# Automated Data-Driven Hint Generation for Learning Programming

**Kelly Rivers**
Ph.D. Thesis Proposal
February 11th, 2016
Human-Computer Interaction Institute, Carnegie Mellon University

**Committee:**
Kenneth R. Koedinger (Chair) (HCII & Psychology, CMU)
Brad Myers (HCII & ISR, CMU)
Vincent Aleven (HCII, CMU)
Sharon Carver (Psychology, CMU)
Tiffany Barnes (CS, North Carolina State University)

**Abstract**
Introductory programming has become a highly desirable subject of study, but it can prove to be very difficult for novices. In particular, novice programmers often get stuck and frustrated while they are attempting to solve problems. One way to provide support for these students is to use intelligent tutoring systems (ITSs), which can help students master material with more support. However, these systems take much time to create, especially for problems with very large potential solution spaces.

My research has focused on whether we can automate the production of ITS components such that they are still well-formed and useful for novices. I have designed and constructed ITAP, the Intelligent Teaching Assistant for Programming, which uses collected student data to automatically construct next-step hints. This system uses algorithms for state abstraction, path construction, and hint reification to automate the generation of hint messages. I have tested ITAP with novice programmers and identified potential research areas for exploration and improvement.

For my thesis work, I plan to conduct a series of experiments that determine the effect hint availability has on student performance and learning. I also plan to study how students' internal motivation and help-seeking beliefs affect their practice habits and learning, and I intend to evaluate ITAP's ability to improve its performance as data is collected. I anticipate that the findings of these planned studies will result in modifications to ITAP that make it a practical, useful, and usable system.

**Table of Contents**

# Introduction

Computational thinking has been recognized as an essential skill for students to learn in order to succeed in the current world (Wing 2006). However, programming, a core component of computational thinking, is notoriously difficult to learn, with high dropout and fail rates in introductory programming courses (Watson & Li 2014). This difficulty is in large part due to students struggling when they encounter difficulties while trying to solve programming problems, especially when they do not have access to immediate teacher assistance. This struggle is often compounded by the need to practice using new concepts in a language that is also new to the student.

One way to reduce student struggle is to provide fast cycles of feedback, which can ease work while retaining learning. A common way to provide this fast feedback is to have students use intelligent tutoring systems, which are designed to provide personalized problem selection, feedback, and hints. These systems have been shown to help students learn in less time with less unnecessary work (Corbett & Anderson 2001). Hints are an essential component of ITSs, as they ensure that a student can learn what to do immediately when they are struggling, turning impossible problems into worked examples. But hints take a long time and great expert knowledge to make, and they are hard to construct for domains with large solution spaces, like programming (Corbett & Koedinger 1997; Folsom-Kovarik et al. 2010).

Recent trends in academia (including MOOCs and other online courses) have led to more data being collected on how students solve problems while learning new concepts. This newly available data leads to new opportunities. Instead of building tutors the traditional way, by having the instructor map out expected work, what if we used the paths past students created while solving problems? This idea has been investigated with data-driven tutoring, an approach developed by Barnes and Stamper (2010) that collects the process data of students who in a logic tutor to determine which steps students usually take after landing on specific states. These steps can then be recommended to students who get stuck on the same state later on. This implementation works well, but there are some drawbacks to only using previously-seen student paths. For example, relying entirely on previous student processes means that the tutor will not be able to produce hints 100% of the time, as it will sometimes encounter never-seen-before states (Barnes & Stamper 2008). Ideally, we would like for data-driven tutors to provide assistance for any state that a student might encounter.

My previous work has focused on developing and testing ITAP, the Intelligent Teaching Assistant for Programming (Rivers & Koedinger 2015). ITAP is a system that does real-time, automatic next-step hint generation for any given program state, given a problem statement, a correct program, and a set of evaluative tests. Hint generation is even possible if the given program state has not been seen before due to canonicalization, the representation model of the programs, and the path construction method of connecting different program states. In this thesis proposal, I will discuss how ITAP works, how we have evaluated it so far, and what we have determined from observing student interactions with the system. I will then outline the next

steps I plan to take with the goal of investigating how students interact with practice problems and feedback and how effective ITAP can be in introductory programming classes.

