edit wants to add another expression to the Boolean operation, it must create a new Boolean operation altogether. If it wants to delete part of the Boolean operation, it can just remove the unneeded part of the operation. Finally, if it wants to change the interior part of the multi-comparison, it needs to break apart the multi-comparison in the edit to keep from modifying both `b` expressions. Examples of all three of these operations are shown in Figure 23.

Edit Type	Time	Old Context	New Context	Edit
Add	Pre	return (a < b and b < c)	return (a < b and b < c and d)	Add d to Boolean Operation [Boolean Operation Item 2 <- Return Value]
	Post	return (a < b < c)	return (a < b < c and d)	Change (a < b < c) to (a < b < c and d) [Return Value]
Delete	Pre	return (a < b and b < c)	return (a < b)	Remove b < c from Boolean Operation [Boolean Operation Item 1 <- Return Value]
	Post	return (a < b < c)	return (a < b)	Remove < c from Compare [Comparator 2 <- Return Value]
Change	Pre	return (a < b and b < c)	return (a < b and d < c)	Change b to d [Comparator 0 <- Boolean Operation Item 1 <- Return Value]
	Post	return (a < b < c)	return (a < b and d < c)	Change (a < b < c) to (a < b and d < c) [Return Value]

Figure 23: Three examples of reification performed on multi-comparisons. Most of the change happens in the edit, shown on the right.

Most of the compiler optimization functions used in simplification do not need special reification operations, as they only remove code (which therefore will not show up in edits). However, copy propagation is an exception, as it sometimes modifies expressions that were originally represented by variables. In that case, ITAP needs to replace the original variable reference with the whole expression (including the changed component) that it represents. To do this, it tags all children of the copied expression with metadata that includes the variable's global id. This allows it to identify the appropriate location in the original tree where the edit should be made.

ITAP also needs to make sure that proper reification is performed on expressions that have been changed by the ordering constraint. This is especially true of operators, as they might directly change the new value as well as the old. When an operator is reversed or negated, it is given a metadata tag; if that tag is found on the old value, ITAP undoes the reversal and/or negation, and performs the same operation on the new value, so that both values will be reversed/negated. This ensures that operations are represented correctly. It also needs to keep track of reordered binary expressions (as it may want to modify sub-expressions that were not originally grouped together); for these, it moves up in the tree until it reaches an expression with a global id (which therefore must have an equivalent match in the original tree), and performs the operation there. Examples of these reifications are shown in Figure 24.

Function	Time	Old Context	New Context	Edit
Reversed	Pre	`return (a < b)`	`return (a <= b)`	Change < to <= [Compare Op <- Return Value]
	Post	`return (b > a)`	`return (b >= a)`	Change > to >= [Compare Op <- Return Value]
Negation	Pre	`return (a == b)`	`return (a != b)`	Change == to != [Compare Op <- Return Value]
	Post	`return not (a != b)`	`return not (a == b)`	Change != to == [Compare Op <- Unary Value <- Return Value]
Binary Operation	Pre	`return (a + b) + c`	`return (a + d) + c`	Change b to d [Binary Operation Right Value <- Binary Operation Left Value <- Return Value]
	Post	`return a + (b + c)`	`return a + (d + c)`	Change b to d [Binary Operation Left Value <- Binary Operation Right Value <- Return Value]

Figure 24: Example reifications for three simple ordering functions: reversed operators, negated operators, and reordered binary operations.

ITAP also has to address Move and Swap vectors, which can get confused by the reordering of expressions and statements. This most often happens in statement bodies and in Boolean operations, where expressions or statements need to be moved to a different set of items than where they're currently located. For Boolean expressions, it can deal with this by moving up in the tree to encompass the full modification (turning the Move vector into a Change vector); for statement bodies, it deletes the line in its current location and adds it back at a new position. In future work, it would be useful to extend the functionality of Move and Swap vectors to work outside of the constraints of position in the AST.

Finally, ITAP has to deal with the most common cause of complexity in reification: conditionals. Conditionals undergo many potential changes during canonicalization, which leads to many possible problems in reification. Furthermore, it cannot always use the trick of moving up to a higher location in the tree here, as conditional modifications may cover multiple lines in a program. Therefore, ITAP treats each canonicalization distinctly.

First, I address conditional combining, where multiple conditionals are combined into one (using a Boolean operation over multiple tests). This is easiest when ITAP needs to add a new value to the Boolean operation, as it can just add the new value to the first conditional test, and leave the other tests alone. Changing the combined operation is harder; in this case, it can change the first conditional's test to the new value, but it then needs to delete the additional conditionals, so that they do not affect the new functionality. This may increase the number of edit vectors it needs to include. And finally, when it needs to delete part of the Boolean operation, it actually needs to delete *all* of the conditionals that were part of it originally. Examples of these reifications are shown in Figure 25.

Edit Type	Time	Old Context	New Context	Edit

Add	Pre	Function:
		  if (a < b or
		      c < d):
		    return x
			Function:
			  if (a < b or
			      c < d or e):
			    return x
				Add e to Boolean Operation
				[Boolean Operation Item 2
				<- If Test
				<- Function Body Line 0]

	Post	Function:
		  if (a < b):
		    return x
		  if (c < d):
		    return x
			Function:
			  if (a < b or e):
			    return x
			  if (c < d):
			    return x
				Change (a < b) to (a < b or e)
				[If Test <- Function Body Line 0]

Delete	Pre	Function:
		  if (a < b or
		      c < d):
		    return x
			Function:
			  if (c < d):
			    return x
				Remove a < b from Boolean Operation
				[Boolean Operation Item 0
				<- If Test
				<- Function Body Line 0]

	Post	Function:
		  if (a < b):
		    return x
		  if (c < d):
		    return x
			Function:
			  if (c < d):
			    return x
				Remove If statement from Function Body
				[Function Body Line 0]

Change	Pre	Function:
		  if (a < b or
		      c < d):
		    return x
			Function:
			  if (e):
			    return x
				Change (a < b or c < d) to e
				[If Test <- Function Body Line 0]

	Post	Function:
		  if (a < b):
		    return x
		  if (c < d):
		    return x
			Function:
			  if (e):
			    return x
				Change a < b to e
				[If Test <- Function Body Line 0]
				
				Remove If Statement from Function Body
				[Function Body Line 1]

Figure 25: Three examples of conditional combination reifications. The main changes can be seen in the edits, shown on the right.

Next, there are cases where ITAP moved redundant lines outside of conditionals. In the case where an edit wants to delete or change the conditional that the moved line was originally associated with, it needs to make sure that the moved line is preserved. To do this, it tags conditionals whose lines are being moved with metadata that includes the global id of the moved line. If it needs to remove a conditional with that metadata, it instead replaces it with the moved line; if it needs to change it, it keeps the change vector as it is, but adds an Add vector afterwards to put the moved line back in. And finally, it considers the case of conditional collapsing, if the edit includes the test of the conditional. If that test needs to be deleted, it changes the delete vector to instead replace the original conditional with its body, so that the semantics are preserved. Examples of these three cases are shown in Figure 26.

Function	Time	Old Context	New Context	Edit

Redundant Line Delete	Pre	`Function:`
		`  if x:`
		`    x += 1`
		`  return x`
			`Function:`
			`  return x`
				`Remove If Statement from Function Body`
				`[Function Body Line 0]`

	Post	`Function:`
		`  if x:`
		`    x += 1`
		`    return x`
		`  else:`
		`    return x`
			`Function:`
			`  return x`
				`Change If Statement to Return Statement in Function Body`
				`[Function Body Line 0]`

Redundant Line Change	Pre	`Function:`
		`  if x:`
		`    x += 1`
		`  return x`
			`Function:`
			`  print(“foo”)`
			`  return x`
				`Change If Statement to Print Statement in Function Body`
				`[Function Body Line 0]`

	Post	`Function:`
		`  if x:`
		`    x += 1`
		`    return x`
		`  else:`
		`    return x`
			`Function:`
			`  print(“foo”)`
			`  return x`
				`Change If Statement to Print Statement in Function Body`
				`[Function Body Line 0]`
				
				`Add Return Statement to Function Body`
				`[Function Body Line 0]`

Collapsed Conditional Delete	Pre	`Function:`
		`  (x[0] == “a”)`
		`  return x`
			`Function:`
			`  return x`
				`Remove Expression from Function Body`
				`[Function Body Line 0]`

	Post	`Function:`
		`  if x[0] == “a”:`
		`    return x`
		`  else:`
		`    return x`
			`Function:`
			`  return x`
				`Change If Statement to Return Statement in Function Body`
				`[Function Body Line 0]`

Figure 26: Examples of three further conditional reifications: deleting redundant lines, changing redundant lines, and collapsing conditionals. Most of the changes occur in the edits, shown to the right.

Additionally, ITAP does several small optimizations during reification to ensure that the edits will not be nonsensical and will not cause broken code. For example, if a reified edit ends up not causing any change at all, it is removed. ITAP also attempts to simplify edits when possible; instead of telling a student to replace `(x + 1)` with `(x + a)`, it tells them to replace `1` with `a`. Finally, it makes sure that function, conditional, and loop bodies are never deleted to the point of being empty (as this breaks syntax); if this is in danger of occurring, it instead replaces the last line with a `pass` statement.

There are a few canonicalizations that I have not yet implemented reifications for, as they occur rarely; these include function inlining and multi-assignment lines, both of which will eventually require potentially complex reifications. Additionally, it is worth noting that there are some cases where overzealous dead code elimination removes student code which is then *added back in* during the path construction process. I attempt to address this problem during hint generation, but a better solution might be to recognize when an expression is being added that already exists in reification, so that the addition can be ignored completely.

Hint Generation

Now that I have described canonicalization, path construction, and reification, they can all be put together to implement my main goal, hint generation. I will illustrate how this process works with a real student program, to tie the whole system together. I then evaluate the system on its ability to generate hints and on its ability to improve in performance over time.

Example

In this example I will use a real student submission to the problem one_to_n. In this problem, the student is given a number n and needs to generate a string which contains the numbers from 1 to n. The student submission, shown at the top of Figure 27, attempts to solve this problem by generating a list of numbers and then joining them together with the `.join` function; however, this submission generates a runtime error, as `.join` can only be called on a list of strings, not a list of integers. A human teacher would likely suggest that the student change those integers into strings by using `l.append(str(i))` instead of `l.append(i)`.

To generate a hint automatically, ITAP tests the student program to determine whether it is syntactically incorrect, semantically incorrect, or correct. If there is a syntax error, it uses a separate process (described in a later section) to generate text hints. Otherwise, it parses the code into an AST and generates AST-cleaned, anonymized, and canonicalized versions of the code. AST-cleaning is done by transforming the AST back into text using a tree-to-text function (which normalizes whitespace); the other functions have been described already. The AST-cleaned, anonymized, and canonicalized versions of this example program are shown in Figure 27.

Original	`def one_to_n(n):`
	`    l = []`
	`    string = ''`
	`    for i in range(1,n+1):`
	`       l.append(i)`
	
	`    return ''.join(l)`

AST-cleaned	`def one_to_n(n):`
	`    l = []`
	`    string = ''`
	`    for i in range(1, (n + 1)):`
	`        l.append(i)`
	`    return ''.join(l)`

Anonymized	`def one_to_n(p0_one_to_n):`
	`    v0_one_to_n = []`
	`    string = ''`
	`    for v1_one_to_n in range(1, (p0_one_to_n + 1)):`
	`        v0_one_to_n.append(v1_one_to_n)`
	`    return ''.join(v0_one_to_n)`

Canonicalized	`def one_to_n(p0_one_to_n):`
	`    v0_one_to_n = []`
	`    for v1_one_to_n in range(1, (p0_one_to_n + 1)):`
	`        v0_one_to_n.append(v1_one_to_n)`
	`    return ''.join(v0_one_to_n)`

Figure 27: The different versions of the example program, which currently causes a runtime error when it attempts to join numbers together in a string. Note how each version provides some new form of normalization.

If the program is correct, ITAP simply adds these states to the solution space and then returns a statement telling the student they’re already correct. Otherwise, it has to determine what steps they should take next. To do this, it applies path construction to both the anonymized and canonicalized versions of the code. It checks both formats because in some cases the closest solution will be near the anonymized code, while in others it may be near the canonicalized version. The goals chosen from the problem’s solution space for this example’s anonymized and canonicalized versions are shown in Figure 28.

Anonymized Goal	`def one_to_n(p0_one_to_n):`
	`    v0_one_to_n = []`
	`    v1_one_to_n = ''`
	`    for v0_one_to_n in range(1, (p0_one_to_n + 1)):`
	`        v1_one_to_n += str(v0_one_to_n)`
	`    return v1_one_to_n`

Anonymized Edit	`'string' - 'v1_one_to_n' :`
	`[Variable ID <- Assign target <- Function body line 1]`
	
	`'v1_one_to_n' - 'v0_one_to_n' :`
	`[Variable ID <- For loop target <- Function body line 2]`
	
	`v0_one_to_n.append(v1_one_to_n) –`
	`v1_one_to_n += str(v0_one_to_n) :`
	`[For loop body line 0 <- Function body line 2]`
	
	`''.join(v0_one_to_n) - v1_one_to_n :`
	`[Return value <- Function body line 3]`

Canonicalized Goal	`def one_to_n(p0_one_to_n):`
	`    v0_one_to_n = []`
	`    for v1_one_to_n in range(1, (p0_one_to_n + 1)):`
	`        v0_one_to_n += str(v1_one_to_n)`
	`    return ''.join(v0_one_to_n)`

Canonicalized Edit	`v0_one_to_n.append(v1_one_to_n) –`
	`v0_one_to_n += str(v1_one_to_n) :`
	`[For loop body line 0 <- Function body line 1]`

Figure 28: The goals and edit paths chosen for the example’s code states. Note that they are very different, due to `string = ‘’` being removed in the canonicalized version.

Once ITAP has this information, it compares the distances (calculated based on the edits) from the anonymized state to its goal and the canonicalized state to its goal. It chooses the shorter of the two paths as its best path, then applies reification to the edits in that path. In some rare cases, this leads to a dead end (usually due to odd canonicalization cases); if this happens, it repeats the process with the alternative solution path. Figure 29 shows the reified version of the goal and edits.

```
def one_to_n(n):
    l = []
    string = ''
    for i in range(1, (n + 1)):
        l += str(i)
    return ''.join(l)
```

```
l.append(i) - l += str(i)
 : [For loop body line 0 <- Function body line
2]
```

Figure 29: The chosen goal and edits post-reification. These edits can now be directly applied to the starting state.

Hint Representation

Of course, ITAP can’t show the student the hints just by displaying the edit vectors; instead, it has to transform them into textual messages that the student can read. The system does this by mapping the edit vectors onto hint templates which use the components of the vector to give the student information. The basic template for a bottom-out hint is:

[Location info] + [action verb 1] + [old value] + [action verb 2] + [new value] + [context].

In this template, location information and context come from the path, the action verbs come from the vector type, and the old and new values come from the old and new values of the vector. The hint provided for the example problem is shown in Figure 30.

On line 5 in column 8, replace **l.append(i)** with **l += str(i)** in the for loop body.

Figure 30: The hint generated based on the post-reification edits. This is what ITAP can show to the student.

It is worth noting that the hint template has changed throughout ITAP’s development. Early versions of ITAP attempted to provide minimal next-step information to users (as some research suggests that having students engage with smaller solution steps leads to more learning (Roll et al, 2014)). However, based on reactions from users and some findings on how students learn from hints, later versions greatly expanded the amount of content shown. Though the current version of ITAP does not apply most of the original reduction methods, I describe them here to give historical context for previous studies.

Early versions of ITAP reduced the amount of information provided to students by tokenizing the new value of the hint data so that only one new token was provided to students at the time. This was done in an effort to model the next-step hints generated by intelligent tutoring systems, which would target exactly one step of the problem. To tokenize an expression, ITAP would identify the highest node in the new value AST (the node to keep) and would then replace the child nodes of the AST with filler strings. These filler strings originally took the form of `‘[stuff]’` and `‘[more stuff]’`, using a range of string contents in square brackets. In the usability study, these were modified to contain contextual information about the AST; instead of telling a student to change a value to `‘[stuff]’ + ‘[more stuff]’`, ITAP would tell them to change the value to `‘~left side~’ + ‘~right side~’`. Finally, in the final learning evaluation, this code obfuscation was replaced with full hint content (e.g., `sum + count`). Reasoning for these changes is included in later chapters.

Hint provision was also affected by the number of hints available to students. In the first pilot study, I sought to replicate the traditional next-step hints provided in intelligent tutoring systems by providing three levels of hints: point, teach, and bottom-out (VanLehn, 2006). The point hint only told students where the error occurred, the teach hint showed which type of edit needed to be made, and the bottom-out hint showed the new value to be used. An example of these three levels is shown in Figure 31. Students could unlock the more detailed levels of hints by clicking the hint button again if the first hint did not help enough.

`In line 2 consider` **`'powerlevel'`** `in the left side of the comparison`
`In line 2 replace` **`'powerlevel'`** `with something in the left side of the comparison`
`In line 2 replace` **`'powerlevel'`** `with` **`'powerLevel'`** `in the left side of the comparison`

Figure 31: Point, teach, and bottom-out hints generated by the pilot version of ITAP.

Students in the pilot study seemed to find the first and third hint levels useful, but generally skipped past the second level. Therefore, in the first and second classroom studies, ITAP provided only two levels of hints (with and without the new value), as is shown in Figure 32. I also added statements prompting students to ask for hints again if they needed help, as this was not clear to students originally.

`In line 2 change` **`x`** `to something in the left side of the binary operation.`
`If you're still stuck, ask for the next level of hint!`

`In line 2 change` **`x`** `to` **`(x % '[all the stuff]')`** `in the left side of the binary operation`

Figure 32: Point and bottom-out hints generated by initial classroom versions of ITAP.

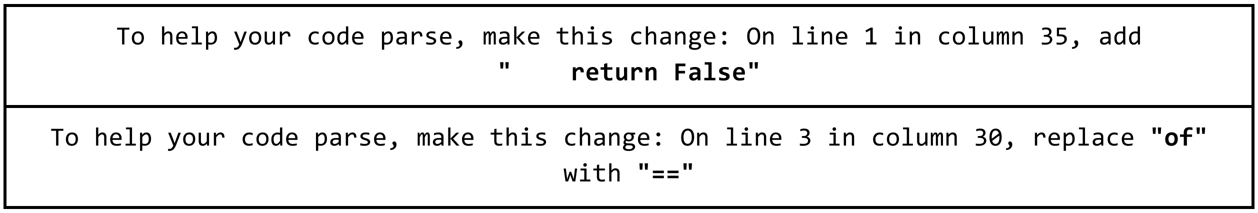
Further feedback from those studies showed that the combination of tokenization and the point-only hints made hints too abstract; therefore, starting with the usability study, ITAP provided only bottom-out hints (as was shown in Figure 30 above). This is the current hint format used by ITAP, though that format may change in the future based on findings from the usability study.

Generating Syntax Hints

The text above describes how to give hints for semantically incorrect programs, but it cannot be used to give hints for programs with syntax errors. This is a major flaw, as the newest students most in need of help are more likely to make syntactic errors in the first place. However, ITAP can still run a simplified version of the path construction algorithm on the text of the program to identify *text edits* that can help a program parse.

In syntax path construction, the goal is not to transform an AST into a version that passes all test cases; instead, ITAP aims to transform the code text into a version that can be parsed. Therefore, in this process, the ‘goal states’ are any states that can parse, and programs are compared via text diffing instead of AST diffing. ITAP computes these comparisons with the Python difflib, which gives it characters that need to be added, removed, or left alone. These characters can be combined into substrings to provide Addition and Deletion edits. ITAP also treats whitespace edits specially (as Indent and Deindent edits), since students have difficulty reading the number of spaces or tabs in a string, so it is easier to report the number of spaces/tabs needed to be added or deleted instead.

The algorithm compares the non-parsing code sample with every code sample that has been seen before. For each state, it then tries all possible subset combinations of edits to see if there is a more minimal approach towards fixing the program. This runs the risk of taking an astronomically long time, so ITAP optimizes by cutting off the search at the length of the best (shortest) edit seen so far. Examples of syntax hints generated by this algorithm are shown in Figure 33.



```
To help your code parse, make this change: On line 1 in column 35, add
"    return False"
```

```
To help your code parse, make this change: On line 3 in column 30, replace "of"
with "=="
```

Figure 33: Two syntax hints generated by ITAP. The first tells a student to add code to an empty block; the second suggests that a non-valid token should be replaced with a valid one.

There are obvious flaws with this approach: the more different a student program is from others seen before, the more nonsensical the suggested edits are likely to be. However, I am working with others on smarter approaches towards generating syntax hints that rely on partial parse trees instead of text (Mudgal, 2016); these approaches are very promising (Sykes & Franek, 2004), and will likely be able to provide syntax hints in most cases. Text-path construction can be saved as a fallback, for when these better approaches do not work.

Evaluation of Hint Chaining

To test the success of ITAP, I want to determine whether it succeeded at the main goal: generating hints that could be followed to produce a correct state. For the technical evaluation I test this literally, by checking whether the reified edits can be directly applied to student code to produce working solutions. Assuming that students can translate from hint messages to edits, this should be the direct equivalent of a student using hints to turn their problem into a worked example, by constantly following bottom-out hints until they reach the correct solution. I call this process *hint-chaining*, as it involves generating hints, applying them to the code, and repeating until the correct solution is reached.

For this evaluation, I again use the set of 41 problems mentioned in the previous section. I perform hint-chaining on all (syntactically or semantically) incorrect states in the dataset in the order that the states were originally submitted, building the solution space over time and measuring the algorithm's performance on two success conditions. First: how often does hint-chaining eventually lead to a correct state? And second: how long does it take for this process to be performed? The first question tests feasibility, while the second tests reasonable use; students are not likely to use hints if they take too long to generate.

I tested 11,051 incorrect states across the 41 problems for this evaluation, and found that all but one followed the hint chain to correct completion. The one state which did not reach a correct state was plagued by a hint-application loop where it would add a new variable assignment, which would then be removed in canonicalization (as it would not semantically impact the rest of the function). Since this edge case is vanishingly rare, I leave fixing it for future work.

To test how long hint generation took, I timed the hint-chaining process for each state, from start to completion. The distribution of resulting times is shown in Figure 34. Note that this figure only shows times for incorrect states; correct states finish in less than five seconds 97.5% of the time, as should be expected. On the other hand, only 77.7% of incorrect states finish in less than five seconds, and 9.4% take more than 10 seconds to finish. This is problematic, as students are not likely to rely on hints if they take too long to generate, even if the delays only happen occasionally.

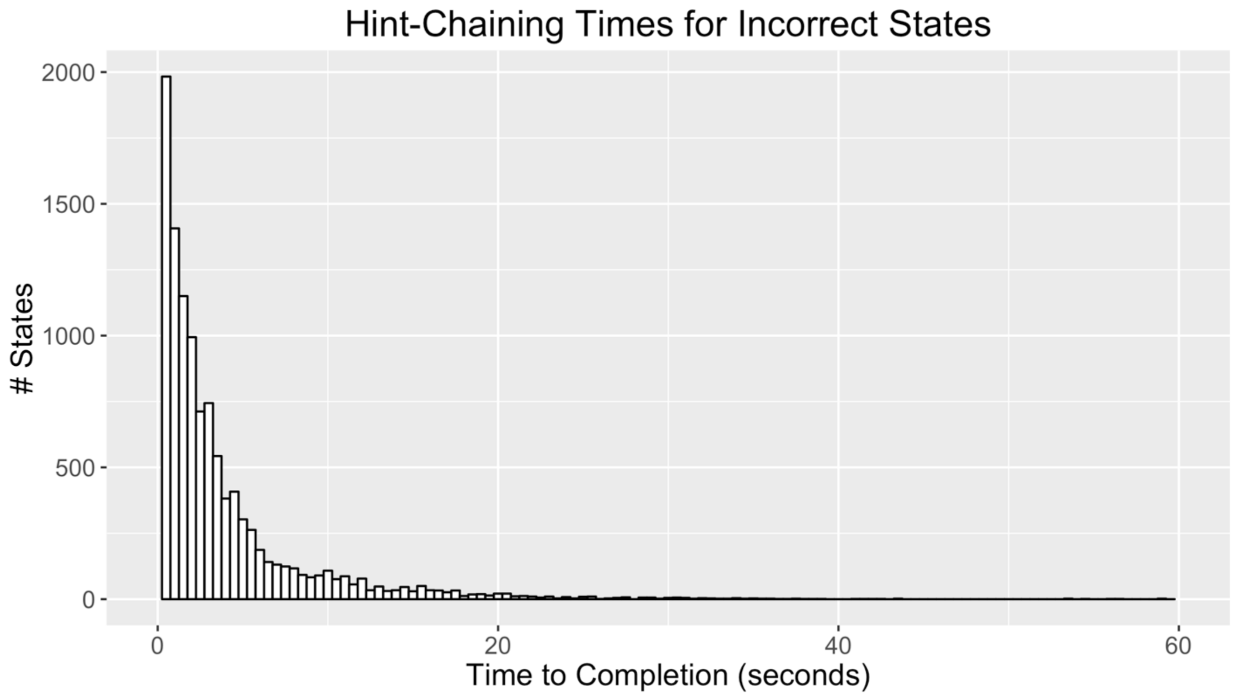


Figure 34: A histogram showing hint-chaining times for incorrect states. 10 states which take longer than 60 seconds are not included.

However, this time analysis was run on full hint-chains, not on individual hint requests (which students actually use). Rerunning the analysis while timing *only* the first request for a hint reveals the time distribution shown in Figure 35. Now, 94.2% of the states are completed in 5 seconds or less, and only 1.1% of the states take longer than 10 seconds. As this is the interaction students will actually see, this is much more acceptable for normal use.

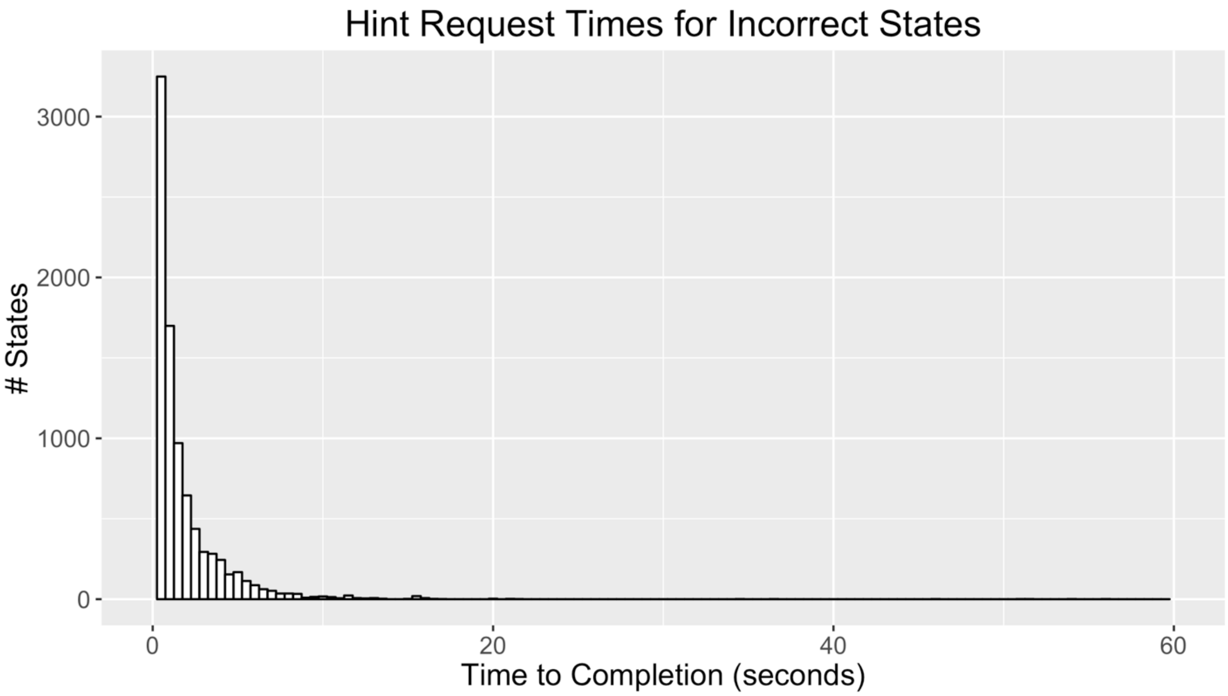


Figure 35: A histogram showing hint generation times for incorrect states. 3 states which take longer than 60 seconds are not included.

But this still leaves one important question: why do some states take so long to perform hint generation? One hypothesis is that this has to do with the complexity of the problem; the more complex a problem is, the longer it may take to generate hints. I represent complexity based on the number of tokens in the teacher's solution to the problem, and find that time for hint generation is, indeed, correlated with complexity ($r = 0.24$ for hint-chaining, $r = 0.08$ for regular hint generation, $p < 0.001$ for both). Altogether, one can expect faster hint generation on simpler programs than more complex ones.

Evaluation of Self-Improvement

In addition to seeing whether ITAP worked at a basic level, I also wanted to examine how ITAP's behavior changed over time as new data was added. Specifically, I want to see whether the hints generated by ITAP truly improve over time as more data is added, and if the hints improve, is there a cutoff point at which more data is not needed? In order to answer this question, I ran a series of technical simulations to measure whether ITAP's performance improved as data was added. These simulations were meant to mimic the cold-start evaluation performed in (Barnes & Stamper, 2008), but by measuring optimization of hints instead of existence of hints.

Previous work has been done with the goal of building self-improving tutoring systems. One of the first approaches attempted to modify instructional approaches experimentally, to some effect (O'Shea, 1978). Other approaches focused on modifying strategies based on student interactions (Dillenbourg, 1989; Soh & Blank, 2008). Still more work has sought to automatically improve or create ITSs based on data-driven methods (Koedinger et al, 2013). I am primarily interested in improving hint effectiveness, and will focus on that aspect instead in my evaluation.

To run this simulation, I drew on the same dataset used in the previous technical evaluations; 41 problems with at least 100 states each. I decided to measure the quality of hints based on the number of tokens included in the edit between the student's starting state and the goal state, as smaller edits are more desirable. First, I generated token edit weights for the baseline and optimal versions of the solution space, to know what the worst and best case scenarios were for each state. In the baseline case, hints are generated for each state with a solution space that consists only of the starter goal state provided by the teacher. In the optimal case, a full solution space based on all data provided by students is generated, then a new hint is generated for each state (resetting the solution space each time). In practice, the hints generated at the baseline are what the very first student using an ITAP tutor would encounter, while the optimal hints would be received by the very last student.

To measure change in hints over time, I ran 20 randomized iterations of solution spaces for each problem, where the order of states was randomized (apart from the starter goal state), then built up a solution space by adding one state at a time. For each state I tracked both how many total states and correct states came before it, as goal states may have a stronger effect than incorrect states on future hints, since incorrect states only impact the choice of edit ordering during path construction. Then I recentered and scaled the edit weights for each state by mapping the range of possible weights for the state (with baseline as max and optimal as min) to the 1-0 range, by calculating (weight - optimal_weight) / (baseline_weight - optimal_weight). This reweighting provides a min-to-max view of how much improvement has been seen for this state, with 0 being optimal. States that do not improve from baseline to optimal (and the few states which have worse performance from baseline to optimal) are excluded, as are goal states (since they do not have edits).

On average across the 41 problems, 56.69% of the states saw normal improvement from baseline to optimal, 39.75% saw no improvement (baseline = optimal), and 3.56% had a baseline score better than optimal (negative improvement). In the twenty random iterations, among the normal-improvement states, 92.60% also saw normal improvement, while 95.49% of the no-improvement states saw no improvement. In other words, this methodology is not perfect (as some states do not fit in the 0-1 measure), but the measurements do make sense for over 90% of cases.

I found that the correlation between the number of states which had been seen before and the adjusted number of edits varied from problem to problem, with a correlation of -0.05 in the worst case (the problem reduce_to_positive) and -0.48 in the best case (the problem first_and_last). The average correlation was -0.24. Surprisingly, the average correlation for number of correct states seen before was no different, as it was within 0.02 points of the correlation for the number of states for each problem. Altogether, there does seem to be an effect of additional data on hint generation, where more states lead to fewer token edits required. This effect is demonstrated in Figure 36, which shows the solution space improvement for how_many_egg_cartons, an average problem.