# Background

As mentioned above, the first iteration of data-driven tutoring with a focus on hint generation was shown in the Hint Factory, a system for logic tutors that uses Markov decision processes to collapse student action paths into a single graph, a solution space (Barnes & Stamper 2008). This approach is dependent on pre-existing student data, but is still able to generate next-step hints for a large number of the states with a relatively small subset of the data. The Hint Factory approach of using pre-existing data has been extended to work in other domains more closely related to programming. One example is a tutor for learning how to work with linked lists, using the data structure's state for solution state representation (Fossati et al. 2009). Another adaptation brought hint generation to educational games with a new form of state representation that used the output of the game to represent the student's current work (Hicks et al. 2014). When this output format is used by the Hint Factory, it is able to generate hints more often and with less data than code-based states (Peddycord III et al. 2014), though the hints are focused on changing the output instead of changing the code.

Our own method of path construction, which will be described further on in the paper, extends the Hint Factory by enhancing the solution space, creating new edges for states that are disconnected instead of relying on student-generated paths (Rivers & Koedinger 2014). This method makes it possible to generate hints for never-seen-before states, which the original Hint Factory could not do. Path construction has been adapted to work within an educational game for programming, using the visual programs that the students create to populate a solution space (Min et al. 2014). Similar work has also been done with the goal of giving students feedback on programming style, by creating chains of correct solutions in order to help students make their code more readable (Moghadam et al. 2015). A recent comparative analysis by Piech et al. (2015) tested multiple solution space generation algorithms (including our path construction and their problem solving policies) to determine how often the selected next states matched the next states chosen by expert teachers. They found that several of the approaches had a high match rate, indicating that this new approach has great potential to generate the hints that students will benefit the most from.

There have been other data-driven tutoring approaches developed in recent years that are not based on the Hint Factory approach. For example, several researchers have used program synthesis as a method for automatic generation of hints. Singh et al. (2013) used error models and program sketches to find a mapping from student solutions to a reference solution. Lazar and Batko (2014) used student-generated text edits to correct programs over time, thus supporting parsable and non-parsable programs. Perelman et al (2014) used all common expressions that occurred in code to create a database of code phrases, which was then used for hint generation, instead of mining the edits themselves. These approaches have great potential for supporting new and obscure solutions, but also have the drawback of only working

on solutions that are already close to correct; they all tend to fail when the code has many different errors.

One other common approach to data-driven tutoring uses clustering to separate code into different solution approaches instead of generating paths. A standard example of such a system uses state representation to identify the closest optimal solution for any given state, then highlights the differences between states as a hint (Gross et al. 2014). Eagle et al. (2012) combined with this approach with the Hint Factory model to enable macro-level hints in a logic proof tutoring environment. They had solution clusters manually annotated and computed paths between clusters so that students could receive algorithm-level hints instead of getting hints on the details they needed to change. These clustering approaches tend to generate either high-level hints or worked examples, unlike the small, token edits generated by ITAP.

# Building ITAP: the Intelligent Teaching Assistant for Programming

ITAP is a system that, given a specified problem and a program state, generates a next-step hint a student may follow to bring their code closer to a correct solution. As only a correct state and a set of test cases are needed to reach a basic working level for a problem, the system can be used to make problems “hintable” very quickly. For states that are syntactically correct (and therefore parsable), ITAP uses three different stages to generate a hint. States with syntactic errors use a simplified version of the algorithm, which will be described in a later section.

The first stage is *canonicalization*, where the code state is transformed into a normalized version (like an equivalence class) in order to remove unneeded syntactic variation. Canonicalization can also be used outside of ITAP as a method for modeling student program states, for the purpose of examining class work at scale. The second stage is *path construction*, where the algorithm identifies the best goal state for the student, finds the needed edits between the two states, then uses those edits to generate a path of intermediate states between the current state and the goal. The final stage is *reification*, where the edits are reified back to the context of the original code state, then translated into textual hints. In this section, I will describe in more detail how these three stages work.

## Canonicalization: A Process for State Abstraction

The construction of a solution space makes it possible for us to find paths from a student’s state to a correct solution, even if we have never seen their state before. However, these hint paths are not always optimal, as students have a tendency to code inefficiently. When examining student solutions it is common to see a variety of approaches that are identical semantically but incredibly different in their syntax. An example of this variety can be seen in Figure 1, which demonstrates a range of syntactically different solutions that all use the same semantic solution.

```
def canDrinkAlcohol(age, isDriving):
    return ((21 <= age) and
            (not isDriving))
```

```
def canDrinkAlcohol(age, isDriving):
    if (age < 21):
        return False
    if isDriving:
        return False
    return True
```

```
def canDrinkAlcohol(age, isDriving):
    if ((age >= 21) and
        (isDriving == False)):
        return True
    else:
        return False
```

```
def canDrinkAlcohol(age, isDriving):
    if ((isDriving == False) and
        (age >= 21)):
        return True
    else:
        return False
```

```
def canDrinkAlcohol(age, isDriving):
    return ((age >= 21) and
            (isDriving == False))
```

```
def canDrinkAlcohol(age, isDriving):
    if (isDriving or (age < 21)):
        return False
    else:
        return True
```

```
def canDrinkAlcohol(age, isDriving):
    if ((age >= 21) and (not isDriving)):
        return True
    return False
```

```
def canDrinkAlcohol(age, isDriving):
    if ((age < 21) or (isDriving == True)):
        return False
    else:
        return True
```

```
def canDrinkAlcohol(age, isDriving):
    return ((not isDriving) and
            (age >= 21))
```

Figure 1: Nine syntactic variations on the same semantic method, taken from real student code.

These different solutions can be used to give students feedback on efficiency and style (Moghadam et al 2015), but we are primarily interested in giving feedback on the correctness of a problem. Therefore, it is helpful to remove any syntactic variability, to reduce the noise in feedback generation. To remove syntactic variation we represent the student's state in a new form that focuses only on the semantics, by applying a set of normalizing functions that map student code into semantic equivalence classes. This process is called *canonicalization*, as it creates canonical examples of different code strategies (Rivers & Koedinger 2012).

During this process, we represent program code with abstract syntax trees (ASTs). 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 2. This tree-based representation makes it possible to apply semantics-preserving program transformations to code. The transformations we use draw from past work in program transformation, and many of them are directly adopted from the standard set of compiler optimizations. Other transformations were developed based on observation of student code, to simplify inefficient patterns often created by novices.