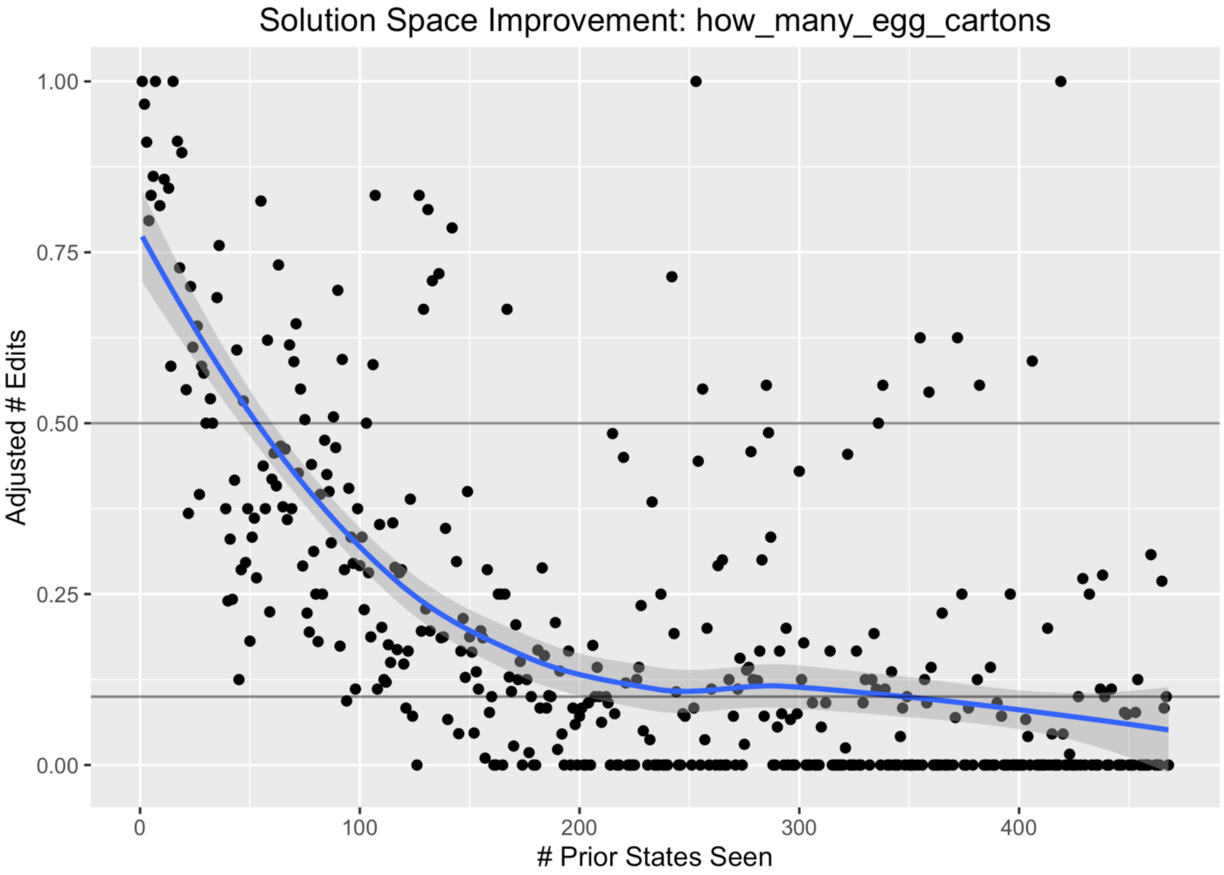


Figure 36: Solution space improvement for how_many_egg_cartons, an average problem in the dataset. 44 outliers (points with adjusted edits less than 0 or greater than 1) have been removed for clarity, with 424 points remaining. As the number of states increases, the number of adjusted edits does too, though the effect levels off after a while. A loess smoothed trendline is included.

Next, I determine whether there is a cutoff point after which I can say the solution space has stopped improving. I check two cutoff points: a soft cutoff (when 50% of the states have reached an adjusted edit of 0), and a hard cutoff (when 90% of the states have reached an adjusted edit of 0). The soft cutoff indicates when the system is optimal more often than not; the hard cutoff indicates when the solution space has reached reasonable optimality.

In the analysis, I found starkly different results for different problems. 19 of the problems reached the soft cutoff immediately (within the first ten states); they started with a fairly optimal solution space already. In contrast, 9 problems did not reach the soft cutoff until seeing at least half of the states, 8 did not reach it until seeing 90% of the states, and 5 did not reach that point at all. The hard cutoff also gave varying results; 7 problems reached the hard cutoff at once, 9 reached it within 90% of the states, and 25 did not reach it at all. The soft and hard cutoffs did not always match up; an example of several problems with varied cutoffs is shown in Figure 37.

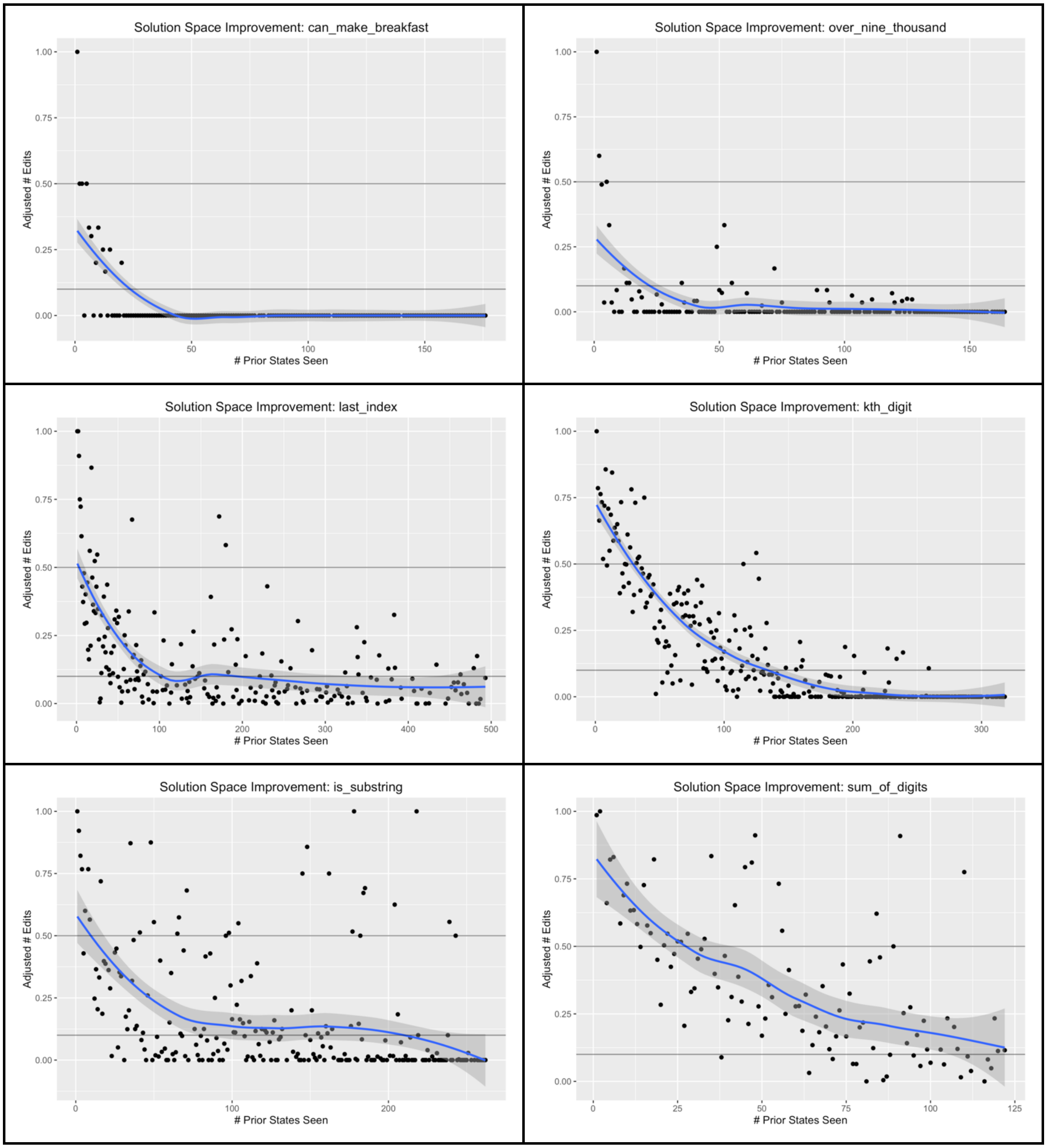


Figure 37: Solution space improvement figures for multiple problems, with lines delinieating the 50% and 90% cutoff points. Upper left: a problem with immediate soft and hard cutoffs. Upper right: a problem with an immediate soft cutoff and a hard cutoff after 111 states (68%). Middle left: a problem with an immediate soft cutoff but no point at which a hard cutoff is reached. Middle right: a problem with a soft cutoff at 27 states (8%) and a hard cutoff at 193 states (61%). Lower left: a problem with a soft cutoff at 69 states (26%) but no point at which a hard cutoff is reached. Lower right: a problem where there is no point at which the soft cutoff is reached.

It is possible that the variation in optimality cutoffs is due to the size of the solution space for the problem, or the complexity of the problem. To determine if this was the case, I checked whether the number of states needed to reach the 50% and 90% cutoff points was correlated with the problem's solution space size or token complexity. I found that the solution space size was strongly correlated (r = 0.50 for 50% cutoff and r = 0.77 for 90% cutoff); in other words, gathering more data leads to more possible routes in the solution space. There is a similarly strong correlation between the number of unique correct solutions (at the canonicalized level) and the cutoff points (r = 0.44 for the 50% cutoff and r = 0.51 for the 90% cutoff), so some of this might be due to new solutions being discovered over time. The problem token complexity was also moderately correlated (r = 0.40 for 50% cutoff and r = 0.30 for 90% cutoff), so longer or more complex problems require more data to fully explore the solution space.

Both of these results imply that solution space improvement will vary widely between different problems, and likely across different datasets and different populations; however, I can still say that these improvement cutoffs do exist, and can be identified in individual problems. This supports the idea that ITAP actually is acting as a self-improving tutoring system. This does not necessarily mean that ITAP-generated hints grow to resemble the hints that human tutors would provide; to measure that, I will need to run studies in the future comparing human-generated hints to ITAP-generated hints. I leave this for future work for now.

3. Identifying Student Help-Seeking in Programming Problems

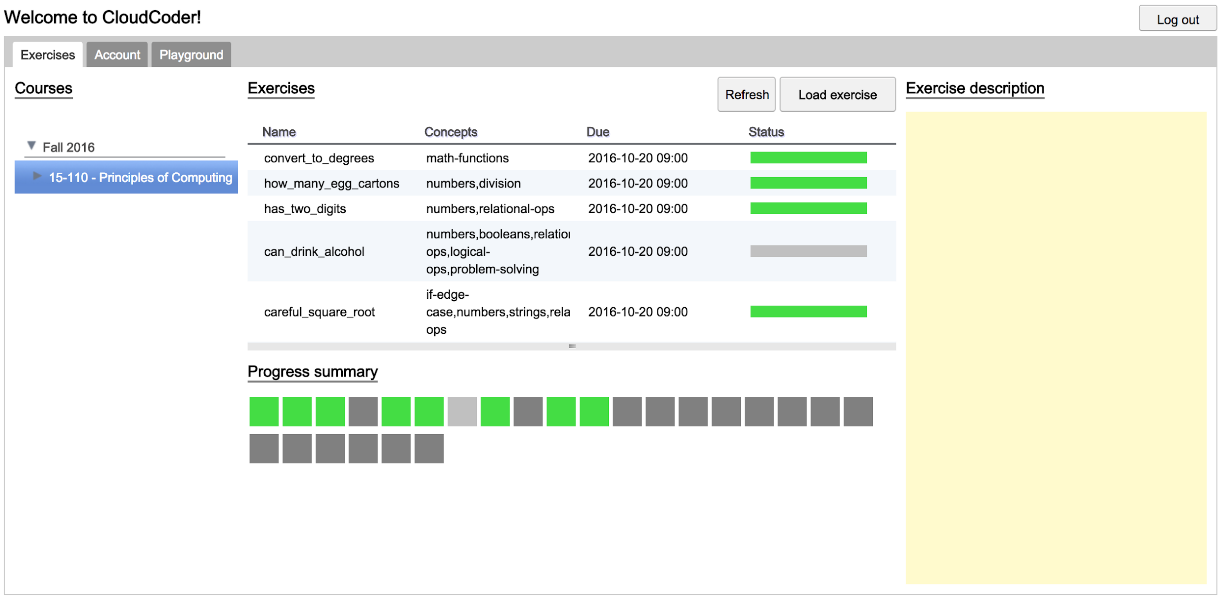
Technical evaluations can go far in describing the capabilities and limits of ITAP, but they cannot demonstrate whether the generated hints will affect student behavior and/or learning. To measure learning, I must test ITAP with real programming students to see how it affects their learning and to get feedback from these students on where the system works well and where it can be improved. Towards this end, I ran a small pilot study and two wider-scale classroom studies with the primary goal of identifying how students view practice, help-seeking, and problem-solving in the context of programming education. In this chapter I discuss these three studies and their results.

First, I briefly report on related work into student help-seeking behaviors during learning. In a general review of help-seeking studies, it was reported that student help use is influenced by student characteristics and is often flawed, but that on-demand help can lead to better learning (Aleven et al, 2003). Student use of more detailed help seems to be associated with shallower learning (Mathews & Mitrovic, 2008), though requesting help on more challenging steps is associated with more productive learning (Roll et al, 2014). Improper help use in ITSs is often represented as help abuse (asking for help without putting forth effort) and help avoidance (refusing to use help sources at all) (Aleven et al, 2016), and can lead to worse learning in some cases. Less investigation has been done into the specific help sources used by introductory programming students, though some studies have reported that students ask different questions in lecture vs. on email newsletters (Postner & Stevens, 2005), and deeper analysis into student questions during office hours shows that they can often be repetitive (Heiner, 2008).

Online IDE Implementation

To make the ITAP generated hints available to students, I needed to insert ITAP into an integrated development environment (IDE) that students could use while solving problems. Online IDEs typically manage test cases, scoring, and the code editor/syntax highlighting. Having a system that already supports these necessary features makes it easier to run studies, as less development needs to be done.

For the following three studies, I used the online IDE Cloudcoder (Papancea, Spacco, & Hovemeyer, 2013) to present practice problems to students. I chose Cloudcoder because it allowed teacher tracking of student accounts (which would let me review participant activity), and because it is open-source, which meant that I could easily create an edited version of the IDE that supported hint generation. After a student has logged in, Cloudcoder shows them a list of problems that have been released in their class (see Figure 38); clicking on the problem leads them to the editor, where they can write code and click ‘Submit’ to test their program (see Figure 39). I modified this system to also include buttons, which, when pressed, would send a hint request to the ITAP system running on a separate server via a PHP request.



Welcome to CloudCoder!

Courses

▼ Fall 2016

▶ 15-110 - Principles of Computing

Exercises

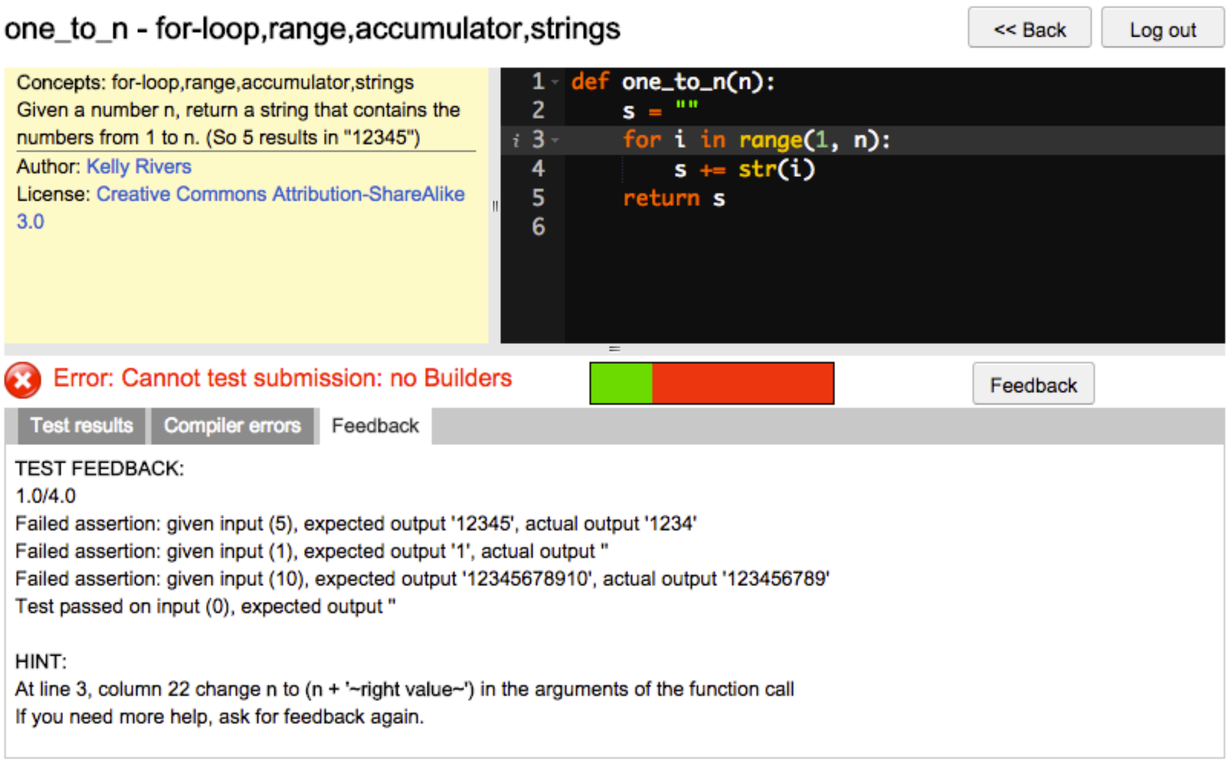
Name	Concepts	Due	Status
convert_to_degrees	math-functions	2016-10-20 09:00	🟩
how_many_egg_cartons	numbers,division	2016-10-20 09:00	🟩
has_two_digits	numbers,relational-ops	2016-10-20 09:00	🟩
can_drink_alcohol	numbers,booleans,relatioı ops,logical-ops,problem-solving	2016-10-20 09:00	⬜
careful_square_root	if-edge-case,numbers,strings,rela ops	2016-10-20 09:00	🟩

Progress summary

🟩🟩🟩⬛🟩🟩⬜🟩⬛🟩🟩⬛⬛⬛⬛⬛⬛⬛⬛
⬛⬛⬛⬛⬛⬛

Exercise description

Figure 38: Cloudcoder's problem selection page. Cloudcoder serves as the 'outer loop' of the tutoring system, where students choose which problem to solve.



one_to_n - for-loop,range,accumulator,strings

Concepts: for-loop,range,accumulator,strings
Given a number n, return a string that contains the numbers from 1 to n. (So 5 results in "12345")

Author: Kelly Rivers
License: Creative Commons Attribution-ShareAlike 3.0

```
1  def one_to_n(n):
2      s = ""
3      for i in range(1, n):
4          s += str(i)
5      return s
6
```

Error: Cannot test submission: no Builders

TEST FEEDBACK:
1.0/4.0
Failed assertion: given input (5), expected output '12345', actual output '1234'
Failed assertion: given input (1), expected output '1', actual output ''
Failed assertion: given input (10), expected output '12345678910', actual output '123456789'
Test passed on input (0), expected output ''

HINT:
At line 3, column 22 change n to (n + '~right value~') in the arguments of the function call
If you need more help, ask for feedback again.

Figure 39: Cloudcoder's problem solving page. Here, students can write code in the editor and get test case results by pressing 'Feedback'. In this case, pressing 'Feedback' also provides the student with a hint.

For each study, I developed a set of practice problems that students would solve. These problems varied across the studies, as I refined the content each semester based on student feedback from the previous semesters. However, some of the problems were reused across multiple semesters. In all cases, problems were designed to be well-aligned with the content being taught in the associated class, so that students would achieve optimal benefit from completing the problems.

Pilot Study

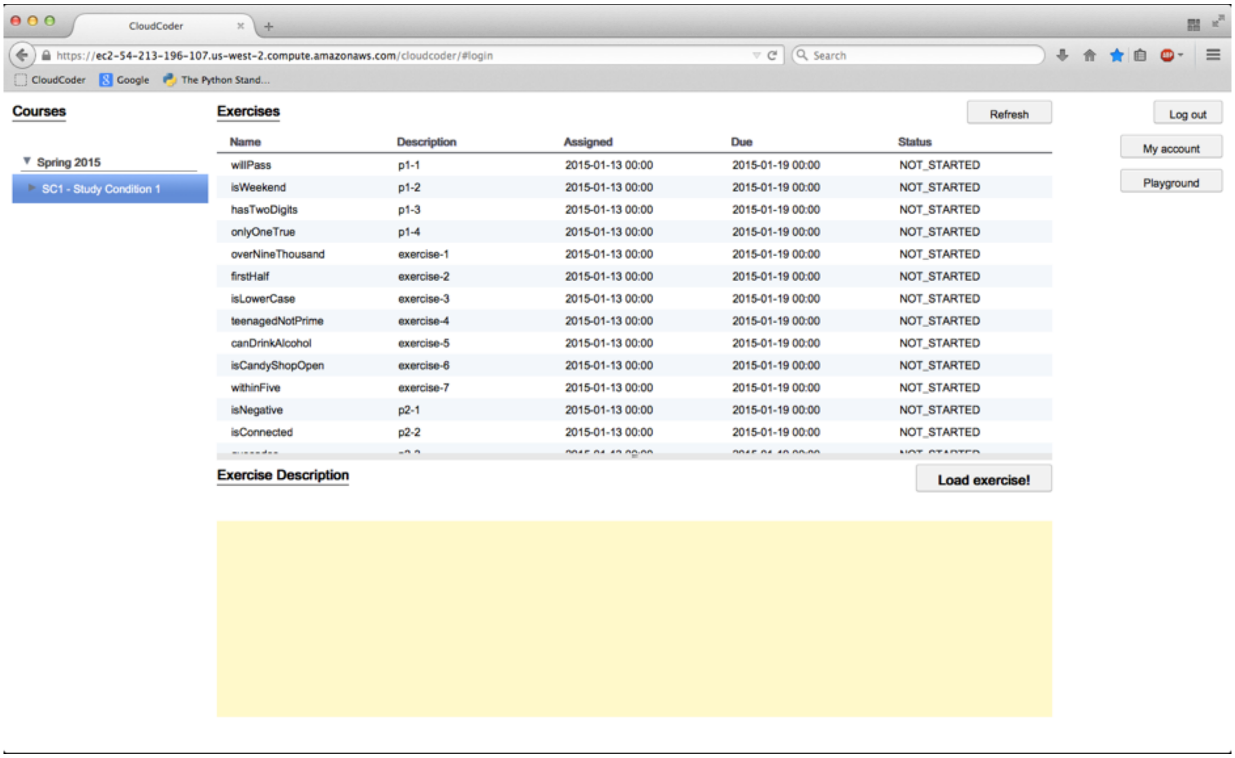
First, in January 2015, I ran a pilot study in order to observe how students solved programming problems, either with or without hints. I originally intended to use this study as an experiment to measure the effects of hints on learning and performance in solving programming problems; however, low participant turnout made it impossible to measure any effects. Therefore, I instead used this study as an opportunity to gather qualitative information on student interaction with programming problems.

Research Questions

The original research question of this study was: does having access to ITAP-generated hints effect student learning or time-on-task during practice? However, I also wanted to answer a more qualitative question: how do real students seek help during practice, and how would they use ITAP-generated hints?

Methods

For this study, I had participants come to the lab for an hour to work on programming practice problems online while I observed, followed by a brief optional interview about their help-seeking practices. The programming activity consisted of 15 problems: four pretest problems, seven practice problems, and four posttest problems (shown in Figure 40). These problems are included in Appendix 1. The study was designed to last one hour, with participants spending up to 15 minutes on the pretest, up to 30 minutes on the practice, and up to 15 minutes on the posttest (with students progressing to the next set if they completed all of a section’s problems before time ran out).



Name	Description	Assigned	Due	Status
willPass	p1-1	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
isWeekend	p1-2	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
hasTwoDigits	p1-3	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
onlyOneTrue	p1-4	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
overNineThousand	exercise-1	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
firstHalf	exercise-2	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
isLowerCase	exercise-3	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
teenagedNotPrime	exercise-4	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
canDrinkAlcohol	exercise-5	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
isCandyShopOpen	exercise-6	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
withinFive	exercise-7	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
isNegative	p2-1	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED
isConnected	p2-2	2015-01-13 00:00	2015-01-19 00:00	NOT_STARTED

Figure 40: The pilot study interface, as seen by students.

At the start of the study, participants were randomly assigned to either the Control or Hint condition, where the Hint condition had a hint button on the seven practice problems, while the Control condition only had the normal Submit button. Students were also assigned to different orderings of two different tests in order to counterbalance the questions, which led to four distinct groups: Control-Test 1-First, Hint-Test 1-First, Control-Test 2-First, and Hint-Test 2-First.

After each participant had finished the practice problems, I asked them if they would be willing to do a short interview about their programming problem-solving process. This interview consisted of the following questions:

1. Is there any problem-solving that you do before you start coding? Please describe it.
2. Have you been stuck on a programming problem while coding before? If so, what have you done to get help?
3. Which of the problems that you worked on today do you think was the most difficult for you? Why?
4. [If they were able to ask for hints]: Were the hints you received helpful? Why or why not?
5. [If they were able to ask for hints]: Was there ever a time that a hint you received didn't align with your idea of the solution? Was this helpful or unhelpful?
6. Do you have any other thoughts you'd like to share?

Participants were recruited from one of the introductory computer science courses at Carnegie Mellon, 15-112, via an announcement during class lecture in the first week of classes and an email sent to the course's dlist. Students were told that the study was optional and would not directly affect their course grade, and that participants would be entered into a raffle, where two participants would win $20 Amazon gift cards.

Altogether, fifteen students (out of around four hundred) volunteered to participate in the study during the first week of the semester. After this point, the students had advanced far enough in the material that the practice problems were no longer a challenge, so no more students could be recruited. These students attempted 13.47/15 problems on average, and solved 10.93/15 on average, with a total of 202 attempts.

Results

First, I examined how students used hints. The 8/15 students in the hint condition had access to hints during the practice problems, and all requested at least one hint. In total, these students requested 46 hints, 21 of which were unique (as some students requested the same hint multiple times), requesting 5.75 hints on average. I found no correlation between the number of hints a student requested and the time they spent on the practice (possibly because time was controlled) or the number of incorrect submissions they made, but I did find a strong negative correlation between the number of unique hints requested and the number of problems a student solved at pretest ($r = -0.56$). In other words, weaker students were more likely to ask for hints.

Next, I checked the performance of students in the two conditions to see if any differences could be noted between the two despite the small sample size. The results are shown in Figure 41. According to a repeated measures ANOVA there was only a marginal effect of test time $F(1, 13) = 4.03$, $p < 0.1$, and no significant effect of condition $F(1, 13) = 1.36$, $p > 0.1$ nor the interaction of condition and test time $F(1, 13) = 0.81$, $p > 0.1$. This lack of condition effect is potentially caused by students being at mastery already at the pretest; 9/15 of the students completed all four pretest problems within the given time limit. Therefore, I chose to focus on the qualitative analysis instead.

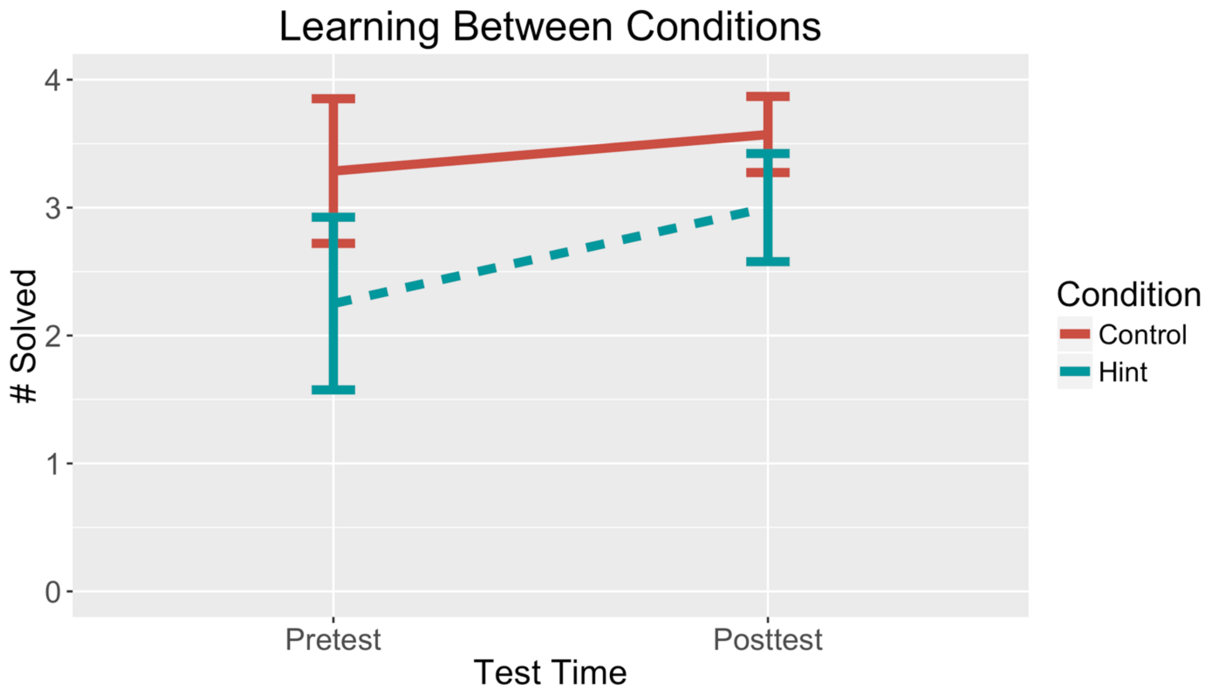


Figure 41: Learning between conditions in the pilot study. There was no significant difference between the two conditions, potentially because the students started at a high level of performance.

Looking at the results of the students reveals three main groups of students: those who have already mastered the material (the six who got all fifteen problems right), those who are successfully learning (the six who got at least five problems right across the activity), and those who are failing to learn (the three who got fewer than five problems right across the activity). In reviewing the activities of these students, there might be indicators for why they perform so differently.

Qualitative Analysis by Performance

First, I analyze the students who had already mastered the material, despite only having been in class for a week. All six of these students completed all fifteen problems within the time limit, and several of them finished with extra time to spare. Four of the six students reported having taken a programming class previously, so it is not particularly surprising that they found the first week's material easy. Most of the students used online resources (like StackOverflow and testing code with repl.it) to find information they didn't already know. They would sometimes encounter bugs (usually related to edge cases), but they could fix them quickly after looking at compiler messages and test case results.

These students also used concrete strategies when approaching each problem. One student reported in the interview that he would read through the problem and try to match the text to the code; another reported that she 'bolded parts of the prompt in her head', to identify requirements. One student said he looked at the parameters and variables to see what he'd need for the end. Altogether, it seemed as if these students used more purposeful strategies when designing solutions for the problems.

Next, I analyze the students who were successfully learning the material. Of the six students, three got perfect scores on pretest and posttest but could not complete all practice problems, while the remaining three either improved or remained static in problem completion from pretest to posttest. Two of the six students had taken a programming class before.

First, looking at the three students who achieved perfect pretest scores shows that they all mainly struggled with the string problems, which were universally difficult for participants since they had not yet learned about strings in class. These problems all required string comparison, which most participants did not know was built into Python. The three students who did not complete the string problems all got stuck developing different (incorrect) algorithms, which led them down unproductive paths. These three students did not report having clear strategies for problem solving in the interview, which might tie into why they had more trouble with strings.

Next, there are three students who struggled with more than just the string problems. All three of these students appeared to have difficulty debugging syntactic and algorithmic errors in their programs; they would make the same mistakes multiple times, and would not immediately understand why their program was wrong. However, they each eventually learned from their mistakes.

One student commonly used `continent == “Australia” or “Antarctica”` in early versions of his code, instead of `continent == “Australia” or continent == “Antarctica”`, but by the posttest, he was able to identify and fix this kind of bug rapidly. Another student commonly used `=` instead of `==` in early programs, but she was able to identify and fix this bug by the time she reached the posttest as well. The final student originally used `“True”` and `“False”` instead of `True` and `False`, but once he realized that the values were built-in, he was able to solve the remaining problems successfully.

Altogether, these students seemed the best representatives of the learning process: they were not masters at the beginning, but they improved over time. When asked about problem-solving strategies in the interview, one student mentioned that he tended to use a lot of functions, while another student mentioned how she tried to think of multiple cases; in general, their plans did not seem as abstracted as the plans of the more successful students.

Finally, I analyze the students who were still struggling at the end. Two of these three students only solved three problems, while the remaining student only solved two. All of these students attempted far more problems than they were able to solve, often skipping to another problem when they couldn’t solve their current problem. None of the students had taken a programming class before, though two of them had previous experience with different languages.

All three of these students struggled greatly with syntax errors throughout the practice session. Some of them were clearly familiar with sophisticated programming concepts (as they attempted to use if statements), but they did not know how to use those concepts in Python. These students spent a great deal of time reading resources online (such as the Python API, StackOverflow, and the course notes), but they were still unable to successfully complete most of the problems (as they would often abandon a problem after it became clear that they could not complete it).

Qualitative Analysis of Hint Use

Outside of these three groups, I can also look at the interactions between students and hints. This analysis led to a few common observations. First, there were several occasions where students asked for hints and got hints that were not particularly helpful, either due to being too vague or due to the hint referencing a part of the code that was not relevant to the problem. Second, reactions to hints that actually were accurate were varied. One student understood the hint immediately and fixed their code accordingly, while another was confused by a hint that clearly stated what needed to be changed. Therefore, having accurate content does not necessarily imply that a hint will make sense to a student; context matters as well. Also, students rarely got much out of the second-level hint; most students wanted to get to the bottom-out hint (which would tell them what to do).