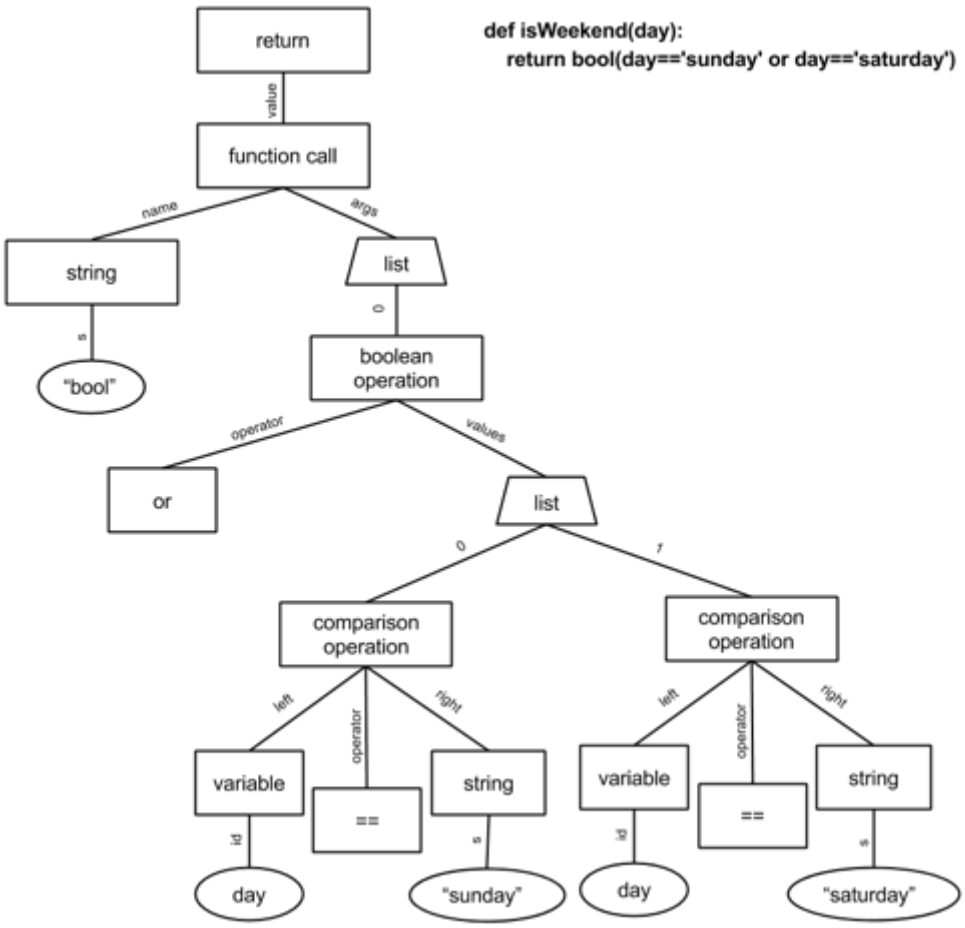


Figure 2: An example of a program code state and the corresponding AST.

We are not the first researchers to use program transformations in the context of computer science education. A few others have also taken this approach, though not for the purpose of intelligent tutoring. Xu and Chee (2003) first used program transformations in order to diagnose errors in student programs, and Gerdes et al. (2010) developed a set of transformations for Haskell that were used to verify program correctness without using test cases. Several of our transformations are similar to the ones developed by these researchers.

First there are a few preprocessing operations applied to the standard Python AST format to make it easier to modify during the rest of the process. These operations mainly involve simplifying multiple-target assignment (a = b = 4 to b = 4; a = b) and augmented assignments (a += 4 to a = a + 4). They also turn multiple-operator comparisons (1 < a < 4) into combinations of single-operator comparisons (1 < a and a < 4).

We also add metadata to the entire AST, giving each node a unique global ID (to enable tracking later on) and, when possible, tagging variables with the types that they contain. Type tagging is done by using teacher-provided data to specify what the types of the method's parameters are, and then extending these types throughout the program as far as possible.

Having the types of variables makes it much easier to determine which parts of the program have the potential to crash, and which expressions are safe.

The final and most important step of preprocessing involves anonymizing all variable names used within the code, to standardize names across student solutions. To anonymize, the algorithm assigns each variable a new name the first time it is encountered. Variables are given a letter (v for normal variables, r for undefined variables (which are usually typos), and g for global values), and the letter is followed by an incremented number. Helper functions are also anonymized, unless the teacher specifies that specific functions should be left alone. In complex problems anonymization provides a large portion of the solution space reduction all on its own, as we expect student variables to differ across solutions.

Outside of these preprocessing transformations, the canonicalizing functions we use fall into three main categories: simplifying, ordering, and domain-specific functions. *Simplification functions* reduce the size of a program by collapsing values in the function and removing code that is not being used. *Ordering functions* are used to reorder values within the code, so that all variations that are semantically equivalent (such as a+b and b+a) can be shown in the same order. This reordering is mainly accomplished through the ordering of commutative operations (using a strict comparison function over all AST node types) and adjustment of certain nested values throughout the code. Finally, there are several *domain-specific transformation functions* that are designed explicitly with novices in mind, functions that aim to reclassify the kind of code only novices will produce. Most of these would have no effect on expert code, but will greatly improve the code of novices who are still learning how to write efficiently. For more information on these program transformations, see our work in (Rivers & Koedinger, 2015).

## Path Construction: A Process for Next-Step Generation

Path construction (Rivers & Koedinger 2014) is the process used to insert new states into the solution space, to determine which actions a student should take to move from their incorrect state to a correct one. This algorithm makes it possible to give hints even when code has never been seen before and identifies new and personalized correct solutions for students. Generated hints take the form of edits between states, where edits are applied sequentially to change the current state into the goal state.

The path construction process is run in parallel with the anonymized code of the student and the canonicalized version of the state, to find the shortest path (as canonicalization occasionally leads to more difficulty in mapping state to goal), with both states represented as ASTs. Since ASTs are stored as tree data structures, we can use tree edit functions to directly compare them. To find a set of differences, we compare the two AST trees of the two program states. If both are AST nodes of the same type (for example, two while loops), we recursively compare each of their child nodes and combine all found changes; if they are two AST nodes of different types, we can instead create an edit at that location. List nodes are compared by finding the optimal matching between the two lists, then adding the remaining changes as adds, removals, moves, and swaps.