These are all unfortunate observations for the hint system at that time, but they do not necessarily imply that the system is useless. Two of the eight students who had access to the hint button attempted to use it during the posttest (when it was not available), and were disappointed to discover it was not there. Additionally, two of the three students in the third group tried to use the hint button when having syntax problems; however, syntax hints were not available at the time.

Discussion

Overall, though I was unable to distinguish learning effects from this pilot study, I made several valuable observations. First, it is important for students to have access to both semantic *and* syntactic assistance, as different students struggled with these two types of errors. In particular, the students who learned the least were those who struggled with syntax, so it is essential to provide assistance in that area. Second, students already make use of many different help sources when trying to solve a problem, and many of those sources (such as StackOverflow and the Python API) are used to find examples of working code. Therefore, providing targeted example code may be beneficial for student learning. This theme was explored in more depth in work described in further chapters. Finally, it is important that the hints provided to students are well-targeted and suited to the context, as confusing hints lead to befuddled students.

Classroom Study 0: How Do Hints Affect Learning?

After conducting the pilot study and updating ITAP based on the observations made (including adding syntax hints as a feature), I ran a large-scale classroom study in Fall 2015 to evaluate the impact of hints in a more authentic learning environment. The goal of this study was to measure the effect of hints on learning by having students work on optional practice problems during the course, in the ordinary environments where they would work on assignments.

Research Questions

In this study, my main research question centered around the effect of hints on learning. Specifically, I aimed to see whether having access to hints during practice improved student learning in course assignments or quizzes.

Methods

For this experiment, I studied students enrolled in 15-112, one of the introductory programming courses at Carnegie Mellon, as participants. Students were given login credentials to Cloudcoder, where they could complete practice programming problems if they chose to participate. New problems were released each week for six weeks to match the material being taught in class, with 45 problems released altogether (problems are included in Appendix 1). I gathered log data on student interactions with the website, as well as data from the class on student performance on quizzes and assignments.

At the start of the study, all students were randomly assigned to either the hints-first or hints-second condition, where students in the hints-first condition had access to the Hint button in weeks 1-3, while students in the hints-second condition had access to the Hint button in weeks 4-6. All students had access to the Hint button during weeks 7+.

15-112 students were all opted into the study at the beginning of the semester (though students could opt-out of data collection by logging into Cloudcoder and making the request there), but use of Cloudcoder was entirely voluntary and did not directly impact students' grades. Students were made aware of the study via an announcement during the first week of classes and via emails sent with login information. Of the 419 students in the class, 99 logged into Cloudcoder at some point during the semester, and 63 started at least one programming problem. Nine of the logged-in students (including five who started at least one programming problem) requested that I not use their data for research purposes, so they are not included in this analysis, resulting in 90 students who logged in and 58 who started at least one programming problem. 17 of these students requested at least one hint, and this subset of students requested 112 hints total (6.58 hints per student on average).

In this and all following studies, I run regression analyses on varying sets of factors. In all regression results reported, I use step regression to search through the set of possible factor configurations to identify the optimal set of factors according to AIC. I report on all of those chosen factors, regardless of whether they are significant or not.

Learning Metrics

To measure learning, I had to select appropriate learning events from the semester. Quizzes and assignments both happened weekly, but the first two assignment scores were both near the maximum possible grade (mean = 95.9/100 and 98.4/100), which makes it difficult to measure pretest differences between students. The first two quiz scores offered more variance (mean = 84.9/100 and 90.9/100), making them better indicators for starting ability, so I use the quiz scores to estimate learning over time.

Observing how students used the hint system during the semester made it clear that by far the most practice was done within the first two weeks of the semester, when the system was still new. 47 of the 58 students I analyze here opened a problem within the first three weeks, while only 9 students opened a problem in the following three weeks (when the hints-second condition had hints available). Therefore, I focus this analysis on the first three weeks of the semester, where there is enough student interaction to achieve some power for analysis. During this time period, there were four quizzes I can use to measure learning. Quiz 1 occurred before the practice system was released to students, and Quiz 4 occurred directly after the conditions switched (when usage was very low), so I use these quizzes as pretest and posttest.

Results

Since so many students chose not to use Cloudcoder, I can compare several different subsets of students. In this analysis, I focus primarily on a few different groups. First, I look at *intention to practice:* students who did not choose to use the practice system, students who intended to use the practice system (by logging into Cloudcoder), and students who actually used the practice system (by writing code for at least one problem). Second, I look at the *effect of hints*: students who practiced and were in the hints-first condition vs. students who practiced and were in the hints-second condition, including information on whether or not these students actually used hints.

Analysis of Hint Conditions

First, I answer my initial research question: did having access to hints improve students' learning over time? The pretest and posttest are not balanced (as they are testing different content entirely), but they can still show how different subsets of students differ in performance when compared to each other. Removing students who did not take both Quiz 1 and Quiz 4 leaves 373 students who scored an average of 85.78/100 on Quiz 1 and an average of 79.57/100 on Quiz 4. According to a repeated measures ANOVA, there is no significant effect of condition $F(1,371) = 0.86$, $p > 0.1$ nor interaction between test time and condition $F(1,371) = 0.02$, $p > 0.1$, which is not surprising, as a majority of the students chose not to use Cloudcoder and therefore would not be impacted by the hints.

Investigating only those students who attempted at least one problem in Cloudcoder leaves only 35 students across both conditions (14 in hints-first, 21 in hints-second), but the numbers here are trending in a positive direction; students in the hints-first condition did not see a decrease in scores going from Quiz 1 to Quiz 4 while students in the hints-second condition saw the same decrease as the normal population. This difference is shown in Figure 42. A repeated measures ANOVA showed no significant effect of condition F(1,33) = 0.18, p > 0.1 nor interaction between test time and condition F(1,33) = 1.91, p > 0.1, potentially due to the small sample size. Digging further into the data, I investigate whether this effect may be due to the students in that population who actually chose to use hints. Half of the population (seven students) requested a hint at least once, and indeed, those students saw a positive learning trend (79 to 80.1) whereas the students who did not request hints saw a negative trend (84.1 to 81.8). Again, according to a repeated measures ANOVA, there is no significant effect for condition F(1,12) = 0.45, p > 0.1, nor interaction between condition and test time F(1,12) = 0.25, p > 0.1.

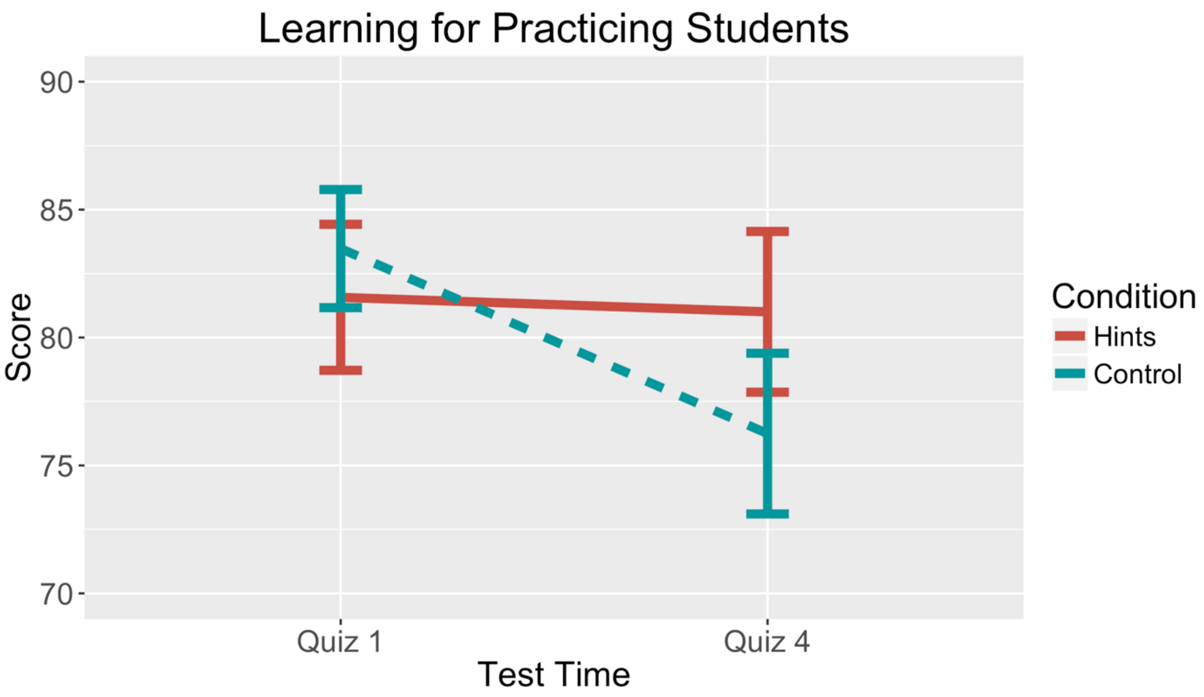


Figure 42: Performance on pre- and post-tests for students who attempted problems in Cloudcoder across conditions. Though students who had access to hints seem to be doing better, the effect is not significant.

I also investigated whether the different conditions affected other elements of students' performance. For this analysis, I narrowed the data set to only include students who started at least one problem (opening it in Cloudcoder), as students did not know what condition they were in until that point, and therefore their performance should not have been affected until then. 43 students met this criterion. Note that this number is greater than the number of students who *attempted* problems, because eight students opened at least one problem but never clicked the 'Submit' button to check their code. Surprisingly, the majority of the students who started but submitted no code were in the hints-first condition (see Table 3). In fact, the difference in problem dropout between conditions is significant based on a Fisher Exact Test ($p < 0.05$). Further investigation into these students showed that three of the seven students in the hints-first condition who made no attempts actually requested hints before exiting the system, though none of them spent more than a minute in the interface.

	Attempted Problem	Dropped Problem	Total
Hints-first	14	7	21
Hints-second	21	1	22
Total	35	8	43

Table 3: Students who, upon opening a problem in Cloudcoder, either chose to write code (attempt the problem) or not attempt it at all. Significantly more students in hints-first did not attempt problems than in hints-second ($p < 0.05$).

I also investigated whether the hint conditions or hint use affected how much time students spent working on problems, but a linear model found that only the number of problems started and the score on Quiz 1 affected how much time students spent on problems. Condition did not have a significant effect (see Table 4). Finally, I investigated whether student use of hints was impacted by pretest score, and found that it was not a significant predictor.

	Estimate	Std. Error	t value	Probability
(Intercept)	**8416.01**	**3174.76**	**2.651**	**0.0116**
Condition: hints-second	-588.75	794.82	-0.741	0.4634
# Problems Started	**557.07**	**58.99**	**9.444**	**< 0.0001**
# Hints Requested	119.61	77.71	1.539	0.1320
Quiz 1 Points	**-111.95**	**37.65**	**-2.973**	**0.0051**

Table 4: A linear model predicting time spent in Cloudcoder. Original factors: condition, # problems started, # hints requested, and Quiz 1 score. Hint condition does not appear to have a significant effect. Adjusted R-squared: 0.7535

Analysis of Cloudcoder Use

Next, I investigate whether there is are any learning differences between students who chose to use Cloudcoder and students who did not. Running a logistic model on students who logged into Cloudcoder shows that scores on Quiz 1 have a significant negative effect on logging into Cloudcoder ($p < 0.05$), as is shown in Table 5. In other words, students who did worse on Quiz 1 were more likely to try Cloudcoder. However, the same significant effect is *not* seen in logistic models for students who start a problem ($p > 0.1$), nor for students who attempt problems (p > 0.1), once I control for logging in. In fact, according to a repeated measures ANOVA, logging in has a marginally significant effect on score F(1,371) = 3.34, p < 0.1, as is shown in Table 6. This effect is demonstrated in Figure 43.

	Estimate	Std. Error	t value	Probability
(Intercept)	0.8432	0.8790	0.959	0.3374
Quiz 1 Score	**-0.0258**	**0.0104**	**-2.489**	**0.0128**

Table 5: A logistic model predicting whether students logged into Cloudcoder. Original factor: Quiz 1 score. Students who performed worse on Quiz 1 were more likely to log in.

	Mean Sq	NumDF	F.value	Probability
Test Time	**3778.2**	**1**	**38.979**	**< 0.0001**
Logged In	*323.4*	*1*	*3.337*	*0.0686*
Interaction	152.2	1	1.570	0.2110

Table 6: A repeated measures ANOVA demonstrating the effect of logging in on test scores. Students who logged in did marginally significantly better over time on quizzes (p < 0.1).

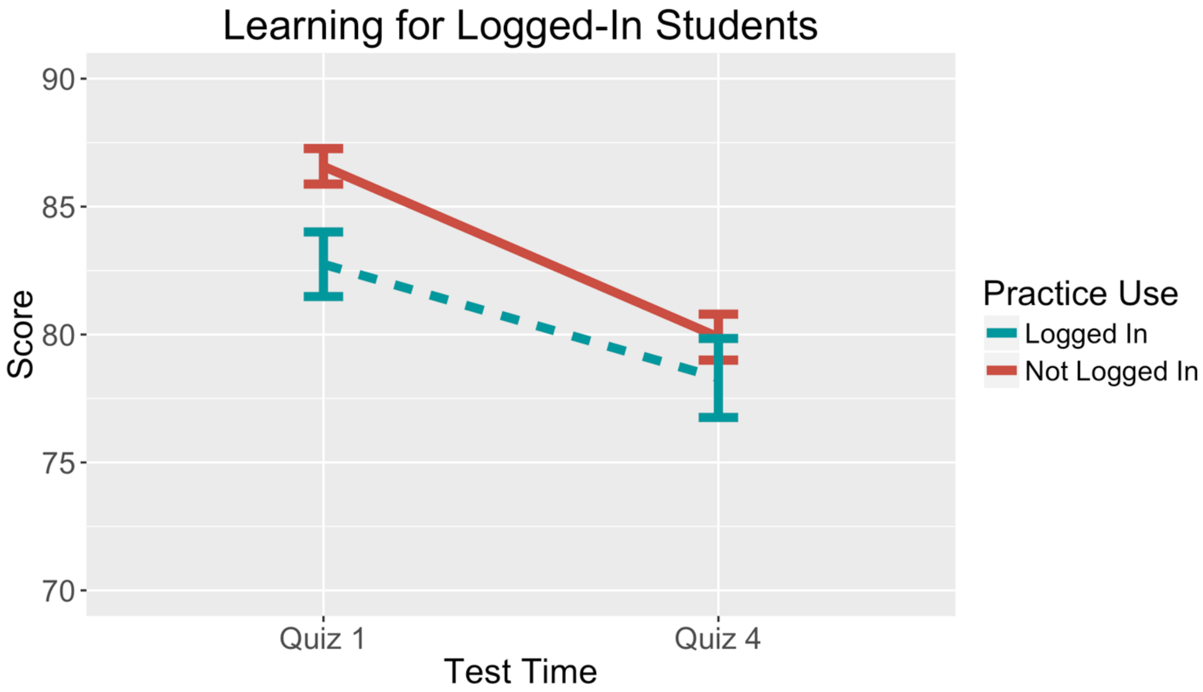


Figure 43: Performance on Quiz 1 and Quiz 4 for students who did or did not choose to log into Cloudcoder. Those who logged in started off significantly worse (p < 0.05 according to a logistic model), but were no different at Quiz 4 (p > 0.1).

I also checked whether interaction with Cloudcoder affected the chance that a student would drop out of the course, as I had data on which students dropped out (44 students in the dataset total). For this analysis, I examined the 398/410 students who had scores for Quiz 1. I found that both Quiz 1 scores and logging in were significantly negatively associated with dropping out according to a logistic model ($p < 0.05$, see Table 7), though there was no effect of condition.

	Estimate	Std. Error	t value	Probability
(Intercept)	**0.8123**	**0.0940**	**8.637**	**< 0.0001**
Quiz 1 Score	**-0.0082**	**0.0011**	**-7.574**	**< 0.0001**
Logged_in	**-0.0730**	**0.0354**	**-2.065**	**0.0396**

Table 7: A logistic model predicting whether students dropped out. Original factors: Quiz 1 score, logged in, started problem, attempted problem, condition. Students who performed worse on Quiz 1 and students who did *not* log in were more likely to drop out.

Discussion

First, I found a few mildly surprising effects by examining the students who chose to interact with the tool (by logging into Cloudcoder). These students had done significantly worse on the pretest, yet they caught up to their peers by the posttest. What is most interesting here is the fact that this effect was not apparent for the subset of students who went on to attempt and solve problems; it existed only for the students who chose to log in. The same effect was seen for dropout behaviors, where logging in was negatively associated with dropping out, regardless of whether a student went on to attempt problems or not. One hypothesis for why starting problems mattered less is that the student's *intention* to practice is more important than actually practicing; in other words, the student's internal motivation or mindset is what resulted in course improvement. To determine whether this hypothesis is true, further analysis needs to be done.

The differences found between conditions lead to both positive and negative hypotheses for how hints affected students. On the positive side, it seems that the students who were exposed to hints performed better from pretest to posttest than their counterparts; though I did not find significant effects, it is possible that the effect could become clearer with more statistical power. On the negative side, it also seems that merely seeing the Hint button (and occasionally interacting with it) scared several students away from using the system.

Why would students leave the system upon seeing a hint button? One hypothesis is that some students might have a negative perception of hints due to classroom culture; seeing a Hint button may have made those students dislike the system and refuse to use it as a result. However, if classroom culture was the cause, I would not have expected any of the problem dropout students to have requested hints, and 3/7 of them did. Alternatively, perhaps students associate hint availability with problem difficulty, and assume that the problems will be too hard as a result. However, the problem text is visible before the hint button appears (which should convince students otherwise), and there is no pretest difference between students in the hints-first condition who dropped the problem versus those who attempted it (though the students who dropped the problem do not see the same learning gain that their counterparts do). Alternatively, it is possible that this effect is simply due to statistical noise. To learn the true reason for this behavior, I will need to do deeper analysis of student conceptions about hints, practice, and problem difficulty.

Classroom Study 1: How Do Students Seek Help?

The results of the previous classroom study led to many new questions about how students perceive help-seeking and how they interact with hints. Therefore, I modified the study design by including a pre- and post- survey on motivational mindsets, ran interviews with students on help-seeking behaviors, and included the same practice problems and learning measurements as before. I ran the second classroom study in Spring 2016 with an expanded set of students.

Research Questions

In this study, I had two primary research questions. First, I aimed to replicate the results of the previous study, to better understand how practice problems and hints impacted student learning. Second, I asked whether student's individual factors (such as help-seeking beliefs) impacted hint usage and general behavior during practice.

Methods

For this study, I used students enrolled in 15-110 and 15-112, the two primary introductory programming courses at Carnegie Mellon, as participants. The main design of the experiment was the same as the design of the previous study: students were given login information for Cloudcoder, where they could complete practice programming problems related to the course. This time, all problems were available from the beginning (though sorted into different sections labeled by week), with 40 problems total (problems are included in Appendix 1). I gathered log data on student interactions with the website and student performance on quizzes and assignments to measure learning over time. Again, students were assigned to hints-first and hints-second conditions, to provide equal access to hints while allowing me to determine the effects of hints on learning.

Additionally, this study included survey and interview components. The survey was sent to students at the beginning of the semester, at the three-week point (when hint conditions switched), and at the end of the semester. The goal of the survey was to measure students' habits and beliefs regarding help-seeking as well as their grit (Duckworth et al, 2007), achievement goals (Elliot & McGregor, 2001), and mindset (Blackwell, Trzesniewski, & Dweck, 2007), in order to determine whether these factors impacted how they interacted with the hint system. The surveys used can be found in Appendix 3. Previous work suggests that students with mastery-oriented goals or high grit should achieve significantly better grades (de Raadt et al, 2005; Wolf & Jia, 2015), though findings for growth mindset are more mixed (Cutts et al, 2010).

I also varied the amount of encouragement students received in the initial invitation email, where half of the students were told “As a reminder- using ITAP is completely optional though we highly encourage you to try it as our previous studies suggest that working on practice problems may help you learn more.”, while the other half were only told “As a reminder- using ITAP is completely optional.” I hoped this encouragement manipulation might influence a greater variety of students to participate than were seen in the previous study.

I asked students who had used Cloudcoder by the three-week point if they would be interested in participating in a half-hour interview about their experiences with help-seeking and using the practice problem system. The interviews were based on a set of seven questions, shown below, about the students’ past experiences with programming, help-seeking, and ITAP. Interviews were conducted in my office.

1. In non-programming classes, do you ever feel like you need help while learning? How do you seek it out? Tell me about a time you did so.
2. Did you have a different help-seeking experience in your programming course than your other courses? How was it different?
3. Can you think of a time when you were working on a programming problem and you got stuck? Describe what that experience was like for you.
4. Did you manage to solve the problem you were stuck on? If so, how? If not, what did you do?
5. Have you ever gotten help from a teacher/TA in the programming course? Think back to the last time this happened. What did they do that was effective/ineffective?
6. Have you ever used ITAP to practice programming?
    a. Did you ever get stuck while using ITAP? If so, what did you do?
    b. Did you ever have access to feedback options (Test, Hint buttons) on ITAP? What was using each feedback type like?
7. In general, what kind of help do you think would be the most valuable for a novice programmer like you?

Finally, I collected basic demographic information by extracting name, major, and year data from Carnegie Mellon’s directory based on each student’s id. I estimated gender based on students’ names using an API which contained a database of over 200,000 name-gender matches (Genderize.io), and estimated 80% of the population’s genders with this approach.

All 15-110 and 15-112 students were opted into the study at the beginning of the semester (again, students could opt-out of data collection via Cloudcoder), and participation in the surveys, interviews, and use of Cloudcoder was all entirely optional and did not directly impact students’ grades. I announced the study in lecture (with permission from the course instructor), which was followed by emails containing login instructions and links to the surveys. 15-110 contained 207 students at the start of the semester, while 15-112 contained 478. Of these 685 total students, 5 requested that I not use their data for research purposes, leaving 680 to analyze. The number of students who participated in each facet of the study is shown in Table 8; it mainly demonstrates that both 15-110 and 15-112 had students use Cloudcoder, but the majority of students who participated in other parts of the study were from 15-112.

	ITAP Use	Survey 1	Survey 2	Survey 3	Interview
15-110	50	4	0	10	1
15-112	65	37	7	35	5

Table 8: Participants in each optional part of the study, split by course. The majority of survey and interview participants came from 15-112.

Additionally, I examined how many students used hints in the two courses and found that students requested 345 hints total (207 in 15-110, 138 in 15-112). 16 students in 15-110 requested hints (12.94 hints on average), while 15 students in 15-112 requested hints (9.2 hints on average).

Learning Metrics

To measure learning, I needed to identify appropriate assessments in both 15-110 and 15-112. 15-112 has weekly assignments and quizzes, but the initial assignments are again too close to ceiling to be usable (99.7/100 and 98.8/100 for the first two assignments, compared to 86.5/100 and 94.9/100 for the first two quizzes). 15-110 has weekly written assignments and programming assignments, but not weekly quizzes. I use the programming assignments, even though they start fairly close to ceiling (9.5/10 and 9.0/10 for the first two), for lack of better data.

Just like in the previous study, 15-112 students again mostly used the system at the beginning of the study (in fact, in the first week), and usage decreased dramatically after that point. 15-110 students, on the other hand, had a low but steady rate of participation until weeks 4-5, when a larger number of students used the system just before the first exam of the course. This effect is shown in Figure 44. Unfortunately, this again makes it difficult to do a proper crossover analysis, as most hint use in 15-112 happened during the hints-first phase and the majority of hints use in 15-110 happened during hints-second. Therefore, I will attempt to analyze the effect of hints in 15-112 primarily during the first three weeks with Quiz 1 as a pretest and Quiz 4 as a posttest, and I'll primarily analyze 15-110 during the following three weeks with Assignment 3 as a pretest and Assignment 7 as a posttest.

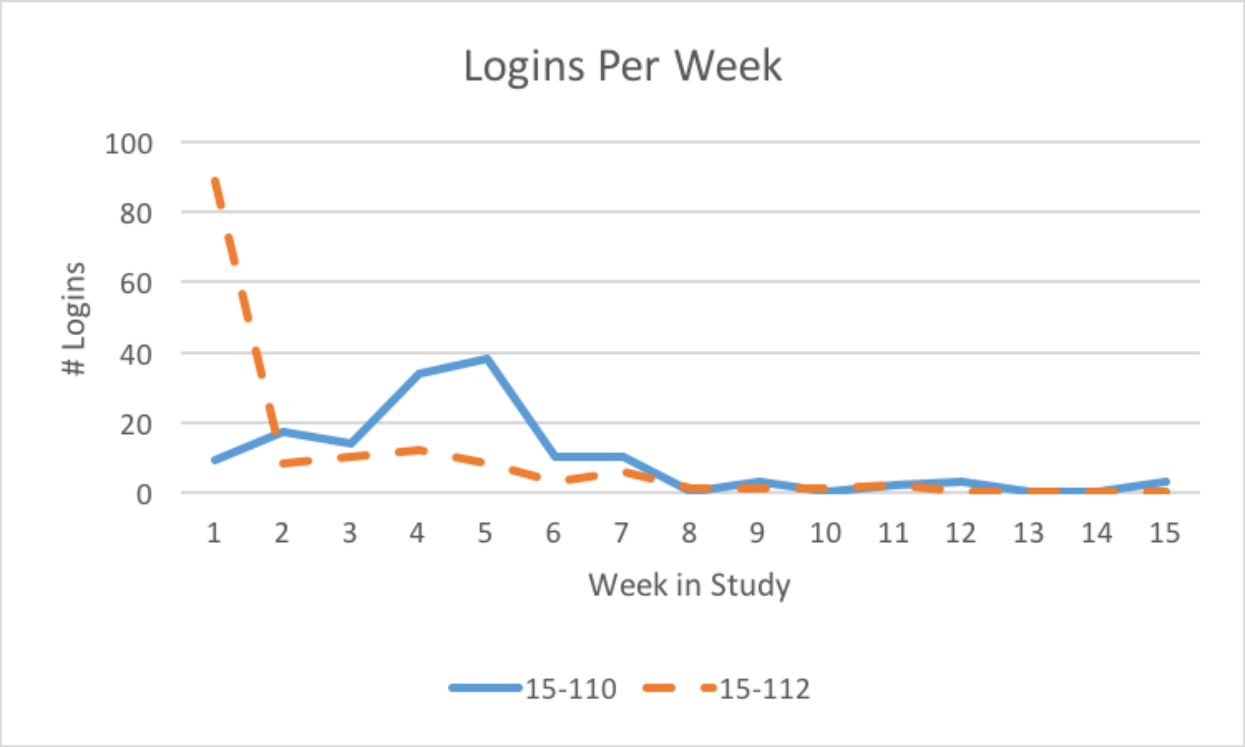


Figure 44: Number of logins per week during the study for each course. 15-112's peak occurred at the beginning, while 15-110 peaked at the 4-5 week mark, before their first exam.

Results

There are several different factors that I investigated in this study. First, I re-ran the analyses from the previous study to determine whether hints impact learning and/or whether choosing to practice impacts learning. Second, I investigated whether there is a relationship between students' reported motivational and demographic factors and their use of the practice system and/or hints. And finally, I analyzed the feedback students provided in the surveys and interviews to look for any important trends or insights into how students view the help-seeking process.

Analysis of Hints

First, I re-run the analyses of the previous study, to see if they are replicated when controlling for demographic information. The first question: does having access to hints improve students' learning over time? For this analysis and the following ones, I remove students who did not complete the pretest or posttest, as well as non-traditional students (staff members) who do not fit well into the year-of-study model. 397 students in 15-112 and 165 students in 15-110 remain. In 15-112, students scored an average of 86.7/100 on the pretest and 73.3/100 on the posttest; in 15-110, students scored an average of 8.31/10 on the pretest and 7.82/10 on the posttest. Overall, according to a repeated measures ANOVA, there is no evidence of significant impact of hint condition (or encouragement condition) on learning from pretest to posttest in either class, as is shown in Table 9.

15-110	Mean Sq	NumDF	F.value	Probability
Test Time	*16.8447*	*1*	*3.45*	*0.0651*
Hint Condition	2.3427	1	0.48	0.4896
Encourage Condition	0.0025	1	0.00	0.9819
Interaction: Test Time & Hint Condition	0.4265	1	0.09	0.7680
Interaction: Test Time & Encourage Condition	0.0604	1	0.01	0.9116
Interaction: Hint Condition & Encourage Condition	0.0713	1	0.01	0.9040

15-112	Mean Sq	NumDF	F.value	Probability
Test Time	**35798**	**1**	**294.65**	**< 0.0001**
Hint Condition	33	1	0.27	0.6043
Encourage Condition	6	1	0.05	0.8243
Interaction: Test Time & Hint Condition	63	1	0.52	0.4723
Interaction: Test Time & Encourage Condition	55	1	0.46	0.5003
Interaction: Hint Condition & Encourage Condition	0	1	0.00	0.9490

Table 9: Two repeated measure ANOVAs checking the effect of condition on performance in 15-110 (top) and 15-112 (bottom). There was no significant effect of condition and no interaction.

Investigating only the students who attempted at least one problem in Cloudcoder leaves 38 students in 15-112 (21 in hints-first, 17 in hints-second) and 28 students in 15-110 (11 in hints-first, 17 in hints-second). This time, there was no noticeable difference between the conditions according to a repeated measures ANOVA, as is shown in Table 10 and Figure 45; therefore, the non-significant learning gain seen in the previous study was probably due to chance. I also examined the effect of condition on attempting problems (after a problem has been started), as that had generated a startling result in the previous study. This time, there was no effect of condition on dropping a problem; only six students across the two courses exhibited this problem-dropping-out behavior at all, two with hints available, four without.

15-110	Mean Sq	NumDF	F.value	Probability
Test Time	4.4908	1	1.04	0.3191
Hint Condition	8.8749	1	2.05	0.1660
Encourage Condition	0.6784	1	0.16	0.6961
Interaction: Test Time & Hint Condition	0.1930	1	0.04	0.8347
Interaction: Test Time & Encourage Condition	0.4499	1	0.10	0.7502
Interaction: Hint Condition & Encourage Condition	2.0783	1	0.48	0.4957

15-112	Mean Sq	NumDF	F.value	Probability
Test Time	**3033.92**	**1**	**28.59**	**< 0.0001**
Hint Condition	28.92	1	0.27	0.6051
Encourage Condition	179.35	1	1.69	0.2023
Interaction: Test Time & Hint Condition	63.76	1	0.60	0.4435
Interaction: Test Time & Encourage Condition	195.55	1	1.84	0.1833
Interaction: Hint Condition & Encourage Condition	17.10	1	0.16	0.6906

Table 10: Two repeated measure ANOVAs checking the effect of condition on performance after attempting a problem in 15-110 (top) and 15-112 (bottom). There was no significant effect of condition and no interaction.

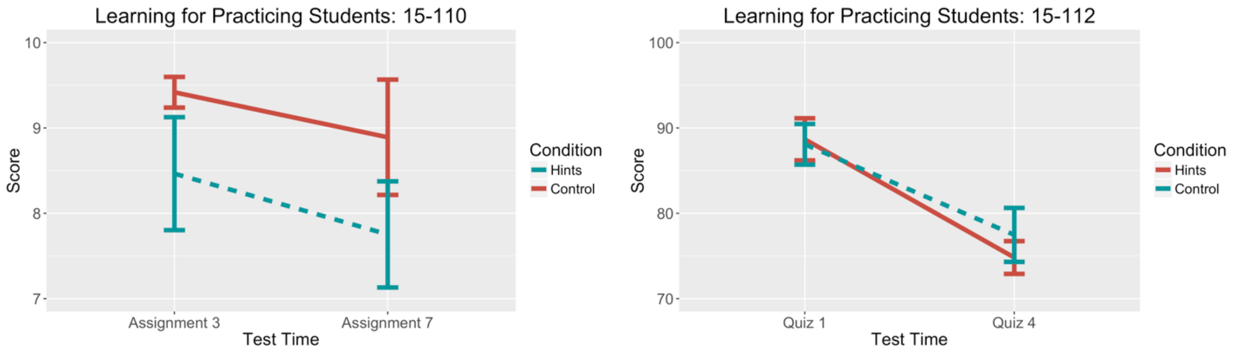


Figure 45: Learning results for practicing students in 15-110 (left) and 15-112 (right). Neither case shows even a potential effect of condition on learning according to a repeated measures ANOVA ($p > 0.1$).

Again, I investigated whether hint/encouragement condition and/or hint use impacted how students interacted with the system. Surprisingly, I found that in 15-110 students in the extra encouragement condition started *fewer* problems than their counterparts in the normal encouragement condition (14.4 problems on average for the control group, 8 problems on average for experimental) with marginal significance according to a t-test ($p < 0.1$). The same effect was not found for 15-112 (8.8 problems for control, 8.1 problems for experimental, $p > 0.1$ on a t-test).

I also investigated the effect of condition on time spent in the practice problem system and found effects in both courses. In 15-112, according to a linear model, time spent was positively affected by both the number of problems attempted and the number of hints requested (see Table 11); in 15-110, according to a linear model, being in hints-second (having hints available) was negatively correlated with time spent, but number of hints requested was positively correlated (see Table 12). In other words, requesting more hints results in spending more time in the system, but it is unclear whether having access to hints has an independent effect on time on task.

	Estimate	Std. Error	t value	Probability
(Intercept)	-131.81	158.47	-0.832	0.4107
Problems Attempted	**160.04**	**14.14**	**11.316**	**< 0.0001**
Hints Requested	**60.82**	**14.10**	**4.315**	**0.0001**

Table 11: A linear model predicting time spent in Cloudcoder for students in 15-112. Original factors: hint condition, encouragement condition, # problems started, # problems attempted, # hints requested. Students who attempted more problems and requested more hints spent more time. Adjusted R-squared: 0.8197

	Estimate	Std. Error	t value	Probability
Intercept	-223.40	591.86	-0.377	0.7094
Condition: hints_second	**-1170.04**	**548.72**	**-2.132**	**0.0444**
Condition: encourage_experiment	838.78	492.85	1.702	0.1029
Problems Started	*581.53*	*284.77*	*2.042*	*0.0533*
Problems Attempted	-420.25	326.17	-1.288	0.2110
Hints Requested	**166.40**	**37.24**	**4.469**	**0.0002**

Table 12: A linear model predicting time spent in Cloudcoder for students in 15-110. Original factors: hint condition, encouragement condition, # problems started, # problems attempted, # hints requested. Students who were in hints-second spent less time, but students who requested more hints spent more time, so the effects cancel out. There is also a marginal positive effect for number of problems started. Adjusted R-squared: 0.8167

Analysis of Cloudcoder Use

Additionally, I investigated whether there is a difference between students who chose to use Cloudcoder and those who did not. This time, a repeated measures ANOVA did not find a difference at pretest or in learning based on whether students logged in F(1,395) = 0.96, p > 0.1 or an interaction between test time and logging in F(1,395) = 1.53, p > 0.1. I also looked at how using Cloudcoder interacted with dropout rates and found that students who attempted problems in 15-112 or logged in in 15-110 were less likely to drop out. Additionally, students in higher years (upperclassmen and graduate students) in both courses were more likely to drop out. These effects are shown in logistic models shown in Table 13 and Table 14. Note that I do not have pretest scores for students who dropped out, so it is possible that these effects would be negated if I did have that data.

15-112	Estimate	Std. Error	z value	Probability
Intercept	**-2.8666**	**0.2373**	**-12.082**	**< 0.0001**
Started Problem	*1.7747*	*0.9457*	*1.877*	*0.0606*
Attempted Problem	**-3.2443**	**1.2162**	**-2.667**	**0.0076**
Year	**0.4845**	**0.0796**	**6.084**	**< 0.0001**

Table 13: A logistic model predicting dropout for students in 15-112. Original factors: logged in, started problem, attempted problem, hint condition, encouragement condition, year. Students who attempted fewer problems were more likely to drop out, as were students in higher years.

	Estimate	Std. Error	z value	Probability
Intercept	**-1.6201**	**0.3160**	**-5.128**	**< 0.0001**
Logged In	**-1.9634**	**0.7562**	**-2.596**	**0.0094**
Year	**0.3202**	**0.1268**	**2.526**	**0.0116**

Table 14: A logistic model predicting dropout for students in 15-110. Original factors: logged in, started problem, attempted problem, hint condition, encouragement condition, year. Students who did not log in were more likely to drop out. Students in higher years were also more likely to drop out.

Analysis of Motivational Factors

Next, I investigate the relationship between students' reported motivational data and their use of the Cloudcoder system. First, I check whether the students who responded to at least one of the surveys accurately represent the overall population of the course. A generalized linear model did not show with any significance that any of the demographic factors impacted whether or not students took the survey, but it did show that students who logged into ITAP were more likely to take the survey ($p < 0.001$). This result is not surprising, as these are students who are opting into participation already, but it does mean that survey results must be viewed skeptically, as the population may not be representative of the whole group of students.

Ideally, I'd like to analyze the survey results of students who completed all three surveys, but only three students satisfy these criteria. Furthermore, only ten students completed both the first and last surveys, which again provides limited analysis capability. When I compare the responses of those ten students, I find that only two factors changed from pre to post (according to a paired t-test): estimated level of programming knowledge (which went up, from 2.5/7 to 4.5/7, $p < 0.001$), and having a mastery approach achievement goal (which went down from 6.7/7 to 6/7, $p < 0.01$). The first is not surprising at all, as one would expect students to learn. The second *is* surprising, but is possibly due to students skewing towards the ceiling in the pretest.

Since most of the factors did not significantly change from pretest to posttest, I combine the data from the two tests to draw from a larger sample in order to analyze the effect of most factors on behavior in the system. To do this, I average the results of students who completed both pretest and posttest, and take the individual results of students who completed either pretest or posttest. I do minimal analysis of mastery approach, prior knowledge, or the other factors only collected at the pretest, as I do not have a large enough dataset to draw from. Any results found with this dataset will need to be viewed skeptically, as I am combining data from two different points of time, but it may still provide hypotheses for future study.

First, I investigate whether there are any differences between the two courses, to see whether the populations are similar and can be combined. The results of the first survey demonstrated that students from 15-112 tended to have more programming experience than students from 15-110, as is shown in Table 15 (where all p values are taken from t-tests between the two courses). This finding is not surprising, as students with more experience are normally advised to take 15-112. The only other significant differences between courses were shown in self-reported performance avoidance ratings, where 15-110 students had a significantly higher reported rating on a Likert scale than 15-112 students, and use of practice resources, where 15-110 students again had a higher rating than 15-112 students. Again, this result can be easily explained, as 15-112 is known to be more difficult than 15-110, so students worried about failing would be more likely to take 15-110.

	15-110	15-112
Programming Knowledge (7-Pt Likert Scale) ***	1.2	2.7
% Students with Prior Language Experience *	20%	84%
% Students with Prior Python Experience ***	0%	54%
Math SAT Self-Report	720	746.77
Grit Self-Rating (7-Pt Likert Scale)	6.29	6.07
Mastery Approach Self-Rating (7-Pt Likert Scale)	6.4	6.73
Mastery Avoidance Self-Rating (7-Pt Likert Scale)	4.79	4.28
Performance Approach Self-Rating (7-Pt Likert Scale)	4.71	4.69
Performance Avoidance Self-Rating (7-Pt Likert Scale) *	5.75	4.5
Fixed-to-Growth Mindset Rating (7-Pt Likert Scale, Fixed low)	4.55	4.73
Practice Resource Use (7-Pt Likert Scale) *	3.5	5.31
TA/Office Hour Resource Use (7-Pt Likert Scale)	3.79	4.27

Table 15: Self-reported survey measures for students in 15-110 and 15-112.
* p < 0.05, ** p < 0.01, *** p < 0.001.

Next, I investigate whether any of the collected factors influence students' use of the practice problem system, in terms of how many problems they started, attempted, and solved. I ran Poisson models on the combined pretest-posttest data for students who logged into Cloudcoder to check the effects of the collected factors. The results of these models are shown in Tables 16-18. Several patterns appeared along the lines of student motivation goals, but it is difficult to be sure that they are consistent, as running individual correlations between problems started/attempted/solved and the mastery goals sometimes flipped the effect sizes. Therefore, I concentrate on grit and growth mindset. I found that students' self-reported grit was associated with starting, attempting, and solving fewer problems. It is unclear why higher grit would lead to less work, especially since one would normally predict the opposite effect; however, this might be a ceiling effect, as the average grit self-reports were very high. Further investigation will be needed to determine why this occurred. Additionally, students with stronger growth mindsets and students who used TA resources more tended to start more problems; neither of these results are particularly surprising.

	Estimate	Std. Error	z value	Probability
Intercept	**1.7170**	**0.4737**	**3.625**	**0.0003**
Mastery Avoidance Ranking	**0.3510**	**0.0559**	**6.281**	**< 0.0001**
Performance Approach Ranking	**-0.1237**	**0.0511**	**-2.422**	**0.0154**
Performance Avoidance Ranking	**-0.1880**	**0.0394**	**-4.768**	**< 0.0001**
Grit Ranking	**-0.6545**	**0.0811**	**-8.070**	**< 0.0001**
Growth Mindset Ranking	**0.5166**	**0.0655**	**7.882**	**< 0.0001**
TA Use Ranking	**0.2940**	**0.0440**	**6.683**	**< 0.0001**

Table 16: A Poisson model predicting number of problems started, using data from the combined pretest and posttest. Original factors: course, hint condition, encouragement condition, year, mastery avoidance level, performance approach level, performance avoidance level, grit level, growth mindset ranking, practice resource use ranking, TA use ranking. On running individual correlations, effect signs hold for all factors except performance approach.

	Estimate	Std. Error	z value	Probability
Intercept	**2.3624**	**0.5399**	**4.376**	**< 0.0001**
# Problems Started	**0.1129**	**0.0088**	**12.829**	**< 0.0001**
Course: 15-112	**-0.7397**	**0.2312**	**-3.200**	**0.0014**
Mastery Avoidance Ranking	**-0.1147**	**0.0566**	**-2.029**	**0.0425**
Performance Approach Ranking	**0.1382**	**0.0605**	**2.282**	**0.0225**
Performance Avoidance Ranking	-0.0601	0.0413	-1.455	0.1457
Grit Ranking	**-0.1437**	**0.0609**	**-2.358**	**0.0184**

Table 17: A Poisson model predicting number of problems attempted, using data from both pretest and posttest. Original factors: # problems started, course, hint condition, encouragement condition, year, mastery avoidance level, performance approach level, performance avoidance level, grit level, growth mindset ranking, practice resource use ranking, TA use ranking. On running individual correlations, effect signs hold for all factors except mastery avoidance.

	Estimate	Std. Error	z value	Probability
Intercept	**2.4677**	**0.5622**	**4.389**	**< 0.0001**
# Problems Started	**0.1116**	**0.0097**	**11.530**	**< 0.0001**
# Hints Requested	**0.0216**	**0.0070**	**3.092**	**0.0020**
Hint Condition: Hints-second	**0.6647**	**0.1986**	**3.346**	**0.0008**
Mastery Avoidance Ranking	**-0.1649**	**0.0628**	**-2.626**	**0.0086**
Performance Avoidance Ranking	**-0.1669**	**0.0499**	**-3.344**	**0.0008**
Grit Ranking	**-0.2546**	**0.0751**	**-3.390**	**0.0007**
Growth Mindset Ranking	0.1320	0.0908	1.454	0.1460

Table 18: A Poisson model predicting number of problems solved, using data from both pretest and posttest. Original factors: # problems started, # problems attempted, # hints requested, course, hint condition, encouragement condition, year, mastery avoidance level, performance approach level, performance avoidance level, grit level, growth mindset ranking, practice resource use ranking, TA use ranking. On running individual correlations, effect signs hold for all factors except hint condition and mastery avoidance.

Finally, I investigated how these different factors affected students' use of hints. I ran a Poisson model using the pretest and posttest combined dataset for students who had hints available during the time period assessed; the results are shown in Table 19. This model is likely overfitting due to the small amount of data in the usable dataset (20 entries), but it seems that performance-oriented students and students with higher grit requested more hints, while students who used more practice resources requested fewer hints. I also see a potential negative association with growth mindset, but an individual correlation between growth mindset and hints requested returned a positive association, so I cannot be sure.

	Estimate	Std. Error	t value	Probability
Intercept	**-6.6842**	**1.7634**	**-3.790**	**0.0002**
# Problems Attempted	**-0.8633**	**0.3272**	**-2.638**	**0.0083**
# Problems Started	**1.0958**	**0.3071**	**3.568**	**0.0004**
Encouragement Condition: Extra	**-4.1485**	**0.7073**	**-5.866**	**< 0.0001**
Year	**0.2774**	**0.0827**	**3.355**	**0.0008**
Mastery Avoidance Ranking	**-0.4507**	**0.1798**	**-2.506**	**0.0122**
Performance Approach Ranking	**0.4769**	**0.1366**	**3.491**	**0.0005**
Performance Avoidance Ranking	**0.4991**	**0.1856**	**2.689**	**0.0072**
Grit Ranking	**1.7556**	**0.3506**	**5.007**	**< 0.0001**
Growth Mindset Ranking	**-0.6421**	**0.1853**	**-3.466**	**0.0005**
Practice Problem Use Ranking	**-0.5206**	**0.1416**	**-3.677**	**0.0002**

Table 19: A Poisson model predicting number of hints requested, using both pretest and posttest data. Original factors: # problems started, # problems attempted, course, encouragement condition, year, mastery avoidance level, performance approach level, performance avoidance level, grit level, growth mindset ranking, practice resource use ranking, TA use ranking. On running individual correlations, effect signs hold for all factors except mastery avoidance and growth mindset.

Qualitative Analysis

Finally, I analyze the qualitative data gathered from the surveys and interviews. On the final survey, I asked students why they did or did not choose to use the practice problem system, whether the feedback they got was helpful, and if they had any suggestions for improving the system. (I also asked students for suggestions in the mid-survey, and include the responses here). Students who used the system stated that they wanted extra practice problems, or wanted to study for exams. Students who did not use the system said that they had enough practice problems already, or had forgotten that the system was available.

When asked about the feedback, the 4/10 comments which explicitly mentioned hints were, interestingly, largely negative. One student stated that “It was helpful, but not exhaustively helpful”; another said that “The hints were vague and not really specific to the code”. A third student elaborated on these points, saying “Sometimes the feedback wasn't totally clear, or applicable. It would give the same hints/feedback over vs modifying it to help spur useful thoughts”. Combining the survey results with ITAP use logs showed that several of these students received multiple hints in a row which targeted the same part of their solution, but that those hints only gave small amounts of information; perhaps this frustrated the students. Finally, one student did say the hints were “helpful when i got stuck”, and several students gave positive feedback on the test case results.

When asked for suggestions, many students gave feedback on the Cloudcoder interface (especially navigational elements), but some also gave suggestions for better feedback. Several students asked for more problems, or specifically more difficult problems. Two different students stated that they’d like to be able to see the solutions, and two other students asked that the hints be better explained. As one stated: “Improve the hints and make them more descriptive about how to correct the code, even if the hints have to be general.”

In the interviews, students discussed their relationship with practice and help-seeking in more detail. When asked about how they sought out help, the majority (5/6) mentioned getting help from classmates; other help sources included office hours, university-sponsored supplemental instruction, and online resources. Students also mentioned that they got help from TAs a lot in their programming class, potentially due to strict rules prohibiting many forms of collaboration in the introductory programming courses.

When asked to think about a situation where they got stuck on a programming assignment, half of the students brought up how they had to spend several hours debugging in order to solve relatively small problems. One student literally said that it took them two to three hours to find a “very stupid” mistake in a single line of code. One student discussed having difficulty starting an assignment, but another said that it wasn’t hard to find a way to solve a problem, it was just hard to get it right. Several students mentioned using print statements to find a problem, while others said they wrote it out on paper. Only one student specifically mentioned getting help from a TA.

When asked specifically about help from teachers and TAs, some students mentioned that they would help with debugging, but they weren’t as helpful with conceptual problems. Several students said that the amount of help given depended on which TA came to you, with TAs giving varying levels of detail. Students also complained about long wait times in office hours; one student talked about waiting an hour, getting advice from the TA that didn’t really help, and then needing to put her name on the waitlist again immediately, which she said was frustrating. Another student mentioned that she wished she could get help for longer periods of time, but that she “... can’t have a personal TA for the whole time she’s doing homework”.

When asked about their use of Cloudcoder, two of the six students hadn’t used it (because, they said, they were too busy), and two said they had done some of the problems but had found them too easy. One student mentioned that she had tried to do some of the problems, but that she gave up when she had difficulty with a runtime error. Further investigation revealed that she did not have hints available when she encountered the runtime error. The final student (who was the only student in 15-110) said that he liked the system, that he never got stuck because he used the test cases and the hints a lot. He said that he often encountered syntax errors, and that clicking the hint button a lot helped him find his errors.

Finally, when asked about what type of help they thought would be most useful, half of the students mentioned specific, targeted help during problem-solving (which hints would provide, though only one student made this connection). Two others wanted more peer assistance, and other students mentioned alternative solutions and conceptual reviews.

Discussion

First, it appears that I have found evidence against most of the apparent results of the previous study, as I saw neither additional learning for students with hints nor additional problem dropout for students with hints. The lack of problem dropout might have occurred because I changed the Cloudcoder interface to show a greyed-out Hint button for students in the control condition, so the interface would not change too dramatically between conditions. I also no longer see a difference at pretest between students who chose to use Cloudcoder and those who chose not to, but I do see that students who attempt problems in the system are less likely to drop out, which corroborates the previous result where students who logged into the system were less likely to drop out.

One new and unusual effect I found was related to the encouragement manipulation, where half of the students were shown an additional line in their invitation email about how doing practice problems could help them do better in the course. There was no difference between conditions on how many students tried problems in Cloudcoder, but, surprisingly, the students who received extra encouragement started *fewer* problems than their counterparts in the control condition. It is unclear why this might be the case.

I also found that asking for more hints was correlated with spending more time in the system, but it is possible that students who need more help (and therefore need more time) just ask for more hints. Without a more representative population, it is difficult to tell whether the hint use caused the extra time spent or not.

When I analyzed the survey results, I found a few small effects that may help give insight on how students view help-seeking. For example, students with higher grit ranking started, attempted, and solved fewer problems, which is surprising and deserves further investigation. The results on hints are shaky due to low sample size, but it seems possible that students who request hints have higher grit and are less likely to engage with other practice resources.

Students’ open-form feedback in the surveys and the interviews indicated that hints could be improved by making them less vague and more extensive. However, hints were viewed as helpful to one student I interviewed, who asserted that they helped him solve syntax errors. In general, it seems that there is a need for targeted, direct feedback, but work still needs to be done to present that information in an optimal way for students to use.

4. Evaluating Hint Representations in Different Contexts

The analysis of student feedback from the previous study demonstrated that the format and level of detail of the hints provided by ITAP was lacking, in the students' opinion at least. It can be debated whether or not more information would lead to better learning, but learning is not the only factor that needs to be considered in the question of how to format hints. After all, most hints are provided upon student request; if students do not believe in the hints, they will not request them, and therefore will get no use out of them at all. Therefore, I decided to run a usability study to test different forms and detail levels of hints, to see which forms were preferred by students. The different experiences of students in 15-112 and 15-110 led me to believe that user preference might shape the type of hint they desire, so I collected information on the users' prior experience and help-seeking beliefs. I also think it possible that the type of error a student encounters might influence what type of hint they would most benefit from, so I experimented with various types of errors as well.

First, I provide a brief review on different feedback representations that can be provided during the programming process. A full classification has been provided in (Le, 2016), which separates feedback broadly into yes/no, syntax, semantic, layout, and quality feedback. I am particularly interested in semantic feedback, which can be separated into two categories: intention-based and code-based analysis. Intention-based analysis is somewhat tied to the concept of a reference program (which can provide intention), while code-based analysis relies more on the student submission itself. Until now, I have focused mainly on code-based approaches; an intention-based approach could prove useful as well. It can also be beneficial to provide algorithmic feedback, which focuses more on the algorithmic components needed than the syntax of the code; this approach was shown to lead to more productive edits in previous work (Sudol-DeLyser, 2014).

When generating hints specifically, there are several hint types used by human tutors. A formative study of hints provided by teachers on an online forum found that teachers provided many specific types of hints, including five amenable to automatic generation: transformation, location, data, behavior, and example (Suzuki et al, 2017). The next-step hints I have provided so far fall under the transformation category, with location information as well. Data and behavior hints tell the student more about *why* the error is occurring, while example hints clarify expected output for the problem.

New Hint Representations

I took insights from observations gained in the previous studies in designing the new types of hints which would be evaluated in this usability study. First, I determined that there were two factors that would influence the hints presented: the *content* of the hint and the *level of detail* provided. In other words, I could present a hint that dealt with the whole solution at low detail, or a hint that focused on only one part to be changed in high detail.

Next, I analyzed student feedback to identify possible new content for hints that could be derived via data-driven approaches. I identified two major pieces of content that are created by ITAP: the edit between the student's state and its goal, and the personalized goal itself. I then generated four hint types based on these two pieces of content.

First, there are *location hints*. These use the edit content to identify where in the code the edit must happen, but they do not address what the edit should be (examples shown in Figure 46). Location hints are like error highlighting in IDEs, showing where the syntax errors may be located without telling the student how to fix them. I chose to include these hints as they are similar to hints given by some TAs.

`Bugs occur in the following locations:`
`At line 5`

`Bugs occur in the following locations:`
`At line 5, column 24`
`At line 7, column 22`

`Bugs occur in the following locations:`
`At line 5, column 24`
`At line 7, column 22`
`At line 5, column 27`

`Bugs occur in the following locations:`
`At line 5, column 24`
`At line 7, column 22`
`At line 5, column 27`

Figure 46: Examples of location hints at all four levels of detail.

Second, there are *next-step hints*. These are the traditional hints I've been using in previous studies, but potentially without the obfuscation used previously (examples shown in Figure 47). These hints are akin to autocomplete, as they suggest what the student could do next.

At line 5, column 24 swap **i** with **j** in the tuple
At line 5, column 24 swap **i** with **j** in the tuple At line 7, column 22 replace **1** with **(-1)** in the tuple
At line 5, column 24 swap **i** with **j** in the tuple At line 7, column 22 replace **1** with **(-1)** in the tuple At line 5, column 27 change **i** to **(('~left value~' - '~right value~') - i)** in the tuple
At line 5, column 24 swap **i** with **j** in the tuple At line 7, column 22 replace **1** with **(-1)** in the tuple At line 5, column 27 change **i** to **((len(l) - 1) - i)** in the tuple

Figure 47: Examples of next-step hints at all four levels of detail.

Third, there are *structure hints*. These hints use the goal state generated by ITAP, but obfuscate most of the details, only revealing the main statement types on each line (examples shown in Figure 48). I chose to use these hints as some students wanted hints from the very beginning of the problem, and this type of hint would provide some idea of how to approach the problem without giving everything away. This approach has been used before to some success in the AutoTeach system, which reveals high-level portions of a specified solution while obfuscating the interior code to provide multiple levels of hints (Antonucci et al, 2015).

```
Here is the structure of the working program:
def ...
    for ...
        for ...
            if ...
                return ...
    else ...
        return ...
```

```
Here is the structure of the working program:
def ~function name~(~var~):
    for ~var~ in ~function name~(~arg~, ~arg~):
        for ~var~ in ~function name~(~arg~):
            if (~var~[~var~][~var~] ~op~ ~string~):
                return (~var~, ~left side~ ~op~ ~var~))
    else:
        return (~number~, ~number~)
```

```
Here is the structure of the working program:
def ~function name~(~var~):
    for ~var~ in ~function name~(~number~, ~function name~(~var~)):
        for ~var~ in ~function name~(~function name~(~arg~)):
            if (~var~[~var~][~var~] ~op~ ~string~):
                return (~var~, (~left side~ ~op~ ~number~) ~op~ ~var~))
    else:
        return (~number~, ~number~)
```

```
Here is the structure of the working program:
def ~function name~(~var~):
    for ~var~ in ~function name~(~number~, ~function name~(~var~)):
        for ~var~ in ~function name~(~function name~(~var~[~var~])):
            if (~var~[~var~][~var~] ~op~ ~string~):
                return (~var~, ((~function name~(~var~) ~op~ ~number~) ~op~ ~var~))
    else:
        return (~number~, ~number~)
```

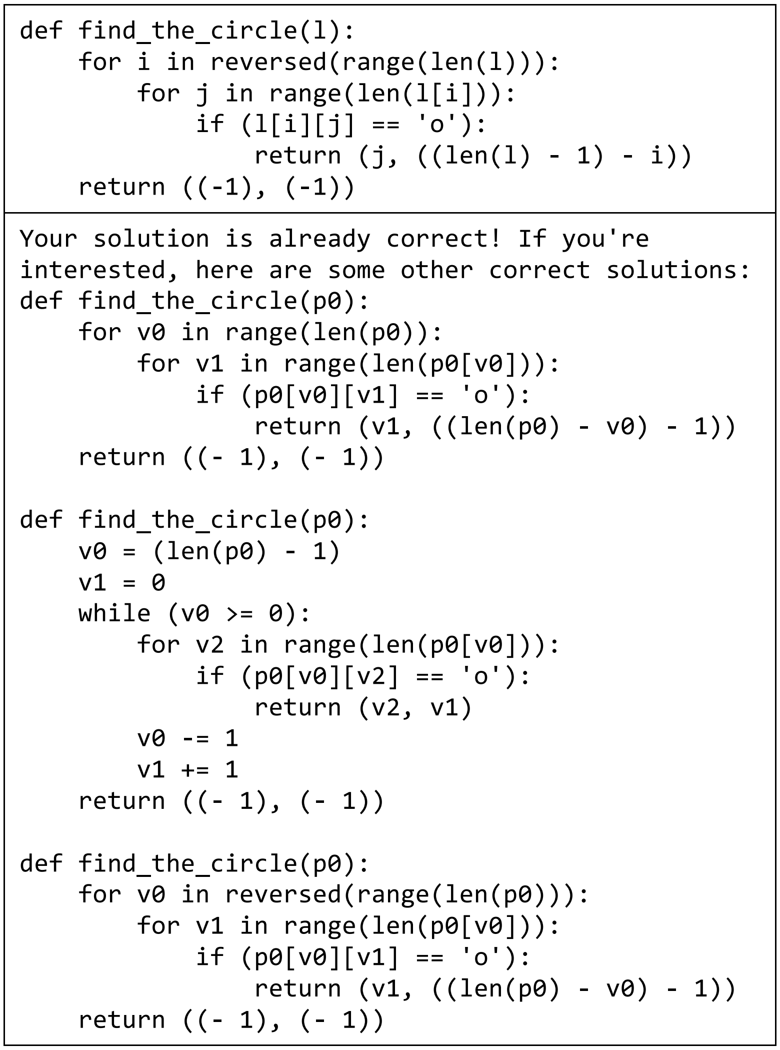
Figure 48: Examples of structure hints at all four levels of detail.

Fourth, there are *solution hints*. These hints simply tell the student what their personalized goal state is (example shown in Figure 49). They can be viewed as turning problems into example code, instead of worked examples or practice problems. I included this type of hint because many students relied on example code to help them learn syntax and structure at the beginning.

```
Here is an alternative correct solution:
def findTheCircle(l):
    for i in reversed(range(len(l))):
        for j in range(len(l[i])):
            if (l[i][j] == 'o'):
                return (j, ((len(l) - 1) - i))
    return ((-1), (-1))
```

Figure 49: An example of a solution hint at the lowest level of detail. Higher levels of detail include alternative solutions (selected from the solution space); these are not shown to save space.

Finally, I generated *example hints*, which could be given to students who had already gotten the problem right. These hints would show the students examples of other solutions that were similar to their solution, very different from their solution, and the most popular in the solution space (example shown in Figure 50). I generated this type of hint because some students mentioned that they never got the chance to see other ways that problems were solved, and because this could serve as an additional way to see example code over time.



```
def find_the_circle(l):
    for i in reversed(range(len(l))):
        for j in range(len(l[i])):
            if (l[i][j] == 'o'):
                return (j, ((len(l) - 1) - i))
    return ((-1), (-1))
```

```
Your solution is already correct! If you're
interested, here are some other correct solutions:
def find_the_circle(p0):
    for v0 in range(len(p0)):
        for v1 in range(len(p0[v0])):
            if (p0[v0][v1] == 'o'):
                return (v1, ((len(p0) - v0) - 1))
    return ((- 1), (- 1))

def find_the_circle(p0):
    v0 = (len(p0) - 1)
    v1 = 0
    while (v0 >= 0):
        for v2 in range(len(p0[v0])):
            if (p0[v0][v2] == 'o'):
                return (v2, v1)
        v0 -= 1
        v1 += 1
    return ((- 1), (- 1))

def find_the_circle(p0):
    for v0 in reversed(range(len(p0))):
        for v1 in range(len(p0[v0])):
            if (p0[v0][v1] == 'o'):
                return (v1, ((len(p0) - v0) - 1))
    return ((- 1), (- 1))
```

Figure 50: An example of an example hint, to be shown to students with already-correct solutions. The student's original solution is shown at the top, while the hint is shown at the bottom.

Usability Study: How Does Context Affect Hint Choice?

Once I had formulated the new hint types, I designed a multi-part usability study that could test student reactions to the different hints. This study consisted of a survey (to gather individual user information), a set of hint-selection exercises (to see what types of hints users preferred), and a set of free-coding exercises (to observe how users interacted with previously-available hints in a real coding environment).

Research Questions

Within this study, I had a set of research questions I wished to answer. First, I wanted to know whether the types of hints users requested changed based on the users' own personal characteristics; for example, does a more experienced student want less detail in hints than a novice? Second, I wanted to know whether the types of hints users requested changed based on the type of error they had encountered; do larger errors require more content in hints? Finally, I wanted to know which types of hints were considered most helpful, so that I could use those hints in future work.

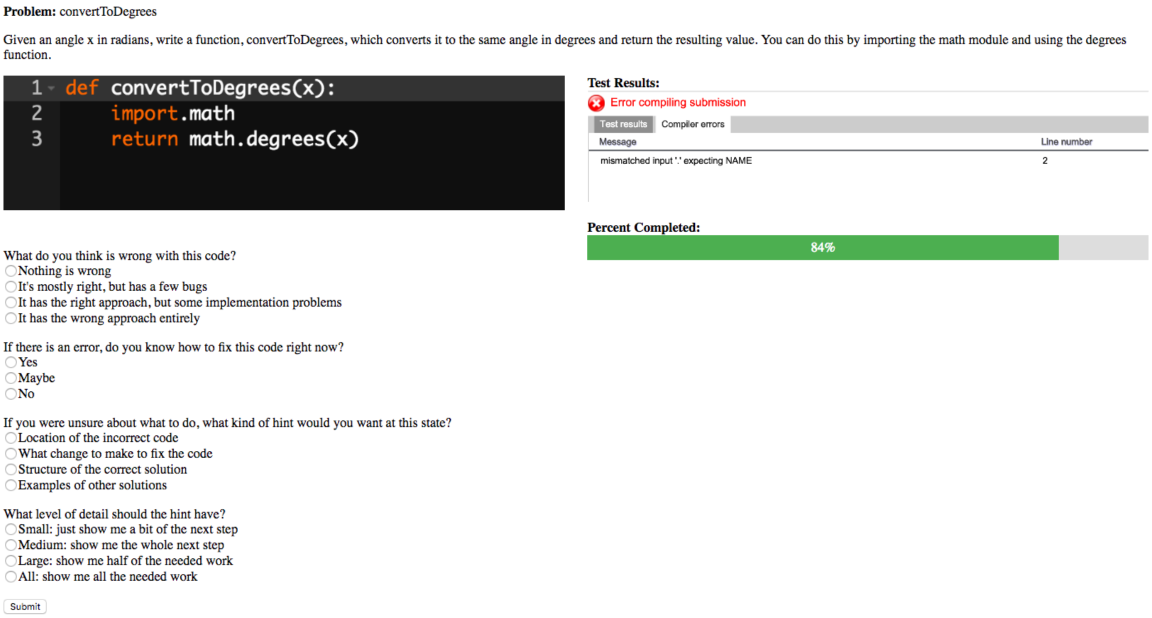
Methods

To run this study, I recruited people who had past experience with programming but (preferably) were not experts in Python, in order to replicate the experience of learning how to program without needing to teach total novices the basics. Users were recruited via the CMU participant pool and from the mailing list of HCI graduate students. The study was run in a lab in the Human-Computer Interaction Institute, where each user would spend up to one hour completing the study activities while I observed. Users were compensated with $10 for their participation.

At the beginning of the study (after signing the consent form), users completed a short screening task, where they were asked to explain the purpose of a short program. I took notes on their explanation and used these notes to create an independent ranking of the users' programming knowledge. Participants then completed a short survey where they gave self-reported measures of their programming knowledge, their grit level, and their use of help sources; the full survey is included in Appendix 3. Basic demographic information (gender and race) was gathered from the participant pool database.

Once this survey was complete, participants were asked to treat the rest of the study as a think-aloud experiment (Ericcson & Simon, 1998). In a think-aloud experiment, participants literally say what they're currently thinking as they do tasks, to help researchers understand the purpose behind their actions and how they view different tasks. When participants fall silent, the researcher reminds them to keep talking, to ensure a consistent stream of thought. I took notes on these comments for qualitative analysis.