We call the edits generated from AST comparison *change vectors*. A change vector has several properties.

- **Path:** A list of nodes and indices that can be traversed through an AST to get to the location where the edit occurs.
- **Old and New Expressions:** These are the old and new values, respectively. For most change vectors, they are code expressions, but for some they are locations; for example, we could be moving a statement from line 1 to line 3, where 1 and 3 are old and new.
- **Type:** Currently, we have several different edit types. The basic type replaces an old value with a new value. Sub vectors describe edits where the old expression is a subset of the new, while Super vectors describe the opposite, edits where the new expression is a subset of the old. Add vectors insert a new value into a location, and Delete vectors remove an old value from a location. Finally, Swap vectors give the locations of two values that can be swapped with each other, and Move vectors identify a location that should be moved to a different place.

With the comparison method and change vectors, we can define a distance metric between ASTs that computes the percentage of nodes that occur in both trees. The distance is measured by the number of edits needed to change one AST into another. At this point, we can define the path construction algorithm.

First, the algorithm determines the optimal goal state for the given state. This step is done by comparing the given state to every correct state within the solution space to determine which state is closest (according to the distance function).

Sometimes the goal state initially chosen perfectly represents the best goal state for the student. However, there are often differences between the current state and the goal state that do not change the results of the test cases. It is inefficient to give students hints for these types of edits, since they are not necessary and might not contribute to learning. Therefore, we seek to personalize the goal state by reducing the set of edits between the given state and the correct state to a minimal set.

To reduce the edit set, we compute the power set of the set of edits, i.e., the set of all possible subsets of edits. Each subset of edits is then applied to the given state, to determine whether any of them lead to a correct solution. If there are some subsets that do lead to correct solutions, we compute their distances from the given state and choose the set that is closest to the given state as the final goal state. Due to the exponential nature of power sets, we institute a cutoff of 15 edits for this process. For states with more than 15 edits, we try to perform incremental optimization (by checking all single-edit and all-but-one edit sets for correctness), but if this fails the state is given a simple path with one edge that leads from the state directly to the goal.

Once the personalized goal state has been identified, the algorithm moves to the second step: finding all possible next states. Here, we again generate the power set of all edits between original state and goal, and apply all subsets of edits to create intermediate states. Not all of these intermediate states are good states; some of them do much worse than the original state, and others might produce awkward code. To remove these unwanted states, we identify three properties that are required to classify an intermediate state as *valid*.

1. A valid state must be well-formed and compilable.
2. A valid state must be closer to the goal than the original solution state.
3. A valid state must do no worse than the original solution state when tested.

The first two properties are straightforward; there is no sense in telling a student to go to a state that is not well-formed, and there is little point in making a change if it does not move the student closer to the solution. In fact, if the difference function is properly defined, these two properties should always be met. However, the third property can be debated; sometimes, one needs to break a solution to make it better overall. While it is possible that this sort of backtracking may eventually improve a student's solution, it is unlikely that a student will apply a change if they see that it reduces their score.

Once the path construction algorithm has found the optimal goal for the solution state and identified all valid intermediate states between the state and the goal it enters the third step, where it identifies a path using the intermediate states to lead from the solution to the goal. To choose the intermediate states that will be used, we identify several properties that can be used to compute a state's desirability.

- **Times Visited**: a state that has been submitted before is a state that we know is fathomable; otherwise, it would not have been generated by a student in the past. This metric does not ensure that the state is good, or even reasonable, but it does provide some confidence in the state's stability.
- **Distance to Current State**: it is best if a state is close to the student's original solution; this metric ensures that the student will not need to make too many token edits based on the hint.
- **Test Score**: a stable next state should do as well on the test cases as possible, to ensure that the student makes good progress towards the goal.
- **Distance to Goal State**: the state should be as close to the goal as possible, so as to lead the student directly there.

After ranking all of the states with a desirability metric, the state with the best score is set as the first state on the path to the solution. The algorithm then recursively identifies each of the next states that would follow the first by locating all states between the chosen state and the goal (all states that have change vectors containing the vectors in the first edit) and iterating on this step. This process will eventually generate an entire solution path that extends from the original state to the goal.