The first half of the study consisted of several hint-selection exercises, which participants worked on for up to half an hour. An example page of this exercise is shown in Figure 51. In these exercises, participants were shown a problem statement, a piece of code, the compiler errors/test case results that the code produces, and how close to a solution the code is. They were then asked several questions:



Problem: convertToDegrees

Given an angle x in radians, write a function, convertToDegrees, which converts it to the same angle in degrees and return the resulting value. You can do this by importing the math module and using the degrees function.

```
def convertToDegrees(x):
    import.math
    return math.degrees(x)
```

Test Results:

Error compiling submission

Message	Line number
mismatched input '.' expecting NAME	2

Percent Completed:

84%

What do you think is wrong with this code?
○ Nothing is wrong
○ It's mostly right, but has a few bugs
○ It has the right approach, but some implementation problems
○ It has the wrong approach entirely

If there is an error, do you know how to fix this code right now?
○ Yes
○ Maybe
○ No

If you were unsure about what to do, what kind of hint would you want at this state?
○ Location of the incorrect code
○ What change to make to fix the code
○ Structure of the correct solution
○ Examples of other solutions

What level of detail should the hint have?
○ Small: just show me a bit of the next step
○ Medium: show me the whole next step
○ Large: show me half of the needed work
○ All: show me all the needed work

Figure 51: An example exercise from the first half of the study.

- What do you think is wrong with this code?
    - *Nothing* is wrong
    - It's mostly right, but has a few *bugs*
    - It has the right approach, but some *implementation* problems
    - It has the wrong *approach* entirely

- If there is an error, do you know how to fix this code right now?
    - Yes/Maybe/No

- If you were unsure about what to do, what kind of hint would you want at this state?
    - Location of the incorrect code
    - What change to make to fix the code
    - Structure of the correct solution
    - Examples of other solutions

- What level of detail should the hint have?
    - Small: just show me a bit of the next step
    - Medium: show me the whole next step
    - Large: show me half of the needed work
    - All: show me all the needed work

The first question was meant to be used as a representation of the perceived *error type* of the problem, to determine whether it was consistent across participants. The second question could be used to separate participants who already knew how to solve the problem (and did not need a hint) from those who did not. The third and fourth questions then asked the user to choose one of the hint types described in the previous section. I anticipated that several factors would lead to different hint types being chosen per problem. However, it is not certain that student choices will follow any discernible pattern, as a previous study on student choices between error-specific feedback and next-step hints could find no correlation between students' code states and their feedback requests (Gross & Pinkwart, 2015).

Once the participant had made their choices and clicked Submit, a pre-generated hint was given to them based on their choices and was shown alongside the code and other information on the following page. Students were then asked these follow-up questions:

- Do you think this hint would help you move forward in solving this problem?
  - Yes/Maybe/No

- Does the level of detail match what you were expecting?
  - Yes/Maybe/No

- If there's an error in the code, do you understand how to fix it now?
  - Yes/Maybe/No

The first two questions were used to determine whether the actual produced hints matched the users' expectations of what they should get. The final question was used to determine whether certain hints actually helped; that is, whether they could help students who did not know how to solve the problem before find the bug and the fix.

Some students, upon receiving an unsatisfactory hint, instead chose to move back in the interface, choose a different type of hint, and review that hint instead. I allowed this behavior (as it helped me understand which types of hints were more helpful), and I include this backtracking action in the dataset for analysis.

The second part of the study consisted of participants writing code to solve simple programming problems in Cloudcoder, with hint support, while thinking aloud. This part allowed me to observe how the users interacted with actual hints while problem-solving. Problems were sorted based on difficulty so that all students would eventually reach a problem that would cause them to struggle. Only next-step hints were provided for this part due to implementation constraints. Students worked for the remaining time in the hour, attempting to complete as many of the problems as they could.

Results

Overall, 28 participants were recruited (8 female, 20 male); these participants ranged greatly in programming capability, making it possible to explore help-seeking behavior across of range of experience levels. The user survey results are shown in Table 20. Using the screening task as a separate indicator of programming knowledge, I found that 13 of the participants fully understood the code and understood its algorithmic purpose; 7 understood the program flow but missed some algorithmic components, and the remaining 8 struggled to understand the program, with three failing to grasp the main points at all.

Factor	Average Result
Researcher-Rated Python Knowledge (1 to 7)	5.32
User-Rated Programming Knowledge (7-Pt Likert Scale)	4.25
User-Rated Python Knowledge (7-Pt Likert Scale)	2.79
Used Python Before?	54%
User-Rated Grit Ranking (7-Pt Likert Scale)	5.54
Use of Practice Problems (7-Pt Likert Scale)	5.68
Use of Office Hours (7-Pt Likert Scale)	4.57
Use of Online Materials (7-Pt Likert Scale)	6.21

Table 20: Summary statistics for the user survey.

In the first half of the study, a full 34 exercises were included in the dataset, but I did not expect any student to finish all of them, and indeed, no student did. Students completed an average of 11.5 exercises, with a minimum of 3 and a maximum of 21; altogether, there is data for 321 exercise attempts, with 16 additional backtracking events. In the second half of the study, 15 problems were made available on Cloudcoder, again with the purpose of providing more than enough problems to fill the allotted time. Two of the participants were unable to do this part, as server problems resulted in Cloudcoder going offline; of the remaining 26, participants completed an average of 5.6 problems, with a minimum of 1 and a maximum of 11, for a grand total of 145 completed exercises and 16 incomplete exercises. Users asked for 9.19 hints on average during this section, with a minimum of 0 and a maximum of 28, for a total of 239 hints.

When analyzing the data, I found that one user did not vary their responses per question (they classified every error as a bug, and requested small next-step hints every time); I therefore remove this user from analyses. All other users exhibited variance in the bug type they chose, and most tried different levels of detail or hint types across exercises. Overall, users requested 52 location hints, 134 next-step hints, 74 structure hints, and 68 example hints; 100 of the hint size requests were small, 113 were medium, 53 were large, and 62 were all. 16 of the states were classified as having nothing wrong, while 132 were classified as having a bug, 107 were classified as having implementation errors, and 73 were classified as having the whole approach wrong. Therefore, in general, students were biased towards smaller next-step hints and thought that the states did have errors, though they generally thought the errors were smaller as well. It is worth noting here that the 16 states that were classified as having nothing wrong did, in fact, have bugs. Most of these classifications (10/16) occurred on the first problem, which had a whitespace error that was difficult to detect. Also, the bug-type classification was subjective, as there was no correct response for what type of bug a program had. I was primarily interested in the users' impressions of bug severity.

Quantitative Analysis

My first research question asked whether the types of hints users requested changed based on the users' own personal characteristics; for example, do more experienced students want less detail in hints than novice students? I ran a generalized linear model controlling for bug type and whether the student could already solve the problem that checked all the variables collected in the survey, as well as the researcher rating of the student's Python knowledge and gender. I removed entries where the bug type was labeled as 'Nothing' for this analysis, as it does not make sense to choose hints when nothing is wrong.

First, I ran the model of the type of hint chosen. I ordered the four hint types by the amount of content they provide (location, next-step, structure, and solution), so that the resulting model would show positive factors associated with more contentful hint types and negative factors associated with less content. The model (shown in Table 21) had surprisingly clear results. Students were likely to ask for more contentful hints if they had not used Python before, had lower grit, and often used office hours while help-seeking. In other words, more novice students wanted more content in their hints, while stronger students wanted less. It is also worth noting that approach bug-types were associated with more contentful hints, and women apparently requested less contentful hints.

	Estimate	Std. Error	z value	Probability
Intercept	**4.2822**	**1.4314**	**2.992**	**0.0028**
Bug Type: Bug	0.3217	0.7460	0.431	0.6663
Bug Type: Implementation	0.7613	0.7795	0.977	0.3287
Bug Type: Approach	**2.1240**	**0.9274**	**2.290**	**0.0220**
User-Rated Programming Knowledge	-0.2258	0.1489	-1.517	0.1293
Used Python	**-0.8917**	**0.3839**	**-2.323**	**0.0202**
Grit Ranking	**-0.6980**	**0.1960**	**-3.561**	**0.0004**
Office Hour Use Ranking	**0.5556**	**0.1373**	**4.045**	**< 0.0001**
Gender: Female	**-1.0446**	**0.4959**	**-2.106**	**0.0352**

Table 21: A logistic model predicting the contentfulness of the hint type, where contentfulness varied from location = 0, next-step = 1, … to solution = 3. Original factors: bug type, whether the user could solve the problem before, researcher-rated Python knowledge, user-rated Python knowledge, user-rated programming knowledge, whether the user had used Python before, grit ranking, student use of practice resources, student use of office hours, student use of online materials, and gender.

Next, I ran a similar logistic model on the level of detail chosen for the hint, this time including hint type as a parameter. As the levels of detail were already ordered (small, medium, large, and all), I did not have to impose a new ordering. The resulting model can be found in Table 22. This time, students with lower reported programming knowledge and higher office hour use request higher detail in hints; again, this seems to indicate that more novice students want more detail. However, students with higher grit also request more detail, which is surprising. I again see that approach-type bugs require the most detail (though bug and implementation bugs also require more detail), and that women request less detail; also, users who were less capable of solving the problem originally were likely to request more detail.

	Estimate	Std. Error	z value	Probability
Intercept	**-2.5220**	**1.2093**	**-2.085**	**0.0370**
Bug Type: Bug	**1.7785**	**0.8062**	**2.206**	**0.0274**
Bug Type: Implementation	**1.6601**	**0.8081**	**2.054**	**0.0399**
Bug Type: Approach	**2.9629**	**0.8930**	**3.318**	**0.0009**
Can Solve Before Ranking	**-1.0615**	**0.2442**	**-4.347**	**< 0.0001**
User-Rated Programming Knowledge	**-0.2682**	**0.1254**	**-2.140**	**0.0324**
Grit Ranking	**0.4316**	**0.1754**	**2.461**	**0.0139**
Office Hour Use Ranking	**0.4886**	**0.1130**	**4.325**	**< 0.0001**
Gender: Female	**-1.4361**	**0.4195**	**-3.423**	**0.0006**

Table 22: A logistic model predicting the level of detail of a hint, where detail varied from small = 0, medium = 1, … to all = 3. Original factors: hint content type, bug type, whether the user could solve the problem before, researcher-rated Python knowledge, user-rated Python knowledge, user-rated programming knowledge, whether the user had used Python before, grit ranking, student use of practice resources, student use of office hours, student use of online materials, and gender.

I also investigated whether these user factors would predict actual use of hints in the second half of the study. I ran a Poisson model predicting the number of hints a student asked for in the second half, excluding the two students who did not get to complete it. The resulting model can be found in Table 23. This model supports the findings of the previous ones, as it shows that having less programming and/or Python knowledge results in asking for more hints. It also shows that students who completed more problems asked for more hints, and that women asked for fewer hints.

	Estimate	Std. Error	z value	Probability
Intercept	**3.0856**	**0.6975**	**4.424**	**< 0.0001**
Researcher-Rated Python Knowledge	**-0.2167**	**0.0687**	**-3.153**	**0.0016**
User-Rated Programming Knowledge	**-0.2119**	**0.0552**	**-3.841**	**0.0001**
User-Rated Python Knowledge	**-0.1727**	**0.0657**	**-2.628**	**0.0086**
Grit Ranking	0.1321	0.0811	1.629	0.1033
Practice Problem Use Ranking	**-0.2751**	**0.0858**	**-3.208**	**0.0013**
Online Material Use Ranking	**0.2518**	**0.1130**	**2.229**	**0.0258**
Gender: Female	**-0.3426**	**0.1618**	**-2.117**	**0.0342**
# Part Two Problems Solved	**0.1467**	**0.0526**	**2.790**	**0.0053**

Table 23: A Poisson model predicting the number of hints a user requested in the second half of the study. Original factors: researcher-rated Python knowledge, user-rated Python knowledge, user-rated programming knowledge, grit ranking, student use of practice resources, student use of office hours, student use of online materials, gender, number of Part two problems solved, and number of Part two problems attempted but not solved.

Next, I wanted to know whether the types of hints users requested changed based on the type of error they had encountered. First, I investigate whether the bug types which were provided were consistently applied to problems; in other words, was it clear to students what each bug type meant? I focus only on problems completed by at least five students, which provides 15 problems. Of these fifteen problems, only two had a single bug type that was chosen by at least 80% of the participants (one a bug bug-type, the other an approach bug-type). Eleven of the remaining problems reached this cutoff when the second most common bug-type was added in; nine of these were tied between Bug and Implementation errors, while two were tied between Approach and Implementation errors. Therefore, it seems that choice of bug type was at least partially subjective, possibly due to confusion about what an Implementation error was.

Since there were not consistent results between individual error types, I instead view them as a spectrum of error levels, with ‘Nothing’ being the lowest level and ‘Approach’ being the highest. Mapping these bug types to the numbers 0 to 3 gives a numerical rating of how severe the error type is. Now I want to see whether I can predict how severe the error is perceived to be based on the distance from the state to the solution (shown in the exercise as Percent Completed). I averaged the error type numbers for each problem to get an average rating (where the lowest-rated problem had a rating of 1, and the highest-rated problem had a rating of 2.73). I found a negative correlation between the level of bug type chosen and the problem’s percent completed ($r = -0.50$, $p < 0.05$); in other words, code that was further from the correct solution was labeled as having more complex bugs. At this problem level, more contentful hint types were also correlated with more complex bugs ($r = 0.55$, $p < 0.01$), which matches the result seen in the model that included user data.

Finally, I wanted to know which types of hints were considered most helpful, to provide more information on the learning potential of hints. To do this, I use two metrics: change in the participant's rating of whether they could solve the problem (i.e., learning), and the participant's rating of the hint's helpfulness. I also include data from students who backtracked, by marking these attempts as not being able to solve the problem and not finding the hint helpful (which I assume to be true, as participants wanted a different hint). It is important to note upfront that these metrics are *not* true indicators of learning, as they are based on self-report and do not test users' abilities to actually fix bugs. However, they do indicate how hint format impacts student perception.

Linear models found that both learning and helpfulness were positively correlated with larger detail sizes, learning was correlated with solution-type hints, and already being able to solve a problem was correlated with higher ratings of hints, as is shown in Tables 24 and 25. These models also showed that students who originally did not see a bug (the nothing bug-type) learned more, but thought the hints were less useful.

	Estimate	Std. Error	t value	Probability
Intercept	*-0.2807*	*0.1643*	*-1.708*	*0.0886*
Bug Type: Implementation	0.1249	0.1275	0.980	0.3281
Bug Type: Bug	0.1081	0.1249	0.865	0.3876
Bug Type: Nothing	**0.6716**	**0.2264**	**2.967**	**0.0032**
Hint Type: Next-Step	0.1221	0.1350	0.904	0.3665
Hint Type: Structure	0.0565	0.1489	0.380	0.7044
Hint Type: Solution	**0.4257**	**0.1567**	**2.717**	**0.0069**
Hint Size	**0.1773**	**0.0425**	**4.174**	**< 0.0001**

Table 24: A linear model predicting the delta of student's report that they could solve the problem, before and after receiving a hint. Original factors: bug type, hint type, and hint size. Adjusted R-squared: 0.08899

	Estimate	Std. Error	t value	Probability
Intercept	**1.0413**	**0.1889**	**5.514**	**< 0.0001**
Can Solve Before Ranking	**0.1853**	**0.0579**	**3.200**	**0.0015**
Bug Type: Implementation	0.0020	0.1258	0.016	0.9875
Bug Type: Bug	-0.0502	0.1224	-0.410	0.6819
Bug Type: Nothing	**-0.5665**	**0.2375**	**-2.385**	**0.0177**
Hint Type: Next-Step	0.0324	0.1321	0.245	0.8064
Hint Type: Structure	-0.2370	0.1458	-1.625	0.1052
Hint Type: Solution	*0.2625*	*0.1535*	*1.711*	*0.0881*
Hint Size	**0.1750**	**0.0424**	**4.127**	**< 0.0001**

Table 25: A linear model predicting how helpful students will rank a hint as being. Original factors: whether they could solve before, bug type, hint type, and hint size. Adjusted R-squared: 0.1321

I also analyzed the four different types of hints, to see if they differed in how they were perceived by students. Two of the hint types, location hints and structure hints, did not generally see a significant difference in learning according to a paired t-test (location: 2%, $p > 0.1$; structure: 5%, $p > 0.1$), and were only counted as moderately helpful by participants (location: 65% helpful; structure: 60% helpful). On the other hand, next-step hints and solution hints both showed significant improvements in learning before and after hints were shown (next-step: 10% improvement, $p < 0.01$; solution: 24%, $p < 0.001$). Next-step hints were counted as helpful 75% of the time, while solution hints were counted as helpful 86% of the time.

Qualitative Analysis

In addition to the quantitative analysis, I also have copious notes on real-time student interactions during the study. I reviewed these notes to identify useful commonalities between participants and to look for areas where hint representation could be improved.

First, I looked for indicators of what kind of help students need, based on participant comments and actions. A few participants focused primarily on fixing the code, saying things like "I don't think that's very helpful; it does get rid of the error message, but does it fix the function?" Other participants wanted to know not just *how* to fix the code, but also *why* it had to be done a certain way. In particular, one person stated: “The hint should be something... the compiler can't tell. Maybe the message should just say whitespace matters?". Another participant, after receiving a solution hint, said "I feel like I wouldn't understand why this doesn't work, but I'd know it doesn't work".

More students wanted to use examples in order to understand how code worked. This attitude was particularly evident in the second half of the study, where many of the participants asked if they could go back to the first part to look at examples of Python code when tasked with writing it themselves. However, comments about examples occurred during the first half as well. As one student said, "I want an example of how... I don't want other solutions. I want to be able to google, to see how to make it a string". Other students, while trying to think from the point of view of a student who had created the erroneous state, thought examples would be needed to help such a confused student. One said, "I'd guess I want examples in this case, ‘cause... if I thought the len of a number was a reasonable thing to do, I'd want to look at another example". Another student posited: "I'd want structure [hint]... but would they want structure? They'd have to recognize they're fundamentally wrong to want structure".

Overall, the differences between these two points of view seem best summarized by the following quote from one participant: "A hint is what you do when you're stuck. Google is for when you know what to do, but not how to do it". Understanding the difference between a student who knows *what* to do but not *how* to code it and a student who knows *how to code* but not *what to do* may be essential in formatting useful hints. This difference was shown clearly in one problem in Part 2 where participants had to use the `in` operator, which most of them did not already know. Even when participants got hints that specifically told them to add the expression `‘left side’ in ‘right side’`, they did not always follow the hint’s advice; when questioned about this, one participant said he hadn’t even noticed the `in`, as he was expecting something more like `.contains()`. He thought that having an example of how to use `in` would be more useful.

Many participants also commented about the amount of information provided by hints. There seemed to be a general assumption that students should be provided with the smallest hint possible, but also an understanding that, sometimes, more detail would be necessary. This assumption appeared in complaints that small hints did not always provide enough information: "Oh... that didn't help. I think I need more than a small amount of help.", and "Part of me wants a location of the incorrect code, but that implies that I think the error will be obvious. That might be my first hint, though.". It also appeared in complaints that large hints were too large: "This is probably too detailed- it's not a hint, it's just the solution", "It seems like I would never want what change to make, that's just giving me the answer", and “I feel like this is cheating, because it gave me alternative correct solutions, didn't that give me the answer?"

Some participants explicitly stated that they wanted to start with smaller hints, but progress to larger ones if needed. One said, "Once I'd fixed that section, I'd want a bigger hint, but right now I'd want a small hint to fix this". Another mentioned, "I want the smallest amount of help possible, so I can figure it out... but that's under the assumption that I can figure it out". One particularly intriguing comment came from a participant who said "Most of my mistakes are syntactic, that was more logical. Once I know it's logical, I'll want a more detailed hint.”

Finally, I looked for places where participants expressed confusion about hints to better understand how I could improve them. First and most blatantly, several students mentioned code obfuscation as being confusing and unnecessary (in both next-step hints and structure hints). Several comments mentioned this idea directly: "The notation for [the structure] was hard for me to read, at least, and was rarely what I was looking for", "Only when I got the full statement did I know what was going on", and “If you're giving a hint, why not make it entirely clear?". Some participants mentioned that structure hints could be improved if individual values were better represented: “I'd expect the variables to be numbered". However, students generally thought that next-step hints should provide full detail: "I think the whole thing would work, because I didn't know about casting, the hint should tell me that".

Participants also expressed opinions on hints that recommended they delete or replace work that had already been done. One student said, "I thought it was telling me to use a whole different approach, I didn't want to start again" when asked why he wasn’t following a hint. Another scoffed at a delete hint, saying, "Well, I don't think that's a hint". These comments came from participants working on their own code in Part 2; participants in Part 1 who were rating large replace next-step hints did not seem quite so offended, but they were sometimes confused. One participant said, “I sort of understand the old code, but it told me to replace it with new code, which I don't understand what it's doing", while another said, "I guess if I follow this, I could get the right solution, but I don't understand much of what's going on".

Finally, participants also expressed confusion at the term ‘column’ (used in location and next-step hints), as many of them did not recognize what that term meant. This confusion could be remedied by either having columns explicitly shown in the IDE (as many IDEs do), or by using other location indicators, like highlighting or bolding the relevant piece of code. The potential use of this approach was demonstrated by one student who highlighted part of her code before pressing Hint, and others who suggested that highlighting broken indentation would be more useful than the textual hints. I also noticed that the students with the least incoming programming knowledge struggled greatly with applying the hints; two participants encountered syntactic trouble when they tried to apply next-step hints that included filler strings (by not putting quotes around the filler), while another had trouble finding the right location in the code to apply changes, as he had trouble distinguishing ‘right side’ from ‘left side’. Differently formatted hints might improve the experience of students like them.

Discussion

In this study, I observed a few strong connections between a participant’s knowledge level and the hints they requested, as well as preferences for different types of hints for different error types. First, more experienced students (via several factors) wanted less detail, while less experienced students wanted more; this could be viewed as an expertise reversal effect, which states that learners at different levels of knowledge need different types of assistance (Kalyuga et al, 2003). This information could be included in a student model that could change the kinds of hints provided to students. Additionally, only next-step and solution hints were associated with a view of improvements in learning; therefore, it makes sense to focus on these types of hints when providing feedback to students.

Alternatively, based on participant comments, ITAP could construct multiple levels of hints that would provide increasing amounts of information each time a student asked for more; this approach would be akin to the method used in traditional ITSs, where hints are often provided in a point-teach-bottom-out sequence (VanLehn, 2006). However, instead of using varying levels of detail in next-step hints (as I did in the pilot study), ITAP could instead provide different *types* of hints, starting with location, moving on to next-step, and bottoming out at the personalized solution. The type of hint could also be adjusted based on the distance from the solution, as I found that participants wanted more contentful hints when they encountered approach-style bugs (possibly because more improvements needed to be made) and that approach-style bugs were associated with larger distances from the correct code.

The idea of providing examples that do not necessarily give away the solution came up several times throughout the study, which leads to a new question: would it be possible to construct hints for the explicit purpose of providing targeted examples? Integrating worked examples into data-driven tutors has been investigated before (Liu, Mostafavi, & Barnes, 2016), though the worked examples were provided upfront instead of having students attempt problem-solving first, and a method for optimal selection of examples has been proposed by (Gross et al, 2014b). However, it is not clear how these hints and examples might differentially affect learning; further study will be needed to tease the differences of these two feedback types apart. For now, I will continue focusing on data-driven hints, but example hints provide an interesting avenue for future work.

Finally, I must address the concerns raised by students about hints which suggest that they change large portions of code and about hint obfuscation. The first problem occurs when students ask for hints on code that is significantly different from any correct state that has been seen before. This problem may occur on code states which are attempting to reach new solution states, but more often it occurs in code that is confused and hard to rescue. Perhaps a better approach to providing hints in this context would be to provide them with a simpler problem or worked example that would help them learn more material. The second problem is easier to address, as I have now seen it raised several times. Now that students in the classroom study and students in the usability study have complained about code obfuscation, it is clear that it must be changed to something easier to understand. It may not be wise to remove it altogether (as there are times when a teacher may wish to not give away a large amount of code), but ITAP must minimize the amount of filler code used in hints in the future.

5. Measuring the Effect of Hints on Student Learning

In this final study, I address an important question about ITAP-generated hints: do they have any impact on student learning? I performed a randomized crossover classroom experiment that could directly measure the effects of receiving hints on student performance on related coding problems. The main difference between this study and the previous classroom studies is that in this study, student participation was required, where past experiments have encouraged voluntary participation. In this chapter I discuss the conduct and results of this classroom study.

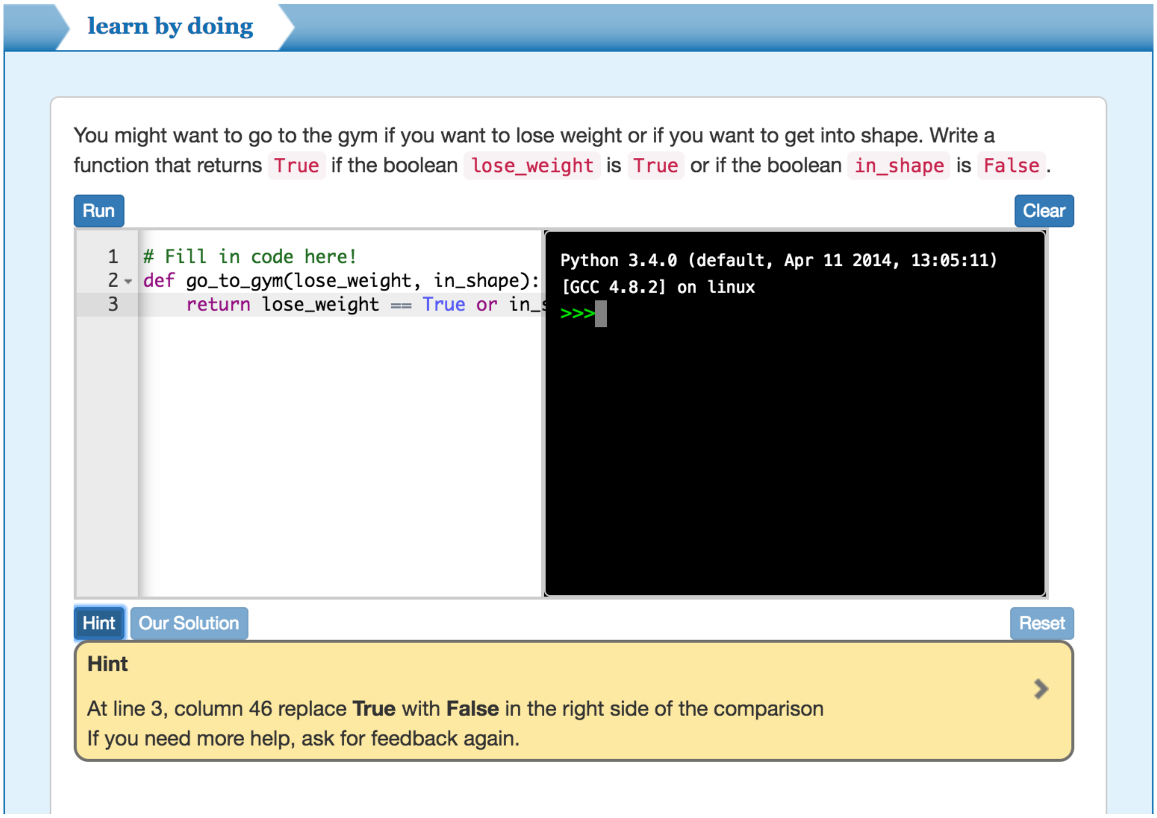
In this study, I anticipate that hints may have an impact on time-on-task and/or learning. There is some support for each of these hypotheses in the previous literature. For the time-on-task hypothesis, previous work on providing novices with detailed messages about why programs behaved in certain ways resulted in the programmers completing their work much more efficiently (Ko & Myers, 2008). Additionally, an evaluation of the Hint Factory showed that having hints helped students complete some problems in a third of the time, when compared to students without hints (Eagle & Barnes, 2013), and a study of an automatic hint generation system for programming found that students who had hints available finished problems in less time (Kim et al, 2016).

General research into hints in ITSs delves more into the effect hints have on learning. One study found that the amount of time a student spends with a bottom-out hint correlates with learning (Shih, Koedinger, & Scheines, 2011), potentially due to self-explanation benefits. Another study comparing error-flagging to hints showed that hints resulted in marginally higher levels of learning (Paquette et al, 2012). However, as was shown in the previous chapter, different students have different experiences with hints; in particular, previous studies have shown that more novice students tend to benefit from more assistance, while more expert students benefit from less (Razzaq, Heffernan, & Lindeman, 2007), so the effects of hints on learning may not be clear.

Updated Implementation

Since this study had required participation, instead of voluntary, I had to ensure that the hint generation system and the online coding modules would be able to support the full class of students accessing it at the same time. As this study required supporting up to 200 sessions at once, the original system of using individual servers to run the online IDE and ITAP instance would not suffice. Therefore, I decided to transition both the frontend IDE and backend ITAP system to work at greater scale by integrating coding problems into the Open Learning Initiative (OLI) system and transitioning the ITAP backend to be supported by Django, a popular Python web framework designed to make systems scalable.

First, a brief background of the OLI system (Lovett, Meyer, & Thille, 2008). OLI serves as an online educational resource that provides interactive online course resources over a variety of topics, including programming. It combines textual lessons with tutor activities (such as the one shown in Figure 52) that allow students to practice as they learn. Some of these tutor activities were made with the tutor-authoring system CTAT (Aleven et al, 2006) and thus come equipped with Submit and Hint buttons. The programming problems used in OLI prior to this study were also built with Submit and Hint buttons, where hints were written for each problem at the problem's creation, and therefore would not change based on student input.



You might want to go to the gym if you want to lose weight or if you want to get into shape. Write a function that returns `True` if the boolean `lose_weight` is `True` or if the boolean `in_shape` is `False`.

```
1 # Fill in code here!
2 def go_to_gym(lose_weight, in_shape):
3     return lose_weight == True or in_
```

```
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
>>>
```

Hint

At line 3, column 46 replace **True** with **False** in the right side of the comparison
If you need more help, ask for feedback again.

Figure 52: An example of an ITAP-enabled tutor activity in the OLI system.

I modified the programming problems in a specific module of OLI's Principles of Computing course to instead provide hints using the ITAP system by submitting the student ID, problem name, and current code to an ITAP server on each student's submission and sending back a JSON object containing the hint message. This was accomplished by designating the ITAP system as a new type of feedback engine within OLI. The OLI project ran its own instance of ITAP on its servers; as the project already supports hundreds of students using the main system every day, it was able to meet the demand easily.

I also needed to make certain modifications to the ITAP system during this transition period. First, the original version of ITAP stored data in csv files, which was not sustainable for high-volume use. I transitioned to a database format for data storage by using Django. Using Django required that I also transition the ITAP codebase from Python2 to Python3, which necessitated many changes within the codebase (and several core changes to the built-in AST format). These changes did not impact the underlying algorithm, but they did lead to several bugs that were not caught until after the study was run, leading to less-than-optimal hint generation in some cases. The primary bugs that might have affected the students caused incorrect variable name display and sometimes caused the algorithm to choose less-efficient goal states.

Classroom Study 2: How Do Hints Affect Learning?

The following section describes the final study of this thesis, conducted in the spring of 2017.

Research Questions

This study was designed to answer two primary research questions. First, does seeing ITAP-generated hints during practice lead to increased learning on similar coding tasks? Second, does seeing ITAP-generated hints during practice lead to faster completion of practice problems?

Methods

To answer these research questions, I designed a study that would require students to complete practice problems during a time of optimal learning. The study consisted of five stages: pretest, practice 1, midtest, practice 2, and posttest. The pretest, midtest, and posttest each consisted of three questions (one each of easy, medium, and hard difficulty). While students could run test cases on their code in these problems, they could not ask for hints. These sections were used to assess learning over time. The two practice sessions consisted of eight coding problems (three easy, three medium, and two hard), which could have hints enabled, depending on the student's condition. The content of the problems in the easy, medium, and hard categories were aligned across sections, so that learning could be measured directly across activities over time. The activities used are included in Appendix 1.

Students were assigned to one of two conditions upon attempting the first problem: hints-first or hints-second. Students in the hints-first condition received hint messages generated by ITAP alongside test case results in the practice 1 section (with every request instead of on-demand); students in the hints-second condition received them in the practice 2 section instead. In this way, I hoped to measure the effects of hints on learning between pretest and midtest and between midtest and posttest. I originally intended to test a new sequence of hints (based on the findings from the usability study), where students would be able to request more detailed hints after viewing the original ones; however, last-minute changes to the OLI interface resulted in the traditional next-step hints being used alone instead.

The study was conducted in 15-110: Principles of Computing, the introductory programming course for non-majors at Carnegie Mellon University. It occurred in weeks 7-8 of the course, directly before the first lab exam. Students were told by the main course instructor that they would receive 5% on the lab exam for either completing all the problems in the OLI module or spending at least two hours in the module (so that students would not be required to spend an undue amount of time in the system). 189 students used the system; of these students, all but seven met the requirement for full participation credit. One student asked that their data not be used in analysis, so I removed them from the dataset.

Results

When the study began, I asked students to complete the modules in order and also asked that they spend a maximum of ten minutes per pretest/midtest/posttest and 45 minutes per practice session, so that they would attempt all sections within two hours. I did not enforce these requests, which turned out to be a mistake, as many students completed activities out of order. Therefore, the following analyses will be done on subsets of students who completed the analyzed activities in the correct order (as causality cannot otherwise be discerned).

First, I attempted to analyze a subset of students who completed the pretest, practice 1, midtest, practice 2, and posttest in the correct order. Of the 188 students, 92 completed all the modules in the correct order. An additional nine students went back and forth between the pretest and practice 1 before opening the midtest, or went back and forth between the midtest and practice 2 before opening the posttest; I include these students, but only count work done on the pretest and midtest before the practices were attempted. Of the remaining 87 students, 65 did not complete one of the five modules, and the other 22 used other odd orderings that could not be redeemed.

Of the 101 students analyzed, 50 were in the hints-first condition and 51 were in the hints-second condition. Since every submission resulted in a hint (for students in the hint condition), I cannot provide the usual statistics on student use of hints; however, I can show how many submissions students made in each practice session as a surrogate metric. In the first practice session, students in the hints-first condition made 20.22 submissions on average, while students in the hints-second condition made 21.98 submissions on average. In the second practice session, students in the hints-first condition (now without hints) made 19.2 submissions on average, while students in the hints-second condition made 21.75 submissions on average.

To determine whether students were a) learning more or b) spending less time on practice, I primarily investigated two variables. First, I analyzed the number of problems that students got correct on the first attempt (controlling for the number of problems they attempted at all and performance on the previous test) in the midtest and posttest. Second, to measure time on task, I analyzed how much time was spent on the practice 1 and practice 2 sections, controlling for number of problems attempted in each.

Analysis of Learning

According to a repeated measures ANOVA there was a significant change in performance on first attempt from pretest to midtest $F(1,99) = 38.80$, $p < 0.01$; however, there was no significant effect of condition $F(1,99) = 2.04$, $p > 0.1$ nor interaction between condition and test time $F(1,99) = 1.51$, $p > 0.1$. A second repeated measures ANOVA from midtest to posttest detected no significant effects for test time $F(1,99) = 0.54$, $p > 0.1$, condition $F(1,99) = 0.80$, $p > 0.1$ nor interaction between the two $F(1,99) = 0.54$, $p > 0.1$. These results are shown in

Figure 53. I also found no significant effect of condition on time on task in practice 1 or practice 2 according to a linear model (Practice 1: 43.56min for the hints-first condition, 51.12min for the hints-second condition. Practice 2: 43.37min for the hints-first condition, 51.67min for the hints-second condition).

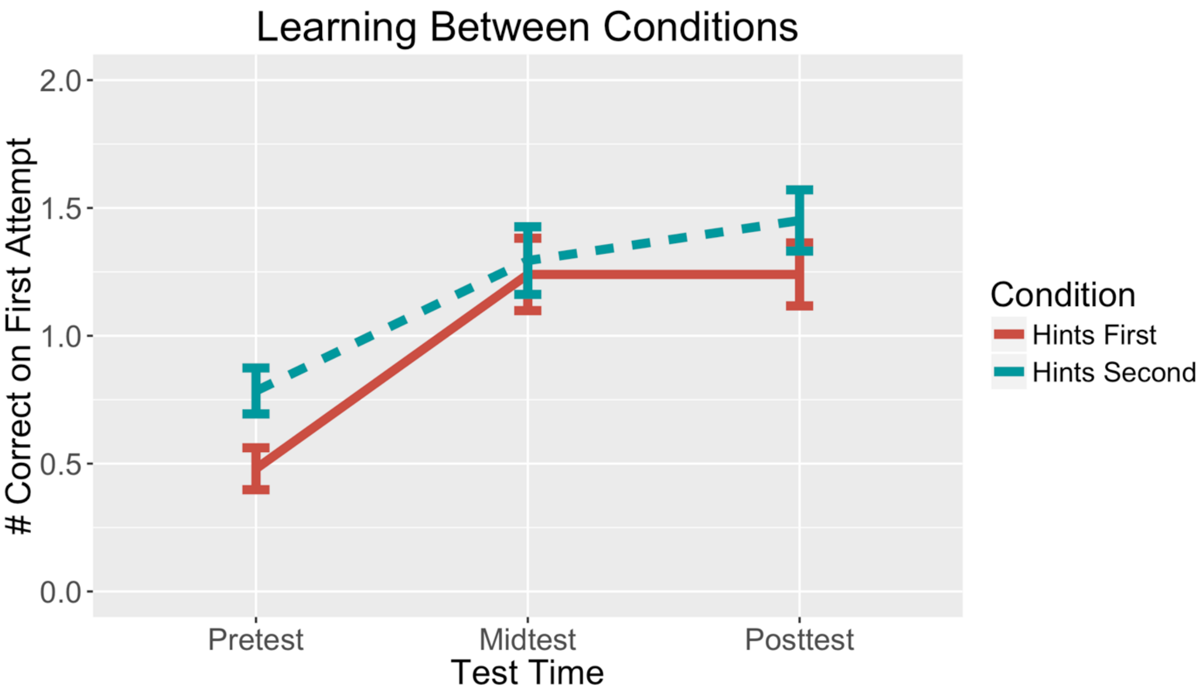


Figure 53: Learning from pre-to-mid-to-post test, in both conditions. Students in the two conditions start at significantly different levels of performance, but have achieved the same level of learning by midtest. The difference in learning between conditions is not significant according to a repeated measures ANOVA.

It is possible that I did not see an effect of condition on learning or time on task due to the reduced sample size. Therefore, I also looked at students who completed the pretest, practice 1, and midtest in the correct order. Of the 188 students, 122 completed these three sessions in the correct order originally. An additional ten students alternated between the pretest and practice 1 before moving on to the midtest; I include these students, but only use their initial work on the pretest for learning measurement. Of the remaining 56 students, 46 did not complete one of the three required elements, and the other 10 used other odd orderings that could not be redeemed.

Of the 132 students analyzed, 64 were in the hints-first condition, while 68 were in the hints-second condition. Again, I analyze how many submissions students made in the first practice session as a surrogate for hints requested; students in the hints-first condition requested 21.22 hints on average, while students in hints-second requested 21.91 hints on average.

A repeated measures ANOVA again showed a significant increase in learning from pretest to midtest F(1,130) = 53.85, p < 0.01, but no significant effect of condition F(1,130) = 0.53, p > 0.1 nor an interaction between the two F(1,130) = 2.07, p > 0.1. This result is shown in Figure 54. However, I did find a marginal effect of condition on time on task. A linear model predicting total time spent in practice 1 found that being in the hints-second condition was positively correlated with spending time on task, but only with marginal significance (p < 0.1), with an effect size of 7.35 minutes. The full model is shown in Table 26.

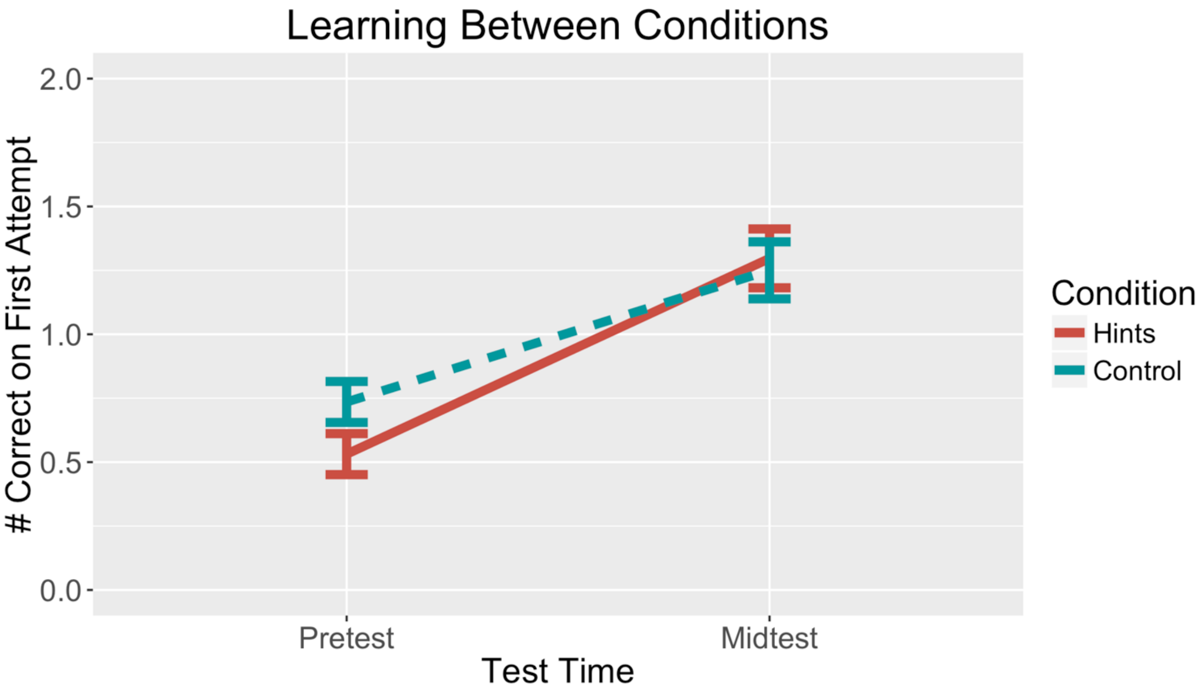


Figure 54: Learning from pretest to midtest was significant, F(1,130) = 53.85, p < 0.01, but learning between conditions was not.

	Estimate	Std. Error	t value	Probability
Intercept	-402.9856	802.3609	-0.502	0.6164
Condition: hints-second	*441.0321*	*238.7987*	*1.847*	*0.0671*
Practice 1 # Attempts	**319.9040**	**108.7219**	**2.942**	**0.0039**
Pretest Total Time	**0.8546**	**0.1873**	**4.561**	**< 0.0001**

Table 26: A linear model predicting time spent on practice problems in the first practice session. Original factors: condition, # attempts in practice 1, total time spent in pretest. Adjusted R-squared: 0.2198

To better understand where this change in time-on-task occurred, I plotted a density graph to show how much time different students were spending in practice 1 across the two conditions. This density graph is shown in Figure 55. This figure shows that most students in the hinted condition spent the recommended amount of time on the practice module (around 45 minutes), while more students in the control condition were likely to spend additional time on practice. In other words, having access to hints may have kept students from getting stuck and spending non-fruitful time on problems.

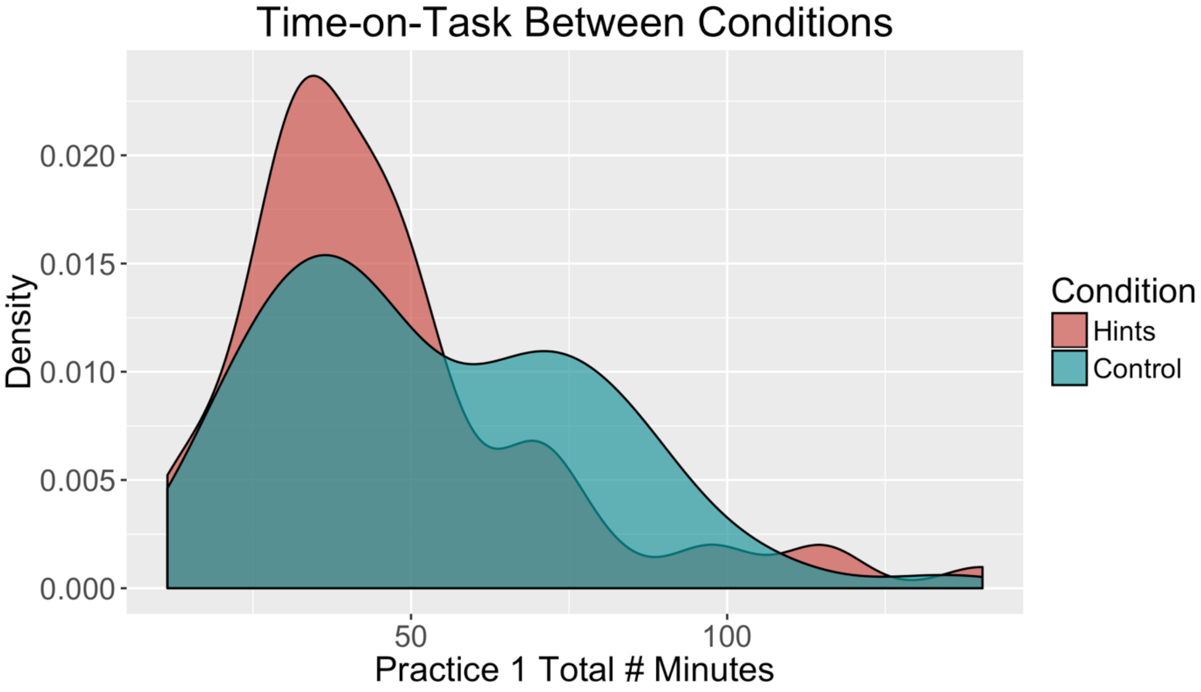


Figure 55: A density graph showing how time spent on practice was distributed across students between conditions.

Discussion

Based on the results, even though students learned from pretest to midtest, it seems fairly unlikely that the hints generated at the time of the study influenced learning. However, there does seem to be a possibility that having hints reduced the amount of time students needed to spend on practice by 13.7%, without negatively affecting learning outcomes. To determine whether hints can actually reduce the time needed to learn, I will need to run further studies to see if the marginally significant effect holds.

It is worth noting that the results found in this study may be weaker than one would typically expect due to several factors. First, as mentioned before, there were several bugs in the ITAP system at the time of the study, which resulted in some less-than-optimal hints. Second, some students reported latency issues during use of the hint system on OLI during times of heavy traffic; the OLI team addressed these concerns efficiently (so that the number of complaints stayed low), but this lag may still have impacted learning. Finally, this study occurred at a time of review, instead of initial learning, and the topics covered had all been taught in previous weeks. This delay may have resulted in different learning results than would be produced with a study conducted earlier in the learning process. However, all of these limitations would lead one to expect weaker effects due to hints, instead of working against my hypothesis.

6. Conclusions, Contributions, and Future Work

In this final chapter, I reiterate my findings from the previous chapters, present the contributions of this thesis, and discuss avenues for future work. As a reminder, my original goal was to extend the field of data-driven tutoring to work in the more complex domain of programming, to see whether this approach could provide useful next-step hints for novice programmers.

First, I built ITAP, the Intelligent Teaching Assistant for Programming, as a technical approach towards adapting data-driven hint generation to a new domain. This system extends work on state representation by normalizing away non-semantic variation, which assists the system in identifying only semantic differences between different student states. The system also enhances research on next-step identification by constructing new paths instead of using paths that had been seen before, which allows it to generate hints for states that have never been seen before. I found that ITAP could generate hints 100% of the time, and that it could even chain hints together to create personalized worked examples for almost all student submissions. I also found that the hints were usually generated quickly enough to be usable. Another analysis showed that ITAP independently improved in performance as it gathered more data, and that several problems were able to reach optimal solutions with a subset of data. These results showed that data-driven hint generation for programming was feasible, but they did not address whether the generated hints would be useful.

In initial classroom studies, I investigated how students interacted with hints and help-seeking, to better understand how hints should be presented. When practice and hints were made optional, I found no evidence of a significant effect on learning, though choosing to practice was associated with lower dropout rates. Surveying students showed that students with higher levels of grit worked on fewer problems but asked for more hints, and that students with stronger growth mindsets started more problems. Additionally, qualitative notes suggested that students wanted more clarity and detail in hint messages, which led to my next question: how should hints be presented?

I ran a usability study where I tested four different types of hints (location, next-step, structure, and example) across a variety of buggy programs. Across a range of participants, I found that less experienced students generally wanted more content and more detail in their hints. Furthermore, analysis showed that more detailed hints were more likely to help students debug their code and were considered more helpful. This result is at odds with a general belief, expressed by several participants including some novices, that initial hints should contain a low level of detail. I also found that approach-type bugs (which were furthest from the goal state) were more often associated with larger content types and higher levels of detail in requested hints. Altogether, the type of hint presented might ideally vary on several dimensions: experience level of the student, severity of the error, and how many hints have been requested already.

I also found that in many occasions, students preferred seeing examples to receiving hints on how to fix their programs. Students would occasionally go out of their way to find examples, by reviewing previous work that had been completed. It seems that when students are stuck, they are often in one of two situations: either they know what to do but not how to code it, in which case an example is most helpful, or they don't know what to do, in which case they want a hint. The example hints and next-step hints best fit these descriptions, and indeed, those two types of hints were the only two that resulted in significant learning according to student reports of their ability to fix bugs.

Finally, I tested the ability of hints to improve learning or time-on-task in a randomized control trial experiment. I found no evidence of an effect of hints on learning outcomes, but having hints appeared to result in students spending 13.7% less time on practice. In other words, it appears students learned more efficiently when they had access to hints. As this effect was found in a condition with non-optimal hints due to a few technical limitations, even stronger results may be expected in follow-up studies.

There are several limitations to the work presented in this thesis. First, on a technical level, ITAP has been tested with introductory-level problems which tend to be short and simple. In contexts with longer or more complex problems it is likely that hint generation would be less effective. ITAP also relies on having test cases to determine whether a code state is correct, and test cases may not be available for several types of problems (including problems with randomization and graphics). Other researchers are investigating how to apply data-driven hint generation in these domains (Price, Dong, & Barnes, 2016), and they are likely to improve on the work presented here.

Additionally, the studies conducted for this thesis all had methodological limitations. The optional studies likely had a skewed population of students opting in, most likely students who would already engage frequently with practice resources. The usability study relied on self-report to determine whether users could solve problems, which could easily be skewed based on user perceptions. Finally, the study on learning delivered hints with every feedback request, which would never occur in normal practice circumstances. I plan to address these limitations by conducting new studies in the future to validate the original results.

The main technical and practical contribution of this thesis has been the ITAP system itself. This data-driven approach has been adopted and improved by other researchers since the first publication of the algorithm (Piech et al, 2015a; Price, Dong, & Barnes, 2016), which may lead to continued improvements in the field of data-driven tutoring, and thus improved feedback for students in classrooms. ITAP has also proven useful in introductory programming courses at Carnegie Mellon University as a resource for practice exercises, and it can continue to be used as a resource in the future. In the realm of data-driven tutoring, this work has demonstrated that it is possible to provide targeted next-step hints for any state automatically, and that these hints improve over time as data is collected. It has also shown that ITAP-generated hints can help students complete practice problems in less time with the same learning results, which can lead to more efficient learning.

For learning science and human-computer interaction contributions, I have also provided insights into how students are interacting with help resources in introductory programming classes, which may lead to modifications in how feedback is provided in the future. For example, I found that all interviewed students reported that they received help from their peers in other classes, but that they had to rely more on TAs in programming, due to different collaboration policies. Given the finding that student stuck-states can be caused by knowing *what* to do but not *how* to do it, it may be possible to institute new collaboration policies that allow peers to share examples, but not fixes. This type of policy could allow students to practice coding independently while still receiving help from social circles.

I also demonstrated that there is a discrepancy between the level of detail that novice students need for learning and the level they think is appropriate. Hints with higher levels of detail were consistently reported to be more helpful and better for fixing bugs, yet users expected lower levels of detail. It may be possible to fix this discrepancy by separating the learning process from the process of assessment, so that novices can receive highly detailed feedback on practice problems without any qualms. Further work will be needed to determine if this approach works.

In future work, I plan to continue refining ITAP so that it might provide robust and useful hints in a variety of contexts. I also hope to eventually extend the system to work for additional languages by adapting the central algorithm to work on abstract ASTs, with different canonicalization and reification libraries for each language. Furthermore, I am interested in examining the relationship between examples and hints as types of feedback, to see how both affect learning. I also plan to extend my current analyses of these thesis studies to more closely examine whether students actually follow the recommendations of hints they receive, to see how much of a direct impact hints have. Finally, I plan to replicate the study results in future courses where I am the head instructor to make sure that the results are robust across different contexts. In particular, I plan to study the long-term effects of hint support during practice, to see if this kind of support impacts student learning or retention throughout a course.

References

Adam, A., & Laurent, J. P. (1980). LAURA, A System to Debug Student Programs. *Artificial Intelligence*, 15(1-2), 75-122.

Ade-Ibijola, A., Ewert, S., & Sanders, I. (2015). Introducing Code Adviser: A DFA-Driven Electronic Programming Tutor. In *Proceedings of the 20th International Conference on Implementation and Application of Automata* (pp. 307-312).

Aho, A. V., Sethi, R., & Ullman, J. D. (1986). *Compilers, Principles, Techniques, and Tools*.

Aleven, V., McLaren, B., Sewall, J., & Koedinger, K. (2006). The Cognitive Tutor Authoring Tools (CTAT): Preliminary Evaluation of Efficiency Gains. In *Proceedings of the 8th International Conference on Intelligent Tutoring Systems* (pp. 61-70).

Aleven, V., Roll, I., McLaren, B. M., & Koedinger, K. R. (2016). Help Helps, But Only So Much: Research on Help Seeking with Intelligent Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 26(1), 205-223.

Aleven, V., Stahl, E., Schworm, S., Fischer, F., & Wallace, R. (2003). Help Seeking and Help Design in Interactive Learning Environments. *Review of Educational Research*, 73(3), 277-320.

Antonucci, P., Estler, C., Nikolić, D., Piccioni, M., & Meyer, B. (2015). An Incremental Hint System for Automated Programming Assignments. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 320-325).

Baker, R. S., Corbett, A. T., Koedinger, K. R., & Wagner, A. Z. (2004). Off-task Behavior in the Cognitive Tutor Classroom: When Students Game the System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 383-390).

Barnes, T., & Stamper, J. (2008). Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems* (pp. 373-382).

Barnes, T., Stamper, J., Lehman, L., & Croy, M. (2008). A Pilot Study on Logic Proof Tutoring Using Hints Generated from Historical Student Data. In *Educational Data Mining 2008: 1st International Conference on Educational Data Mining, Proceedings* (pp. 197-201).

Bhatia, S. & Singh, R. (2016). Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. In *Proceedings of the 2nd Indian Workshop on Machine Learning*.

Blackwell, L. S., Trzesniewski, K. H., & Dweck, C. S. (2007). Implicit Theories of Intelligence Predict Achievement Across an Adolescent Transition: A Longitudinal Study and an Intervention. *Child Development*, 78(1), 246-263.

Bloom, B. S. (1984). The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher*, 13(6), 4-16.

Carter, A. S., Hundhausen, C. D., & Adesope, O. (2015). The Normalized Programming State Model: Predicting Student Performance in Computing Courses Based on Programming Behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 141-150).

Chang, P. P., & Hwu, W. W. (1989). Inline Function Expansion for Compiling C Programs. In *ACM SIGPLAN Notices* (Vol. 24, No. 7, pp. 246-257).

Choudhury, R.R., Yin, H., & Fox, A. (2016). Scale-Driven Automatic Hint Generation for Coding Style. In *Proceedings of the 13th International Conference on Intelligent Tutoring Systems* (pp. 122-132)

Computing Research Association (2017). Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006. http://cra.org/data/Generation-CS/

Corbett, A. T., & Anderson, J. R. (2001). Locus of Feedback Control in Computer-Based Tutoring: Impact on Learning rate, Achievement and Attitudes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 245-252).

Corbett, A. T., Koedinger, K. R., & Anderson, J. R. (1997). Intelligent Tutoring Systems. *Handbook of Human-Computer Interaction* (pp. 849-874).

Cummins, S., Stead, A., Jardine-Wright, L., Davies, I., Beresford, A. R., & Rice, A. (2016). Investigating the Use of Hints in Online Problem Solving. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale* (pp. 105-108).

Cutts, Q., Cutts, E., Draper, S., O'Donnell, P., & Saffrey, P. (2010). Manipulating Mindset to Positively Influence Introductory Programming Performance. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 431-435).

D'Antoni, L., Kini, D., Alur, R., Gulwani, S., Viswanathan, M., & Hartmann, B. (2015). How Can Automatic Feedback Help Students Construct Automata?. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2), 9.

D’Antoni, L., Samanta, R., & Singh, R. (2016). Qlose: Program Repair with Quantitative Objectives. In *Proceedings of the 28th International Conference on Computer Aided Verification* (pp. 383-401).

de Raadt, M., Hamilton, M., Lister, R., & Tutty, J. (2005). Approaches to Learning in Computer Programming Students and Their Effect on Success. In *Proceedings of the 2005 HERDSA Annual Conference* (pp. 407-414).

Dillenbourg, P. (1989). Designing a Self-Improving Tutor: PROTO-TEG. *Instructional Science*, 18(3), 193-216.

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic Test-Based Assessment of Programming: A Review. *Journal on Educational Resources in Computing (JERIC)*, 5(3), 4.

Duckworth, A. L., Peterson, C., Matthews, M. D., & Kelly, D. R. (2007). Grit: Perseverance and Passion for Long-Term Goals. *Journal of Personality and Social Psychology*, 92(6), 1087-1101.

Eagle, M., & Barnes, T. (2013). Evaluation of Automatically Generated Hint Feedback. In *Proceedings of the 6th International Conference on Educational Data Mining* (pp. 372-374).

Eagle, M., & Barnes, T. (2014). Data-Driven Feedback Beyond Next-Step Hints. In *Proceedings of the 7th International Conference on Educational Data Mining* (pp. 444-446).

Eagle, M., Johnson, M., & Barnes, T. (2012). Interaction Networks: Generating High Level Hints Based on Network Community Clustering. In *Proceedings of the 5th International Conference on Educational Data Mining* (pp. 164-167)

Edmison, B., Edwards, S. H., & Pérez-Quiñones, M. A. (2017). Using Spectrum-Based Fault Location and Heatmaps to Express Debugging Suggestions to Student Programmers. In *Proceedings of the Nineteenth Australasian Computing Education Conference* (pp. 48-54).

Elliot, A. J., & McGregor, H. A. (2001). A 2x2 Achievement Goal Framework. *Journal of Personality and Social Psychology*, 80(3), 501.

Ericsson, K. A., & Simon, H. A. (1998). How to Study Thinking in Everyday Life: Contrasting Think-Aloud Protocols with Descriptions and Explanations of Thinking. *Mind, Culture, and Activity*, 5(3), 178-186.

Folsom-Kovarik, J. T., Schatz, S., & Nicholson, D. (2010). Plan ahead: Pricing ITS Learner Models. In *Proceedings of the 19th Behavior Representation in Modeling & Simulation (BRIMS) Conference* (pp. 47-54).

Fossati, D., Di Eugenio, B., Ohlsson, S., Brown, C., Chen, L., & Cosejo, D. (2009). I Learn from You, You Learn from Me: How to Make iList Learn from Students. In *Proceedings of the 2009 Conference on Artificial Intelligence in Education: Building Learning Systems that Care: From Knowledge Representation to Affective Modelling* (pp. 491-498).

Freeman, P., Watson, I., & Denny, P. (2016). Inferring Student Coding Goals Using Abstract Syntax Trees. In *Proceedings of the 24th International Conference on Case Based Reasoning* (pp. 139-153).

Gegg-Harrison, T. S. (1992). ADAPT: Automated Debugging in an Adaptive Prolog Tutor. In *Proceedings of the Second International Conference on Intelligent Tutoring Systems* (pp. 343-350).

Glassman, E. L., Scott, J., Singh, R., Guo, P. J., & Miller, R. C. (2015). OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2), 7.

Glassman, E. L., Lin, A., Cai, C. J., & Miller, R. C. (2016). Learnersourcing Personalized Hints. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing* (pp. 1626-1636).

Gross, S., Mokbel, B., Hammer, B., & Pinkwart, N. (2012). Feedback Provision Strategies in Intelligent Tutoring Systems Based on Clustered Solution Spaces. *DeLFI 2012: Die 10. e-Learning Fachtagung Informatik* (pp. 27-38).

Gross, S., Mokbel, B., Hammer, B., & Pinkwart, N. (2014b). How to Select an Example? A Comparison of Selection Strategies in Example-Based Learning. In *Proceedings of the 12th International Conference on Intelligent Tutoring Systems* (pp. 340-347).

Gross, S., Mokbel, B., Paassen, B., Hammer, B., & Pinkwart, N. (2014a). Example-based Feedback Provision Using Structured Solution Spaces. *International Journal of Learning Technology*, 9(3), 248-280.

Gross, S., & Pinkwart, N. (2015). How Do Learners Behave in Help-Seeking When Given a Choice? In *Proceedings of the 17th International Conference on Artificial Intelligence in Education* (pp. 600-603).

Gulwani, S., Radiček, I., & Zuleger, F. (2016). Automated Clustering and Program Repair for Introductory Programming Assignments. *arXiv preprint arXiv:1603.03165*.

Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010). What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1019-1028).

Hattie, J., & Timperley, H. (2007). The Power of Feedback. *Review of Educational Research*, 77(1), 81-112.

Head, A., Glassman, E., Soares, G., Suzuki, R., Figueredo, L., D'Antoni, L., & Hartmann, B. (2017). Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@Scale* (pp. 89-98).

Heiner, C. (2008). A Preliminary Analysis of the Logged Questions that Students Ask in Introductory Computer Science. In *Proceedings of the 1st International Conference on Educational Data Mining*, (pp. 250-257).

Hicks, A., Dong, Y., Zhi, R., Cateté, V., & Barnes, T. (2015). BOTS: Selecting Next-Steps from Player Traces in a Puzzle Game. In *Proceedings of the 2nd International Workshop on Graph-Based Educational Data Mining*.

Hovemeyer, D., Hellas, A., Petersen, A., & Spacco, J. (2016). Control-Flow-Only Abstract Syntax Trees for Analyzing Students' Programming Progress. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 63-72).

Hume, G., Michael, J., Rovick, A., & Evens, M. (1996). Hinting as a Tactic in One-on-One Tutoring. *The Journal of the Learning Sciences*, 5(1), 23-47.

Jeuring, J., van Binsbergen, L. T., Gerdes, A., & Heeren, B. (2014). Model Solutions and Properties for Diagnosing Student Programs in Ask-Elle. In *Proceedings of the Computer Science Education Research Conference* (pp. 31-40).

Jin, W., Barnes, T., Stamper, J., Eagle, M. J., Johnson, M. W., & Lehmann, L. (2012). Program Representation for Automatic Hint Generation for a Data-Driven Novice Programming Tutor. In *Proceedings of the 11th International Conference on Intelligent Tutoring Systems* (pp. 304-309).

Kaleeswaran, S., Santhiar, A., Kanade, A., & Gulwani, S. (2016). Semi-Supervised Verified Feedback Generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 739-750).

Kalyuga, S., Ayres, P., Chandler, P., & Sweller, J. (2003). The Expertise Reversal Effect. Educational Psychologist, 38(1), 23-31.

Kennedy, K. (1979). *A Survey of Data Flow Analysis Techniques.*

Keuning, H., Jeuring, J., & Heeren, B. (2016). Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 41-46).

Kim, D., Kwon, Y., Liu, P., Kim, I. L., Perry, D. M., Zhang, X., & Rodriguez-Rivera, G. (2016). Apex: Automatic Programming Assignment Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (pp. 311-327).

Ko, A., & Myers, B. (2008). Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering* (pp. 301-310).

Koedinger, K. R., & Aleven, V. (2007). Exploring the Assistance Dilemma in Experiments with Cognitive Tutors. *Educational Psychology Review*, 19(3), 239-264.

Koedinger, K. R., Brunskill, E., Baker, R. S., McLaughlin, E. A., & Stamper, J. (2013). New Potentials for Data-Driven Intelligent Tutoring System Development and Optimization. *AI Magazine*, 34(3), 27-41.

Kulik, J. A., & Kulik, C. L. C. (1988). Timing of Feedback and Verbal Learning. *Review of Educational Research*, 58(1), 79-97.

Lan, A. S., Vats, D., Waters, A. E., & Baraniuk, R. G. (2015). Mathematical Language Processing: Automatic Grading and Feedback for Open Response Mathematical Questions. In *Proceedings of the Second (2015) ACM Conference on Learning@Scale* (pp. 167-176).

Lavbič, D., Matek, T., & Zrnec, A. (2016). Recommender System for Learning SQL Using Hints. *Interactive Learning Environments*, 1-17.

Lazar, T., & Bratko, I. (2014). Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In *Proceedings of the 12th International Conference on Intelligent Tutoring Systems* (pp. 306-311).

Le, N. T. (2016). A Classification of Adaptive Feedback in Educational Systems for Programming. *Systems*, 4(2), 22.

Li, G., Wu, W., Sun, Y., Wang, J., & Lai, T. (2007). Transformation-Based Assessment for C Programs. In *Proceedings of the 9th International Symposium on Signal Processing and Its Applications*.

Lin, C. J., Chou, C. Y., & Chan, T. W. (2008). Developing a Computer-Supported Tutoring Interaction Component with Interaction Data Reuse. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems* (pp. 152-161).

Liu, Z., Mostafavi, B., & Barnes, T. (2016). Combining Worked Examples and Problem Solving in a Data-Driven Logic Tutor. In *Proceedings of the 13th International Conference on Intelligent Tutoring Systems* (pp. 347-353).

Looi, C. K. (1991). Automatic Debugging of Prolog Programs in a Prolog Intelligent Tutoring System. *Instructional Science*, 20(2), 215-263.

Lovett, M., Meyer, O., & Thille, C. (2008). The Open Learning Initiative: Measuring the Effectiveness of the OLI Statistics Course in Accelerating Student Learning. *Journal of Interactive Media in Education*. http://jime.open.ac.uk/2008/14

Marin, V. J., Pereira, T., Sridharan, S., & Rivero, C. R. (2017). Automated Personalized Feedback in Introductory Java Programming MOOCs. In *Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE)* (pp. 1259-1270).