These generated states represent the high-level points that the student needs to traverse in order to reach the end state. In order to reduce them into edits that are more reasonably sized (as we do not want to release too much information at once), we break each edit down further into token edits. A token edit only changes one token in the AST at a time, usually by providing the highest token value in the new subtree and replacing the sub-components of that value with filler strings. These token edits can then be given to a student individually, to give them one small hint step at a time. Currently we only provide token-level hints, but there is still potential for the use of step-level hints in the future, if we determine that students require more information than what a token-level hint provides.

## Reification: A Process for Hint Instantiation

After the entire path construction process has finished, it is necessary to transform the edit back towards the original representation of the student's code, to specify the actual location of the edit and the literal change that needs to be made. If this revision is not made, the resulting message will only confuse the student, as it may seem to refer to code that does not appear to be in their solution. Therefore, we use *state reification* to undo the canonicalization transformations in the code that have caused changes in the change vector.

In the preprocessing stage of canonicalization each node in the AST is given a unique global ID, which is transferred to any equivalent nodes created during the following transformations. These IDs can be used to map most nodes in a normalized AST back to the nodes in the original AST, which accomplishes most of the individualization process as we can just replace the *old expression* part of the change vector with the equivalent node in the original program.

However, there are some canonicalizing transformations that cannot be undone just with this mapping process. These transformations affect the new expressions as well as the old expressions: for example, code that has had its order reversed or has been negated during the canonicalization process (usually by the ordering functions) needs to be reversed/negated in both old and new values. Undoing is accomplished by adding metadata to nodes that have been reversed or negated during the transformations. During state reification, if the old expression value has one of these metadata tags, the inverse function can be applied to both old and new expressions, so that both are reverted appropriately.

Another category of special reification involves the reordering of subtrees within the AST. This reordering happens to boolean operations that have multiple values, where those values may be combined and reordered by transformations; it also happens to comparison operations with multiple operations, which are simplified into multiple comparisons during preprocessing. These reordered expressions cause difficulty as they create different paths leading to the old and new expressions. To fix the path, it is easiest to move up in the tree until there are no more ordering differences between the original and new expressions, then generate a Change Vector at that location. Though moving up in the tree technically makes the edit look larger, it is still encompassing the same idea of what needs to be changed overall.

With the global id mapping and these few special undoing functions, state reification can appropriately map the edit back into the student's original context. The edit can then be used in hint templates to show the student what they should do next within their problem-solving process. Currently, ITAP provides two levels of hints: a point hint and a bottom-out hint. The former only tells the student where the change should be made and what type of change it is, while the latter provides all of the information needed to make the edit. The template for the bottom-out hint is:

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

An example showing how this template is completed is shown in Figure 3. *Location information* and the *context* come from the change vector's path, and demonstrate what line the edit occurs on and the immediate context of the edit. The *action verbs* come from the type of the change vector, and describe what the user should do. The *old and new values* come from the old and new expressions, to show the actual content that needs to be changed. The point hint template replaces the *new value* with a filler expression, so that the student can try to figure the problem out on their own.

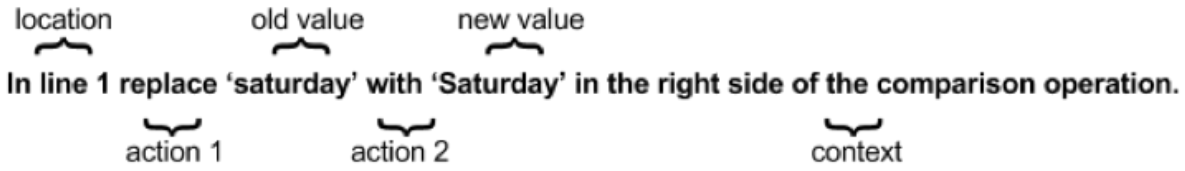


Figure 3: A hint generated using the second-level template.

## Syntax Hint Generation

The primary ITAP algorithm requires that the provided code be compilable; in other words, it has to be syntactically correct. Therefore, the main path construction method cannot be used to generate hints for syntax errors. However, we can still use a simplified version of the path construction algorithm to generate syntax hints.