Mathews, M., & Mitrović, T. (2008). How Does Students' Help-Seeking Behaviour Affect Learning?. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems* (pp. 363-372).

McLaren, B. M., Koedinger, K R., Schneider, M., Harrer, A., and Bollen, L. (2004). Bootstrapping Novice Data: Semi-Automated Tutor Authoring Using Student Log Files. *Human-Computer Interaction Institute*. Paper 155. http://repository.cmu.edu/hcii/155

Min, W., Mott, B., & Lester, J. (2014). Adaptive Scaffolding in an Intelligent Game-Based Learning Environment for Computer Science. In *Proceedings of the Second Workshop on AI-supported Education for Computer Science (AIEDCS 2014)* (pp. 41-50).

Mitrovic, A., & Ohlsson, S. (1999). Evaluation of a Constraint-Based Tutor for a Database. *International Journal of Artificial Intelligence in Education*, 10, 238-256.

Mudgal, A. (2016). Syntactic Hint Generation for Introductory Programming Problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 727).

Nguyen, A., Piech, C., Huang, J., & Guibas, L. (2014). Codewebs: Scalable Homework Search for Massive Open Online Programming Courses. In *Proceedings of the 23rd International Conference on World Wide Web* (pp. 491-502).

O'Shea, T. (1979). A Self-Improving Quadratic Tutor. *International Journal of Man-Machine Studies*, 11(1), 97-124.

Paaßen, B., Jensen, J., & Hammer, B. (2016). Execution Traces as a Powerful Data Representation for Intelligent Tutoring Systems for Programming. In *Proceedings of the 9th International Conference on Educational Data Mining* (pp. 183-190).

Papancea, A., Spacco, J., & Hovemeyer, D. (2013). An Open Platform for Managing Short Programming Exercises. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (pp. 47-52).

Pappano, L. (2012). The Year of the MOOC. *The New York Times*, 2(12), 2012.

Paquette, L., Lebeau, J. F., Beaulieu, G., & Mayers, A. (2012). Automating Next-Step Hints Generation Using ASTUS. In *Proceedings of the 11th International Conference on Intelligent Tutoring Systems* (pp. 201-211).

Peddycord III, B., Hicks, A., & Barnes, T. (2014). Generating Hints for Programming Problems Using Intermediate Output. In *Proceedings of the 7th International Conference on Educational Data Mining* (pp. 92-98).

Perelman, D., Gulwani, S., & Grossman, D. (2014). Test-Driven Synthesis for Automated Feedback for Introductory Computer Science Assignments. In *Proceedings of Data Mining for Educational Assessment and Feedback (ASSESS 2014)*.

Petersen, A., Craig, M., Campbell, J., & Tafliovich, A. (2016). Revisiting Why Students Drop CS1. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 71-80).

Phothilimthana, P. & Sridhara, S. (2017). High-Coverage Hint Generation for Massive Courses: Do Automated Hints Help CS1 Students? In *Proceedings of the 22nd Annual Conference on Innovation and Technology in Computer Science Education*.

Piech, C., Huang, J., Nguyen, A., Phulsuksombati, M., Sahami, M., & Guibas, L. (2015b). Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32nd International Conference on Machine Learning* (pp. 1093-1102).

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012). Modeling how Students Learn to Program. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 153-160).

Piech, C., Sahami, M., Huang, J., & Guibas, L. (2015a). Autonomously Generating Hints by Inferring Problem Solving Policies. In *Proceedings of the Second (2015) ACM Conference on Learning@Scale* (pp. 195-204).

Postner, L., & Stevens, R. (2005). What Resources Do CS1 Students Use and How Do They Use Them? *Computer Science Education*, 15(3), 165-182.

Price, T. W., & Barnes, T. (2015). An Exploration of Data-Driven Hint Generation in an Open-Ended Programming Problem. In *Proceedings of the 2nd International Workshop on Graph-Based Educational Data Mining*.

Price, T. W., Dong, Y., & Barnes, T. (2016). Generating Data-Driven Hints for Open-Ended Programming. In *Proceedings of the 9th International Conference on Educational Data Mining* (pp. 191-198).

Price, T. W., Dong, Y., & Lipovac, D. (2017). iSnap: Towards Intelligent Tutoring in Novice Programming Environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 483-488).

Razzaq, L., & Heffernan, N. T. (2010). Hints: Is It Better to Give or Wait to be Asked?. In *Proceedings of the 10th International Conference on Intelligent Tutoring Systems* (pp. 349-358).

Razzaq, L., Heffernan, N. T., & Lindeman, R. W. (2007). What Level of Tutor Interaction is Best? In *Proceedings of the 2007 Conference on Artificial Intelligence in Education: Building Technology Rich Learning Contexts That Work* (pp. 222-229).

Rivers, K., & Koedinger, K. R. (2012). A Canonicalizing Model for Building Programming Tutors. In *Proceedings of the 11th International Conference on Intelligent Tutoring Systems* (pp. 591-593).

Rivers, K., & Koedinger, K. R. (2013). Automatic Generation of Programming Feedback: A Data-Driven Approach. In *Proceedings of the First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)* (pp. 50-59).

Rivers, K., & Koedinger, K. R. (2014). Automating Hint Generation with Solution Space Path Construction. In *Proceedings of the 12th International Conference on Intelligent Tutoring Systems* (pp. 329-339).

Rivers, K., & Koedinger, K. R. (2017). Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 27(1), 37-64.

Rolim, R., Soares, G., D'Antoni, L., Polozov, O., Gulwani, S., Gheyi, R., Suzuki, R., & Hartmann, B. (2017). Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (pp. 404-415).

Roll, I., Baker, R. S. D., Aleven, V., & Koedinger, K. R. (2014). On the Benefits of Seeking (and Avoiding) Help in Online Problem-Solving Environments. *Journal of the Learning Sciences*, 23(4), 537-560.

Shih, B., Koedinger, K. R., & Scheines, R. (2011). A Response Time Model for Bottom-out Hints as Worked Examples. *Handbook of Educational Data Mining*, 201-212.

Shute, V. J. (2008). Focus on Formative Feedback. *Review of Educational Research*, 78(1), 153-189.

Singh, R., Gulwani, S., & Solar-Lezama, A. (2013). Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 15-26).

So, S. & Oh, H. (2017). Synthesizing Imperative Programs for Introductory Programming Assignments. In *Proceedings of the 24th Static Analysis Symposium*.

Soh, L. K., & Blank, T. (2008). Integrating Case-Based Reasoning and Meta-Learning for a Self-Improving Intelligent Tutoring System. *International Journal of Artificial Intelligence in Education*, 18(1), 27-58.

Srikant, S., & Aggarwal, V. (2013). Automatic Grading of Computer Programs: A Machine Learning Approach. In *Proceedings of the 12th International Conference on Machine Learning and Applications (ICMLA)* (pp. 85-92).

Suarez, M., & Sison, R. (2008). Automatic Construction of a Bug Library for Object-Oriented Novice Java Programmer Errors. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems* (pp. 184-193).

Sudol, L. A., Rivers, K., & Harris, T. K. (2012). Calculating Probabilistic Distance to Solution in a Complex Problem Solving Domain. In *Proceedings of the 5th International Conference on Educational Data Mining* (pp. 144-147).

Sudol-DeLyser, L. A. (2014). *AbstractTutor: Increasing Algorithm Implementation Expertise for Novices Through Algorithmic Feedback* (Doctoral dissertation, Carnegie Mellon University).

Suzuki, R., Soares, G., Glassman, E., Head, A., D'Antoni, L., & Hartmann, B. (2017). Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems* (pp. 2951-2958).

Sykes, E. R., & Franek, F. (2004). Presenting JECA: A Java Error Correcting Algorithm for the Java Intelligent Tutoring System. In *Proceedings of the IASTED International Conference on Advances in Computer Science and Technology*.

VanLehn, K. (2006). The Behavior of Tutoring Systems. *International Journal of Artificial Intelligence in Education*, 16(3), 227-265.

Wang, K., Lin, B., Rettig, B., Pardi, P., & Singh, R. (2017). Data-Driven Feedback Generator for Online Programing Courses. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale* (pp. 257-260).

Wang, T., Su, X., Wang, Y., & Ma, P. (2007). Semantic Similarity-Based Grading of Student Programs. *Information and Software Technology*, 49(2), 99-107.

Watson, C., & Li, F. W. (2014). Failure Rates in Introductory Programming Revisited. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (pp. 39-44).

Wegman, M. N., & Zadeck, F. K. (1991). Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2), 181-210.

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM*, 49(3), 33-35.

Wolf, J. R. and Jia, R. (2015) The Role of Grit in Predicting Student Performance in Introductory Programming Courses: An Exploratory Study. In *Proceedings of the Southern Association for Information Systems Conference* (21).

Xu, S., & San Chee, Y. (2003). Transformation-Based Diagnosis of Student Programs for Programming Tutoring Systems. IEEE Transactions on Software Engineering, 29(4), 360-384.

Zimmerman, K., & Rupakheti, C. R. (2015). An Automated Framework for Recommending Program Elements to Novices. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 283-288)..

Appendix 1: Practice Problems

Due to the length of this appendix, I have hosted it on my website instead of including it within the main thesis. You can find it at http://krivers.net/files/thesis_appendix.pdf

Appendix 2: Technical Evaluation Dataset Statistics

In this appendix, I report summary statistics for the 41 problems used in various technical evaluations. This data was gathered across all the studies reported in this thesis.

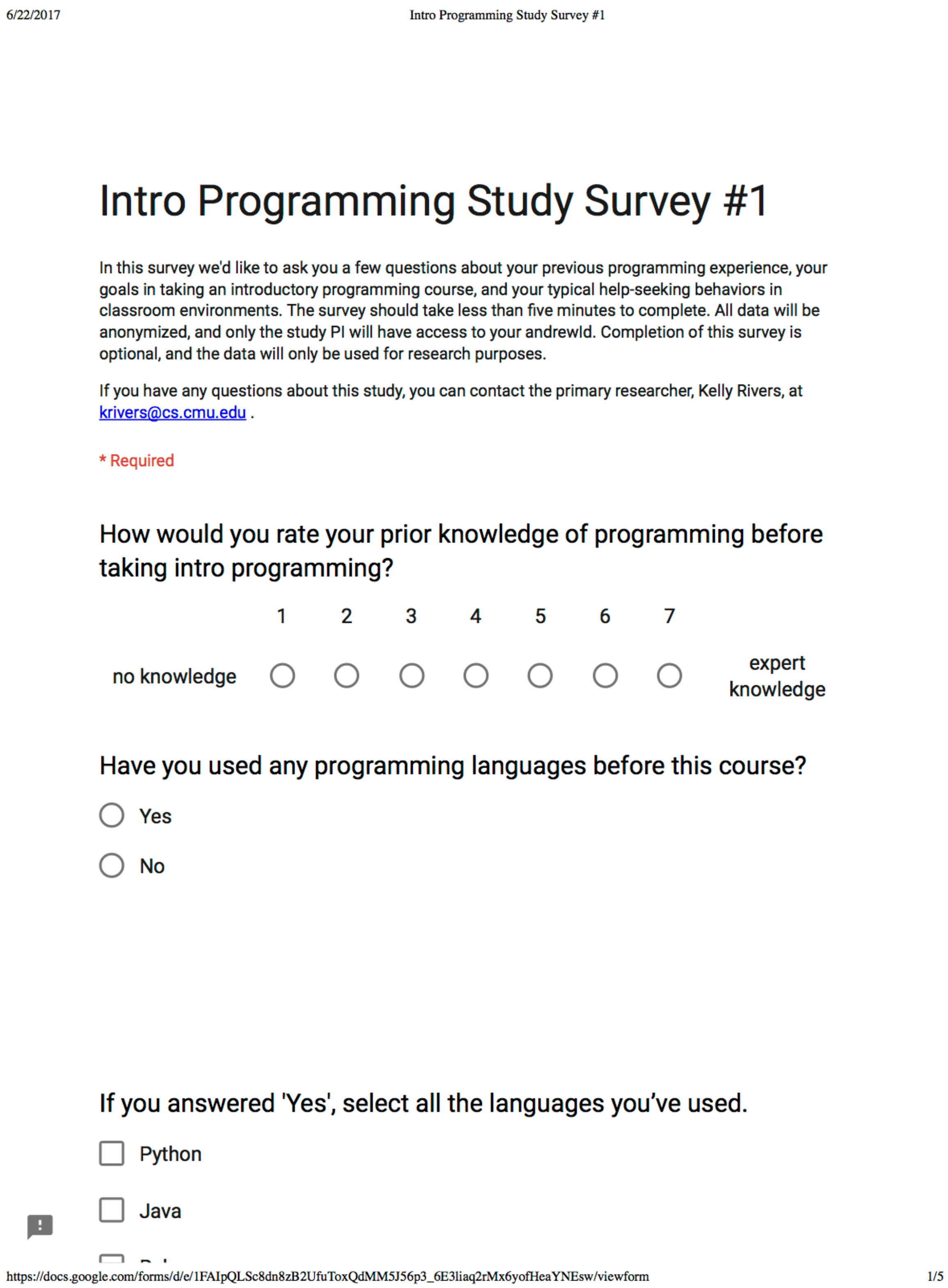
Problem Name	Token Complexity	# Studies	# Students	# Syntax Error States	# Semantic Error States	# Correct States
all_three_chars	18	1	124	0	211	107
any_divisible	22	1	160	46	139	132
any_first_chars	20	1	134	69	140	105
any_lowercase	19	2	165	16	371	115
can_drink_alcohol	10	4	232	66	140	230
can_make_breakfast	10	1	128	0	35	125
convert_to_degrees	8	2	88	17	62	90
count_all_empty_strings	30	1	147	0	264	103
create_number_block	28	1	18	45	97	12
factorial	18	3	27	15	67	31
find_root	41	2	73	22	32	76
find_the_circle	36	3	138	79	279	93
first_and_last	9	1	25	19	65	26
get_extra_bagel	13	1	158	26	55	149
go_to_gym	11	1	149	20	72	139
has_balanced_parentheses	37	2	151	134	281	91
has_extra_fee	10	1	163	0	184	155
has_two_digits	8	5	277	134	333	281
hello_world	3	2	106	83	141	103
how_many_egg_cartons	10	2	71	109	232	68
is_even_positive_int	17	2	55	62	172	52
is_leap_month	13	1	167	95	112	160
is_prime	24	3	38	16	111	46
is_punctuation	9	1	31	51	112	19
is_substring	17	2	37	55	117	54
kth_digit	14	2	67	37	215	34
last_index	55	1	109	69	178	70

list_of_lists	34	2	115	0	179	86
multiply_numbers	17	1	167	30	153	158
nearest_bus_stop	10	2	61	50	141	77
no_positive_even	24	1	125	0	93	111
one_to_n	19	3	186	137	560	137
over_nine_thousand	6	3	83	28	43	83
reduce_to_positive	24	2	148	61	308	99
second_largest	56	1	124	52	203	86
single_pig_latin	10	2	48	61	110	46
sum_all_even_numbers	33	1	137	0	118	109
sum_of_digits	19	2	23	15	76	24
sum_of_odd_digits	28	1	154	93	291	119
was_lincoln_alive	8	1	161	14	72	154
wear_a_coat	11	1	169	33	116	168

Appendix 3: Surveys

In this appendix, I include the surveys used in this thesis. The first three surveys were all from Study 1, and the fourth is from the Usability Study.

Study 1 Survey 1



6/22/2017 Intro Programming Study Survey #1

Intro Programming Study Survey #1

In this survey we'd like to ask you a few questions about your previous programming experience, your goals in taking an introductory programming course, and your typical help-seeking behaviors in classroom environments. The survey should take less than five minutes to complete. All data will be anonymized, and only the study PI will have access to your andrewId. Completion of this survey is optional, and the data will only be used for research purposes.

If you have any questions about this study, you can contact the primary researcher, Kelly Rivers, at krivers@cs.cmu.edu .

* Required

How would you rate your prior knowledge of programming before taking intro programming?

	1	2	3	4	5	6	7	
no knowledge	○	○	○	○	○	○	○	expert knowledge

Have you used any programming languages before this course?

○ Yes

○ No

If you answered 'Yes', select all the languages you've used.

□ Python

□ Java

https://docs.google.com/forms/d/e/1FAIpQLSc8dn8zB2UfuToxQdMM5J56p3_6E3liaq2rMx6yofHeaYNEsw/viewform 1/5

□ Ruby

□ Javascript

□ C

□ C++

□ Scratch

□ Visual Basic

□ Matlab

□ R

□ Logo

□ Alice

□ Blockly

□ Karel

□ Haskell

□ Lisp

□ PHP

□ Perl

□ Other:

Approximately what is your Math SAT score? (200-800)

Your answer

For the next nine questions, please indicate the degree to which you think each statement is true of yourself:

I try to finish whatever I begin.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

It is important for me to do better than other students.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I worry that I may not learn all that I possibly could in this class.

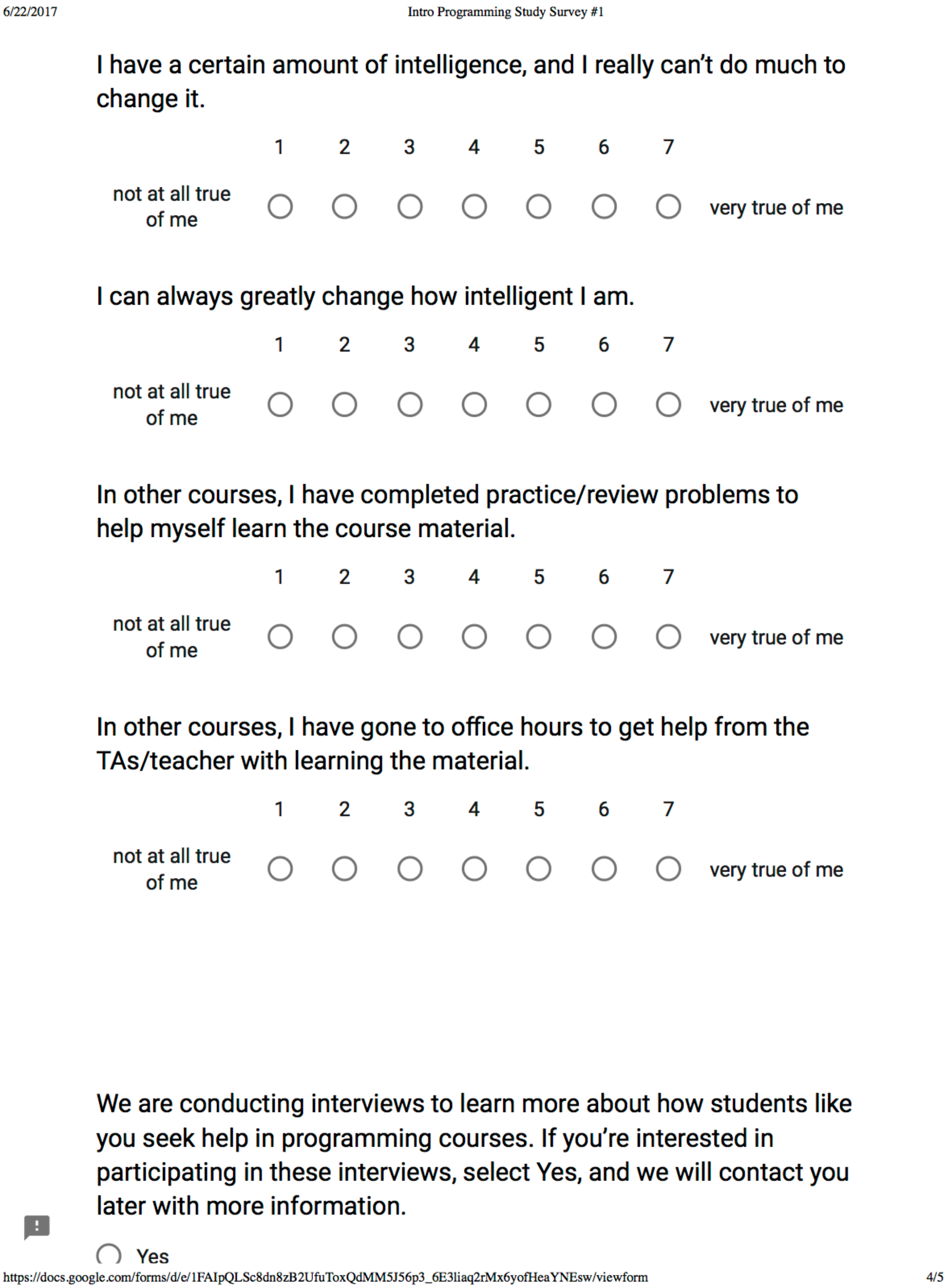
	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I just want to avoid doing poorly in this class.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I want to learn as much as possible from this class.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me



I have a certain amount of intelligence, and I really can’t do much to change it.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I can always greatly change how intelligent I am.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

In other courses, I have completed practice/review problems to help myself learn the course material.

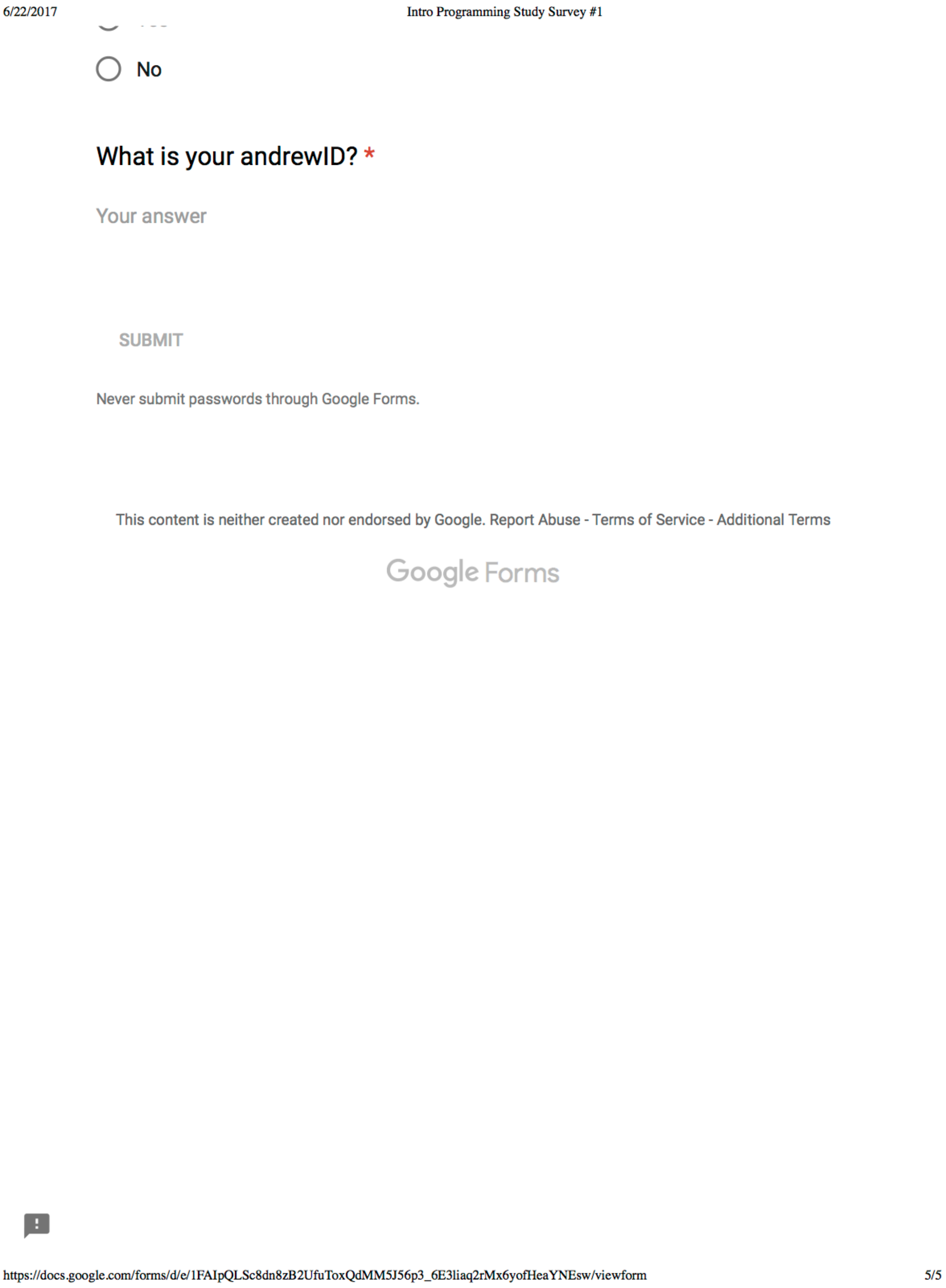
	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

In other courses, I have gone to office hours to get help from the TAs/teacher with learning the material.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

We are conducting interviews to learn more about how students like you seek help in programming courses. If you’re interested in participating in these interviews, select Yes, and we will contact you later with more information.

○ Yes



6/22/2017 Intro Programming Study Survey #1

○ No

What is your andrewID? *

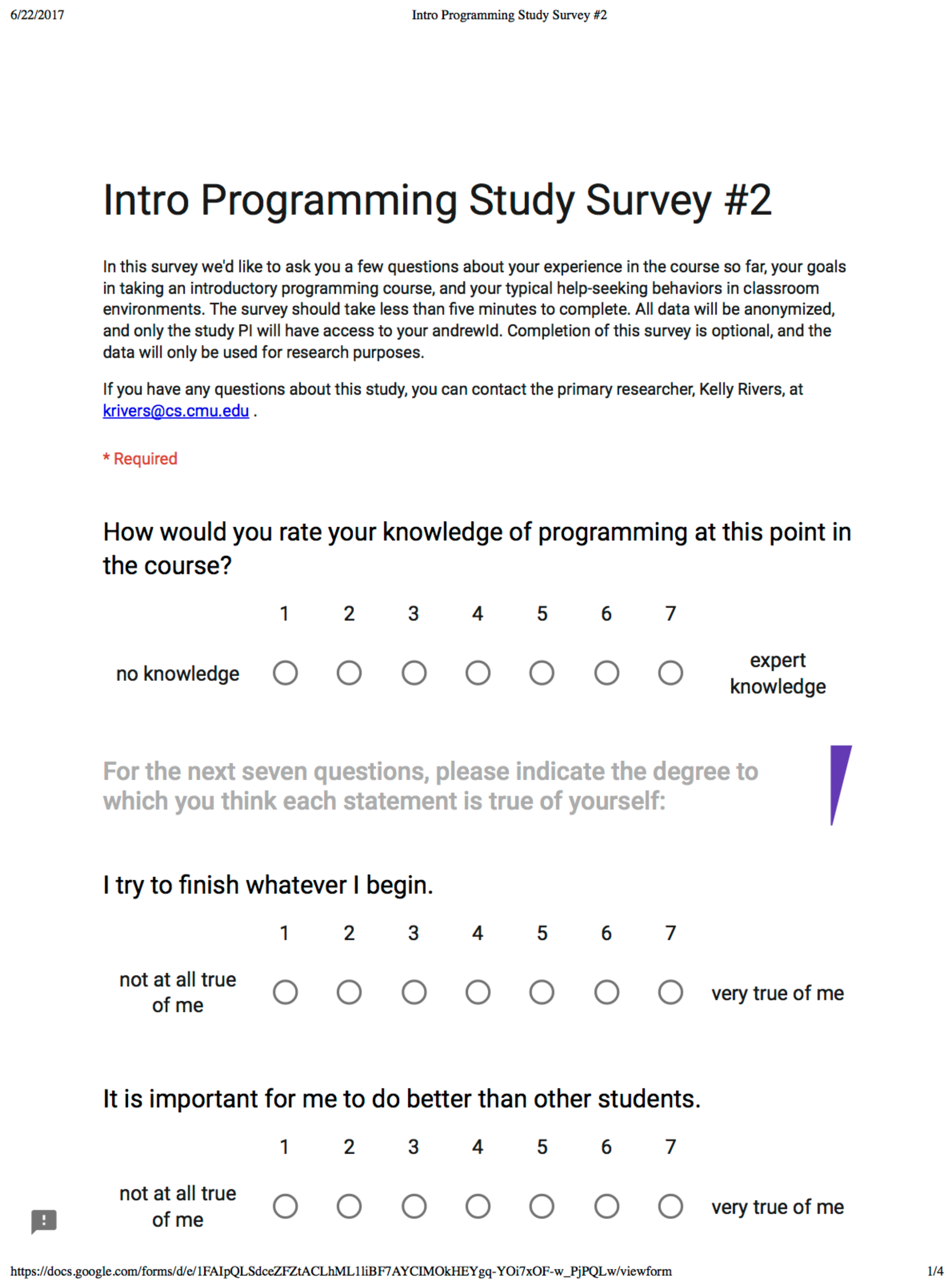
Your answer

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Additional Terms

https://docs.google.com/forms/d/e/1FAIpQLSc8dn8zB2UfuToxQdMM5J56p3_6E3liaq2rMx6yofHeaYNEsw/viewform 5/5

Study 1 Survey 2



6/22/2017 Intro Programming Study Survey #2

Intro Programming Study Survey #2

In this survey we'd like to ask you a few questions about your experience in the course so far, your goals in taking an introductory programming course, and your typical help-seeking behaviors in classroom environments. The survey should take less than five minutes to complete. All data will be anonymized, and only the study PI will have access to your andrewId. Completion of this survey is optional, and the data will only be used for research purposes.

If you have any questions about this study, you can contact the primary researcher, Kelly Rivers, at krivers@cs.cmu.edu .

* Required

How would you rate your knowledge of programming at this point in the course?

	1	2	3	4	5	6	7	
no knowledge	○	○	○	○	○	○	○	expert knowledge

For the next seven questions, please indicate the degree to which you think each statement is true of yourself:

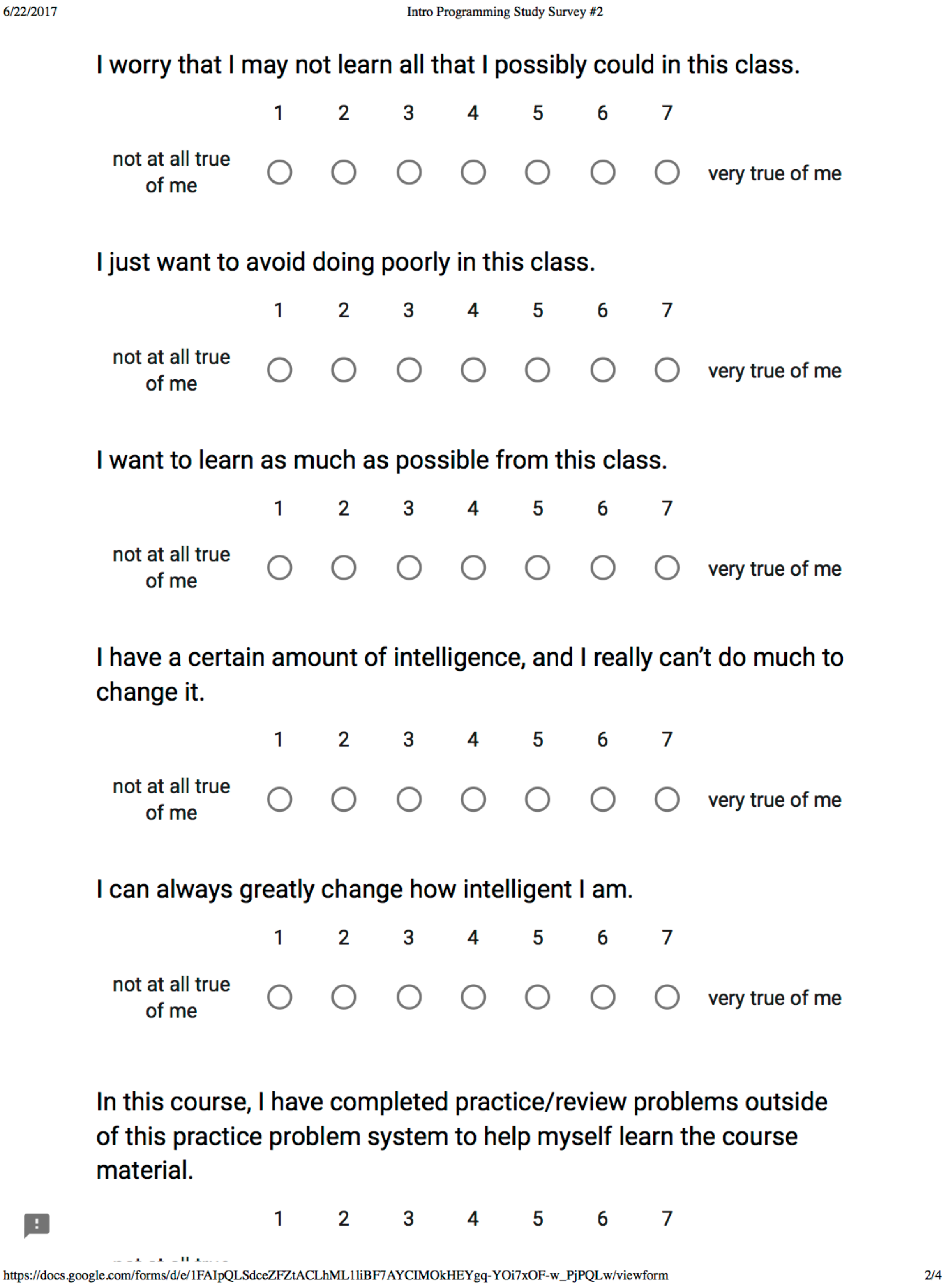
I try to finish whatever I begin.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

It is important for me to do better than other students.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

https://docs.google.com/forms/d/e/1FAIpQLSdceZFZtACLhML1liBF7AYCIMOkHEYgq-YOi7xOF-w_PjPQLw/viewform 1/4



I worry that I may not learn all that I possibly could in this class.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I just want to avoid doing poorly in this class.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I want to learn as much as possible from this class.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I have a certain amount of intelligence, and I really can't do much to change it.

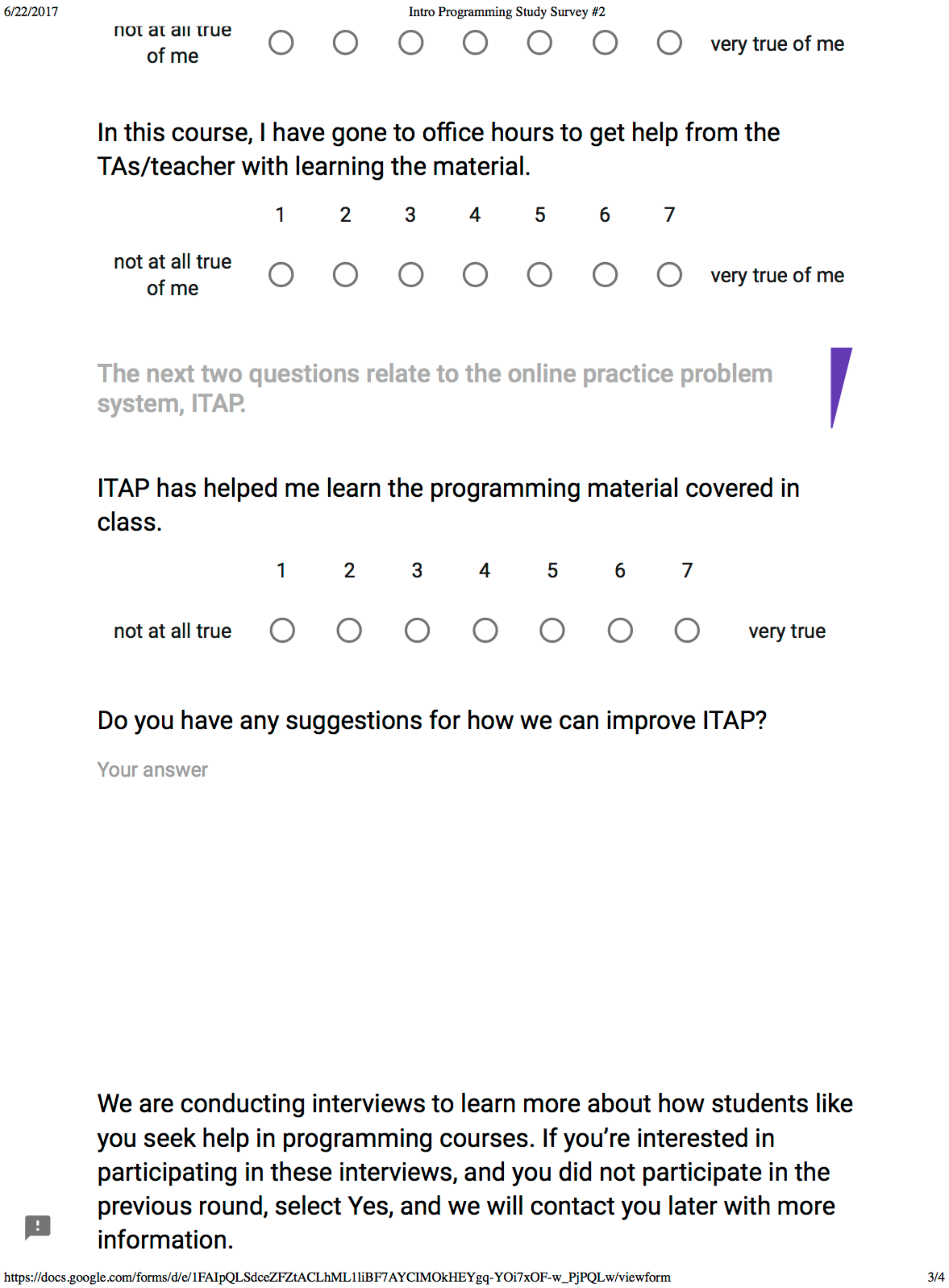
	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I can always greatly change how intelligent I am.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

In this course, I have completed practice/review problems outside of this practice problem system to help myself learn the course material.

	1	2	3	4	5	6	7	



not at all true of me	○	○	○	○	○	○	○	very true of me

In this course, I have gone to office hours to get help from the TAs/teacher with learning the material.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

The next two questions relate to the online practice problem system, ITAP.

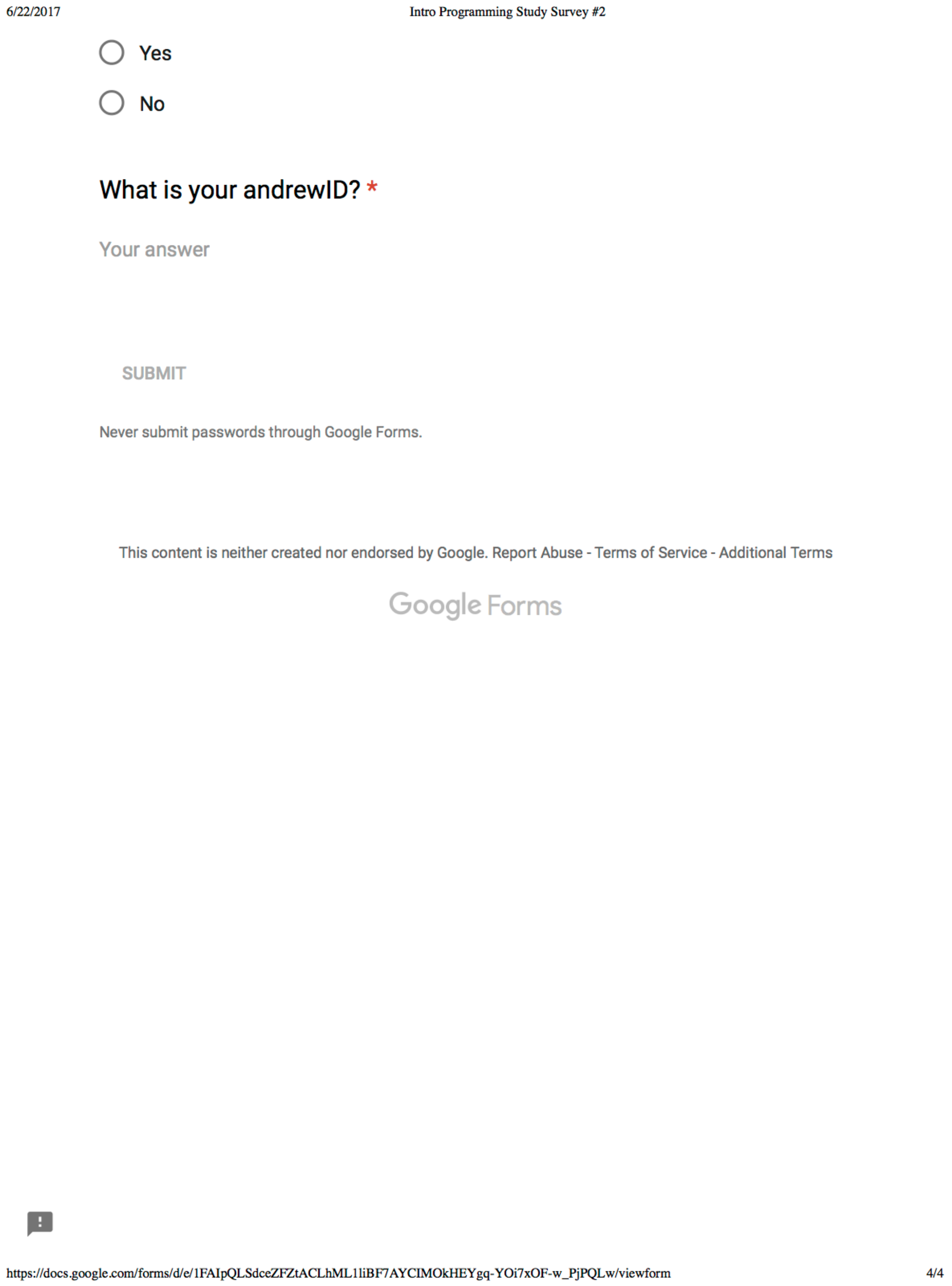
ITAP has helped me learn the programming material covered in class.

	1	2	3	4	5	6	7	
not at all true	○	○	○	○	○	○	○	very true

Do you have any suggestions for how we can improve ITAP?

Your answer

We are conducting interviews to learn more about how students like you seek help in programming courses. If you're interested in participating in these interviews, and you did not participate in the previous round, select Yes, and we will contact you later with more information.



6/22/2017 Intro Programming Study Survey #2

○ Yes

○ No

What is your andrewID? *

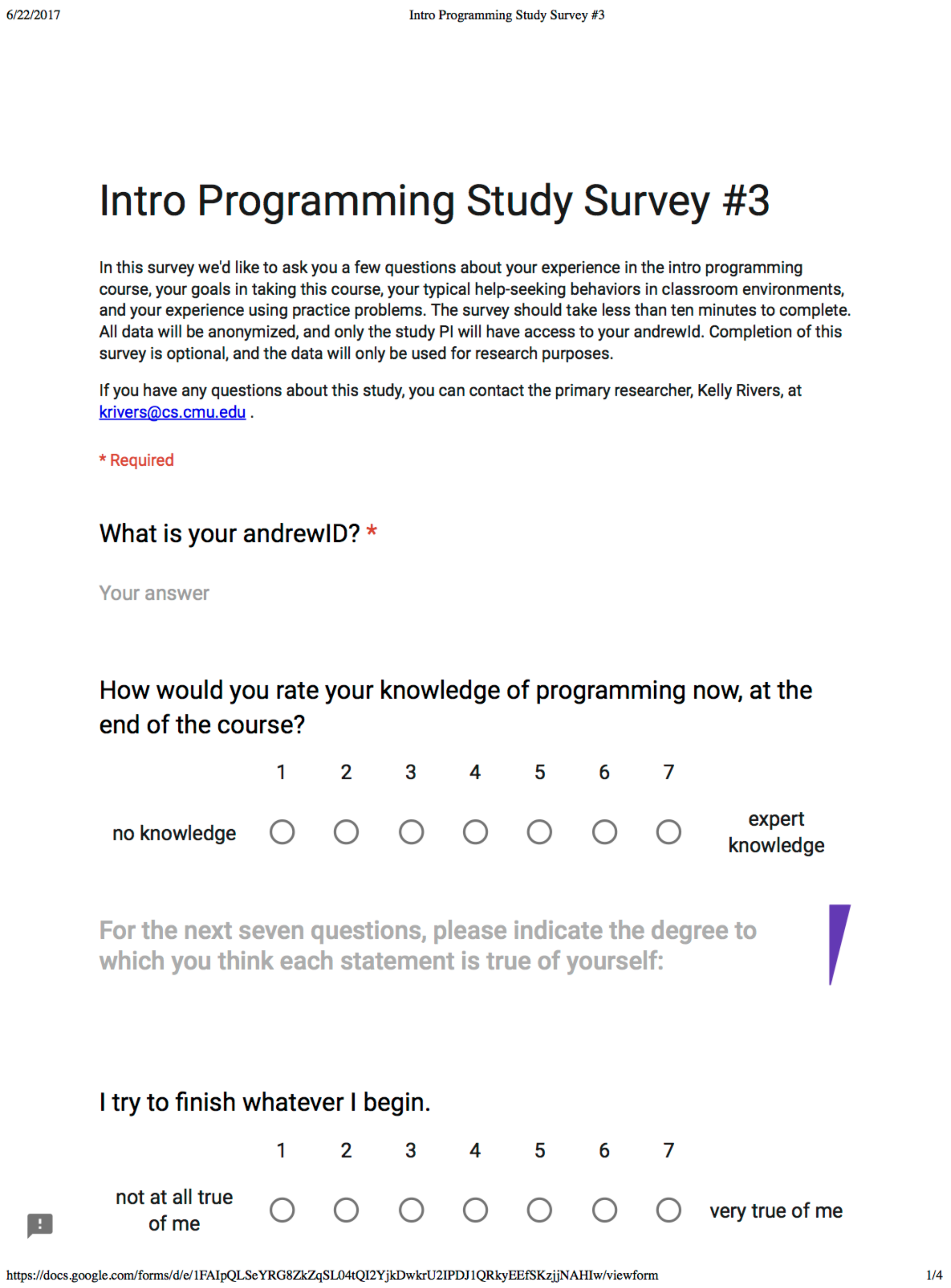
Your answer

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Additional Terms

https://docs.google.com/forms/d/e/1FAIpQLSdceZFZtACLhML1liBF7AYCIMOkHEYgq-YOi7xOF-w_PjPQLw/viewform 4/4

Study 1 Survey 3



6/22/2017 Intro Programming Study Survey #3

Intro Programming Study Survey #3

In this survey we'd like to ask you a few questions about your experience in the intro programming course, your goals in taking this course, your typical help-seeking behaviors in classroom environments, and your experience using practice problems. The survey should take less than ten minutes to complete. All data will be anonymized, and only the study PI will have access to your andrewId. Completion of this survey is optional, and the data will only be used for research purposes.

If you have any questions about this study, you can contact the primary researcher, Kelly Rivers, at krivers@cs.cmu.edu .

* Required

What is your andrewID? *

Your answer

How would you rate your knowledge of programming now, at the end of the course?

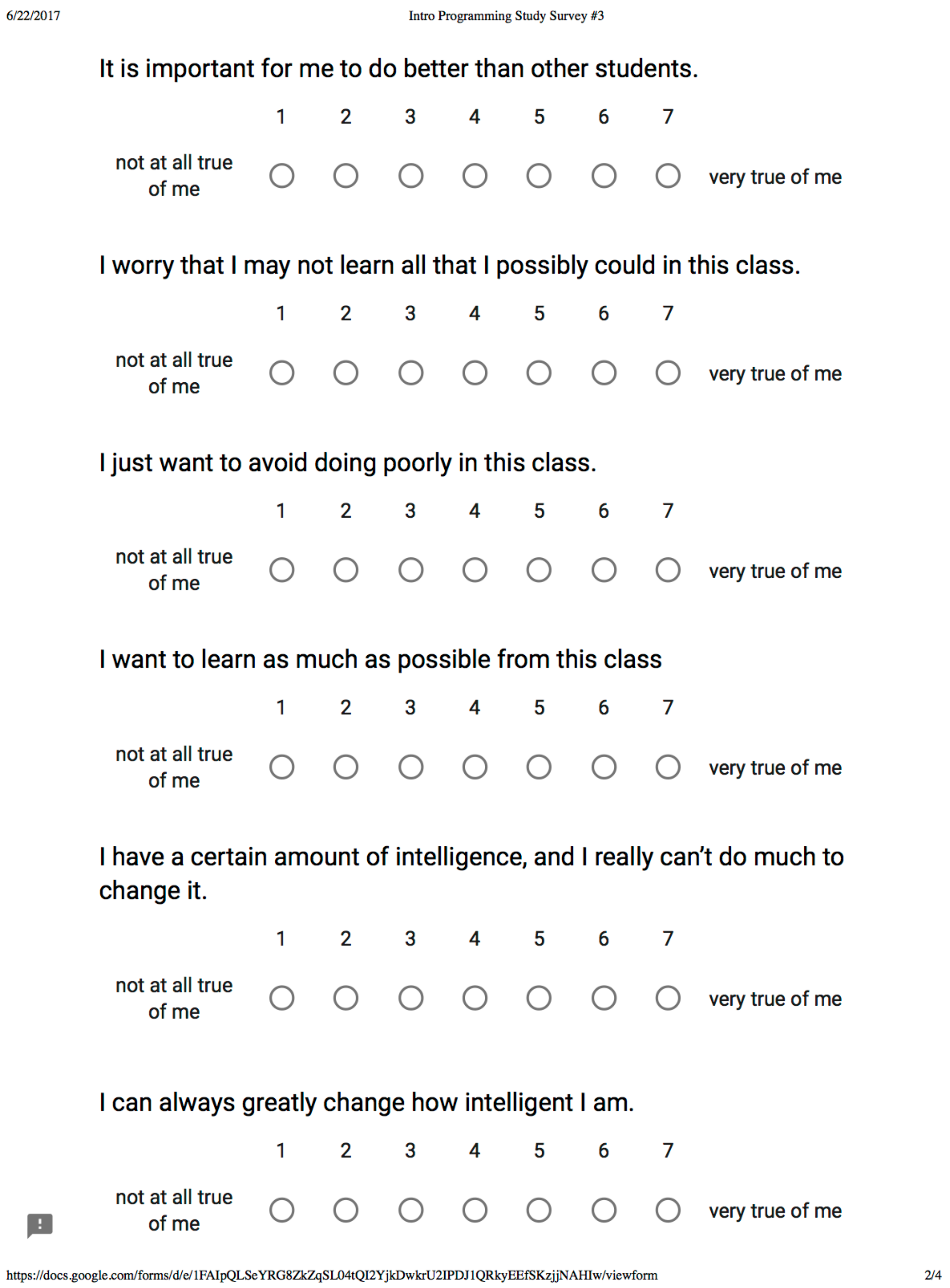
	1	2	3	4	5	6	7	
no knowledge	○	○	○	○	○	○	○	expert knowledge

For the next seven questions, please indicate the degree to which you think each statement is true of yourself:

I try to finish whatever I begin.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

https://docs.google.com/forms/d/e/1FAIpQLSeYRG8ZkZqSL04tQI2YjkDwkrU2IPDJ1QRkyEEfSKzjjNAHIw/viewform 1/4



6/22/2017 Intro Programming Study Survey #3

It is important for me to do better than other students.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I worry that I may not learn all that I possibly could in this class.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I just want to avoid doing poorly in this class.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I want to learn as much as possible from this class

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I have a certain amount of intelligence, and I really can't do much to change it.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

I can always greatly change how intelligent I am.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

https://docs.google.com/forms/d/e/1FAIpQLSeYRG8ZkZqSL04tQI2YjkDwkrU2IPDJ1QRkyEEfSKzjjNAHIw/viewform 2/4

In this course, I have completed practice/review problems outside of ITAP to help myself learn the course material.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

In this course, I have gone to office hours to get help from the TAs/teacher with learning the material.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

How well do you think you met your course goal(s)?

	1	2	3	4	5	6	7	
not at all	○	○	○	○	○	○	○	very well

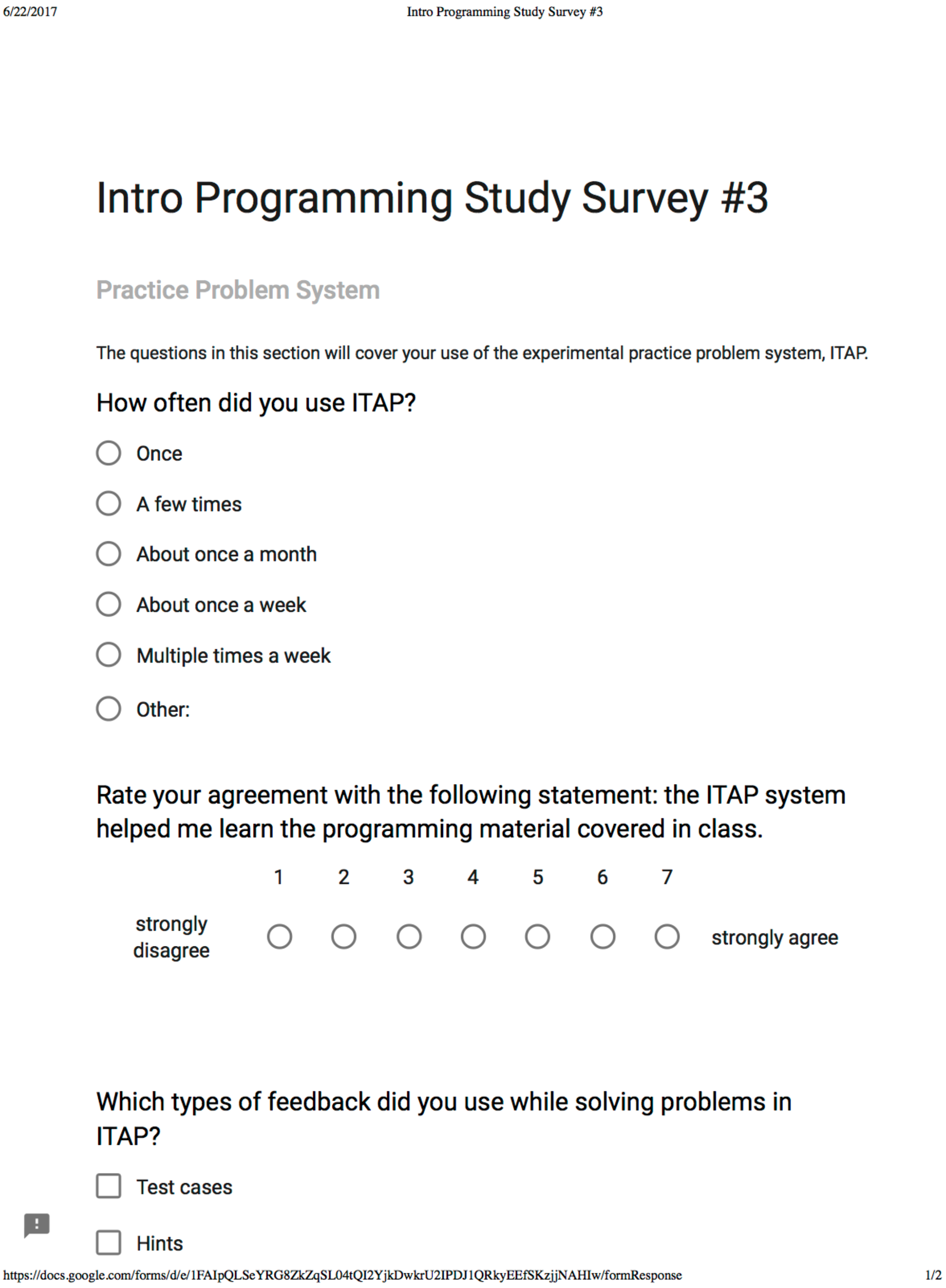
Did you ever use the practice problem system (ITAP/CloudCoder) this semester?

○ Yes

○ No

Briefly describe why you did or did not choose to use the practice problem system.

Your answer



Intro Programming Study Survey #3

Practice Problem System

The questions in this section will cover your use of the experimental practice problem system, ITAP.

How often did you use ITAP?

○ Once

○ A few times

○ About once a month

○ About once a week

○ Multiple times a week

○ Other:

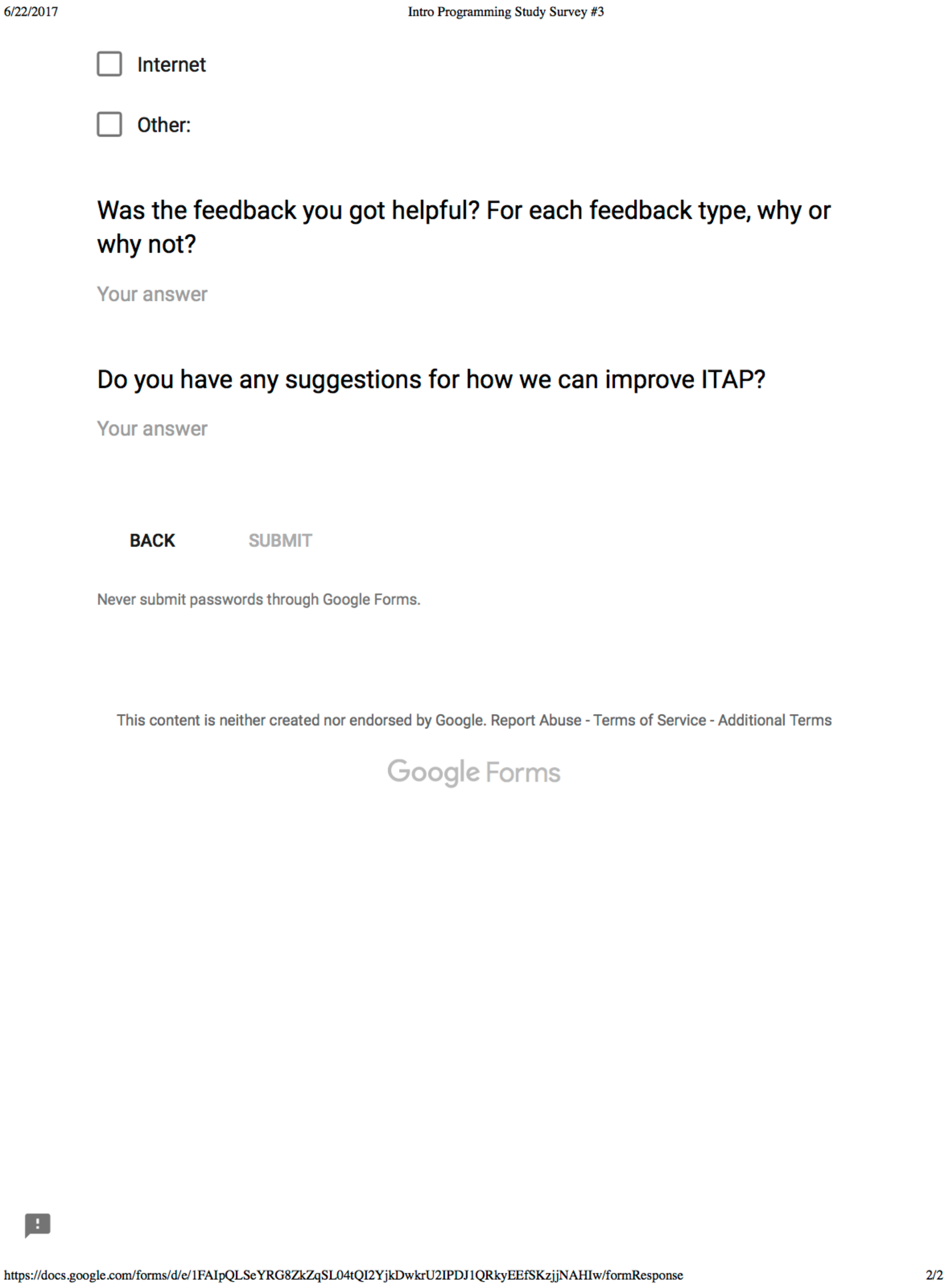
Rate your agreement with the following statement: the ITAP system helped me learn the programming material covered in class.

	1	2	3	4	5	6	7	
strongly disagree	○	○	○	○	○	○	○	strongly agree

Which types of feedback did you use while solving problems in ITAP?

□ Test cases

□ Hints



6/22/2017 Intro Programming Study Survey #3

□ Internet

□ Other:

Was the feedback you got helpful? For each feedback type, why or why not?

Your answer

Do you have any suggestions for how we can improve ITAP?

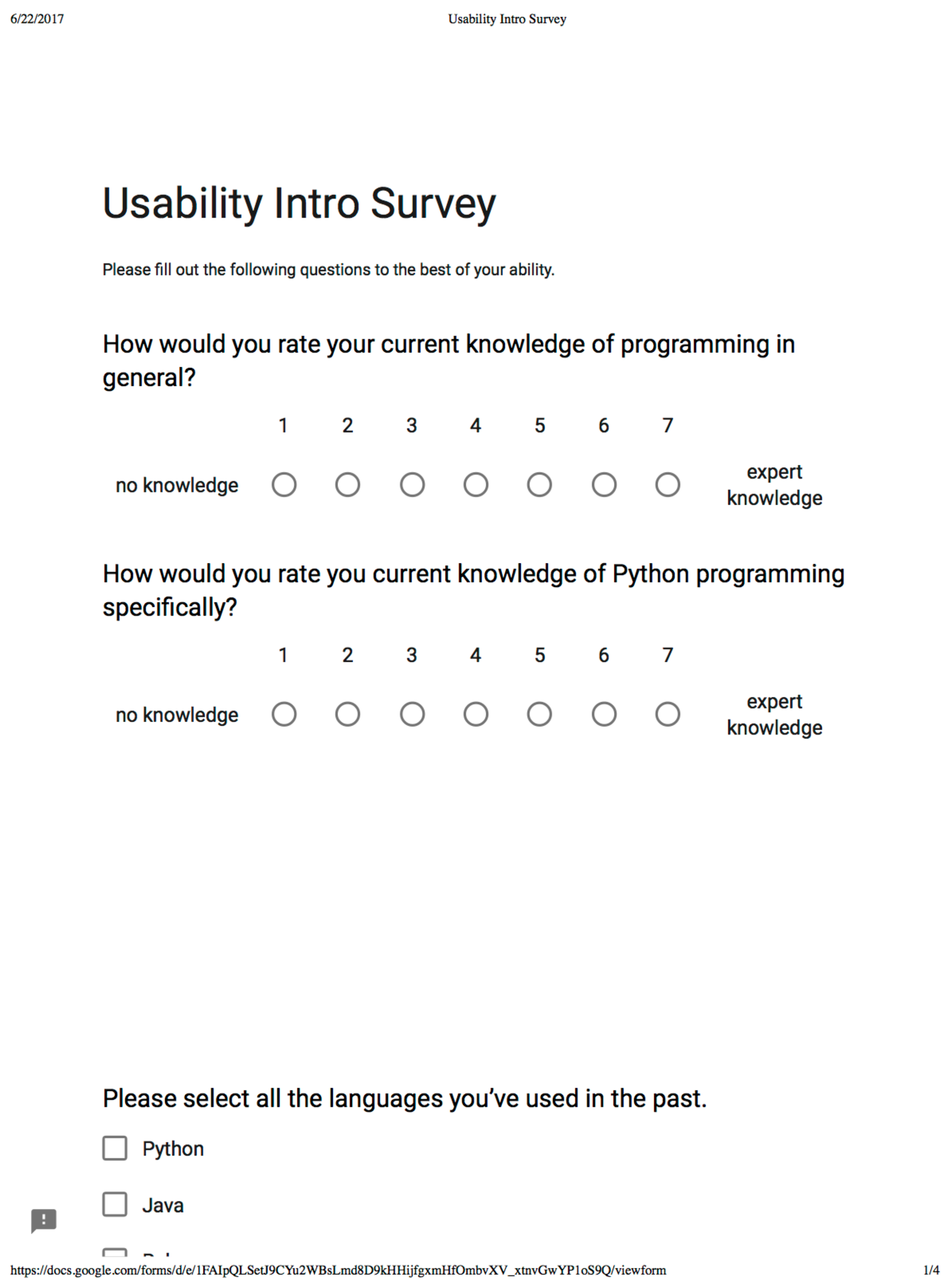
Your answer

Never submit passwords through Google Forms.

This content is neither created nor endorsed by Google. Report Abuse - Terms of Service - Additional Terms

https://docs.google.com/forms/d/e/1FAIpQLSeYRG8ZkZqSL04tQI2YjkDwkrU2IPDJ1QRkyEEfSKzjjNAHIw/formResponse 2/2

Usability Study Survey



6/22/2017 Usability Intro Survey

Usability Intro Survey

Please fill out the following questions to the best of your ability.

How would you rate your current knowledge of programming in general?

	1	2	3	4	5	6	7	
no knowledge	○	○	○	○	○	○	○	expert knowledge

How would you rate you current knowledge of Python programming specifically?

	1	2	3	4	5	6	7	
no knowledge	○	○	○	○	○	○	○	expert knowledge

Please select all the languages you've used in the past.

□ Python

□ Java

https://docs.google.com/forms/d/e/1FAIpQLSetJ9CYu2WBsLmd8D9kHHijfgxmHfOmbvXV_xtnvGwYP1oS9Q/viewform 1/4

□ Ruby

□ Javascript

□ C

□ C++

□ Scratch

□ Visual Basic

□ Matlab

□ R

□ Logo

□ Alice

□ Blockly

□ Karel

□ Haskell

□ Lisp

□ PHP

□ Perl

□ Processing

□ Swift

□ Other:

Please indicate the degree to which you think the following four statements are true of yourself:

I try to finish whatever I begin

I try to finish whatever I begin.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

When learning in the past, I have completed practice/review problems to help myself learn the course material.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

When learning in the past, I have gone to office hours to get help from the TAs/teacher with learning the material.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

When learning in the past, I have looked for additional information/help online while learning the material.

	1	2	3	4	5	6	7	
not at all true of me	○	○	○	○	○	○	○	very true of me

SUBMIT

Never submit passwords through Google Forms.