Instead of using correct states as the goal states, the algorithm can use all compilable states that have been found so far. Each compilable state is more desirable than non-compilable states, as the first priority is to make the code compilable. Then, for each goal state, the algorithm can find the set of text diffs between the provided state and the goal (using the Python difflib), then attempt to optimize that set of diffs using the edit optimization method in path construction. Once all the diffs have been computed, the shortest set of changes is chosen, and the first of these changes can be provided as the syntax hint.

This algorithm is not particularly elegant, as it may remove a great deal of the student's code or add unnecessary lines. However, it will still eventually create a syntactically correct state. At that point, ITAP can switch to using the semantic hint algorithm, which will eventually lead the student to a correct state.

# Evaluation of ITAP

It is not enough to describe how an ITAP works as an algorithm; we also have to demonstrate that it is effective at its original purpose of providing hints that can assist students in learning. First, we performed a set of technical evaluations that determined how well each component of the ITAP framework functioned and how often hints could be generated. Second, we ran a pilot study to observe how novices would interact with practice problems, and how that interaction would change when hints were available. Finally, we ran a large-scale randomized classroom study to determine the effects of practice problems and hints on student learning. We describe the results from each of these evaluations in this section.

## Technical Evaluation

In performing a technical evaluation of ITAP, there are two main questions that need to be answered. First: how necessary is canonicalization? Does this modeling process improve the process of hint generation noticeably? Second: how effective is hint generation? Are hints being generated, and are they created efficiently enough for students to use them?

For this analysis, we used the data collected in Study 0 (described later), programs submitted by students during practice-problem work. The data set includes many incorrect states in addition to final solutions. Each analysis is based on the submitted programs for a set of thirteen individual programming problems that cover concepts ranging from basic expressions up to loops. Each of these problems was attempted by at least ten students, and each problem's state set includes at least five incorrect states.

### Technical Evaluation: Canonicalization

First, we address the question of the necessity of canonicalization. The first metric to look at is how much canonicalization affects the size of the solution space. If it can reduce the number of unique states, it may be easier to map new states into the space as well. For this evaluation, we only looked at the states in the data set that were parsable (as only these states can go through canonicalization); on average, a problem had 69 parsable states. There was an average 27% reduction rate between the original states and the cleaned solution space, where states were standardized in terms of whitespace and comments. Applying canonicalization resulted in an additional 7% reduction, leading to a solution space that was, on average, 66% the size of the original space. It is worth noting that the reduction rate decreases as problems grow more complex (as identified by the original problem ordering), as is shown in Figure 4.

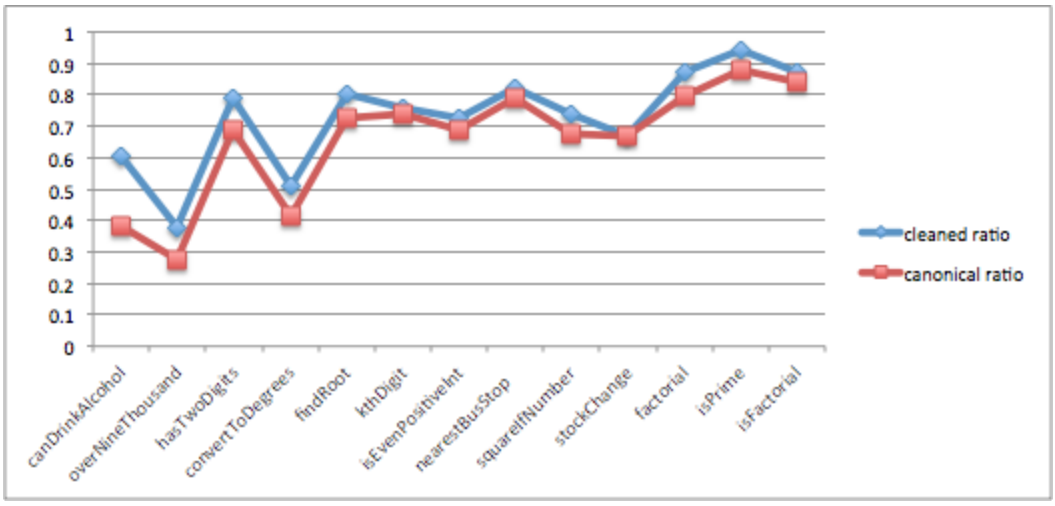


Figure 4: As problems grow more complex, the solution space reduction due to cleaning and canonicalizing states grows less effective.

Next, we address the question of how canonicalization affects hint generation. In ITAP, canonicalization is performed before path construction, with the goal of making the path from state to goal shorter. To see if path length reduction is actually achieved, we can compare the lengths of chosen paths in a version of ITAP with no canonicalization to a version where all states are canonicalized. (Note that in the actual version, path construction is performed using both canonicalized and original states, to ensure that the best possible path is found). We do still anonymize states in the no-canonicalization version, to support other features of path construction that rely on anonymous variable names.

Analysis showed that path construction with anonymized states had a median 8.45 token edits across all problem averages, while path construction with canonicalized states had a median 7.7 token edits. Though this is a small improvement, it does still make the path shorter, meeting the original goal. It is possible that even more improvement would be seen if canonicalized states were compared to original states, instead of anonymized ones; we leave this analysis for future work, as it requires some refactoring of path construction.

## Technical Evaluation: Hint Generation

Now that we have examined the effectiveness of canonicalization, we can turn to path construction and ITAP as a whole to see how well it performs at hint generation. In these analyses we will use an evaluative process called *hint chaining*. In hint chaining, we start running ITAP with a single state, apply the provided single-token hint (as an edit), then repeat the process with the resulting state. This process is repeated until ITAP reaches a correct state, with the process cutting off at forty edits to avoid infinite loops. The process lets us simulate a student following the bottom-out hints all the way from their initial state to the final state, as if they were using the problem as a worked example.

First we ask a simple question: how often can hints be generated in ITAP? To test this question, we attempted to perform hint chaining on each state in our data set (with states run in the order

initially submitted, starting with a single correct solution), comparing states that reached a correct state to those that did not. In 11/13 of the problems, ITAP was able to generate full hint chains 100% of the time; in the other two problems, ITAP achieved 98%+ success, though it did encounter infinite loops (where the hint chain would oscillate between two different goals) due to conditional transformations in a few of the states. Therefore, we can state that in the vast majority of cases, ITAP can reliably generate next-step hints.

Next, how efficient is hint generation? We may be able to almost always generate hints, but that does not mean that they will be created quickly enough for students to use them. In this analysis, we again perform hint chaining, this time examining at how long it takes to generate each next step in the chain. Over the thirteen problems, a median hint generation time of 4.85 seconds was achieved. Five seconds should be reasonable for most students, as running test cases commonly takes up to five seconds as well. However, there was high variance between the problems. While eight of the problems took fewer than five seconds on average to generate a hint, the other six ranged from an average of 5.13 seconds to 49.70 seconds in the worst case. This range of times seems to mainly be due to the use of loops in the three most time-consuming problems; the hint generation process commonly produces infinite loops while testing intermediate states, and these draw out the process more than may be necessary. Future work into identifying states with infinite loops and cutting off the testing process earlier may help reduce the time taken.

Finally, how usable are the hints? This question is tricky to measure without human coding, as the best measure of usability would be to have students or teachers rate the generated hint messages. As a proxy for this measure, we examine student reactions to hint messages in Study 0, to determine how often hint requests led to completed problems. Since our pilot study (described below) showed that students mostly used hints when they were truly struggling with a problem, this is a decent measure of whether the hint message helped students with solving the problems they struggled with most. Out of 33 problem-student pairs (instances where a student asked for at least one hint in a problem), students solved the problem in 20 of the pairs (61%) and did not solve the problem despite attempted submissions in 8 of the pairs (24%). In the remaining 5 cases, students asked for a hint at the very beginning of the problem, then did not ever attempt to submit to ITAP. It is possible that these students either were only interested in seeing how the hint feature worked or decided that the problem was too difficult for them. While these numbers can certainly be improved, they do demonstrate that the generated hints are proving useful to students over half of the time. We intend to see if improved versions of ITAP generate even better results in future studies.

Overall, the technical evaluation demonstrated that ITAP can theoretically perform well at generating hints for students in real classroom settings. However, the system needs to be tested with real students to determine whether this theory is true, which leads to the following studies.

## Pilot Study

In January 2015, we ran a short pilot study in the lab with researcher observation to test the effectiveness of the ITAP algorithm on student performance in a programming practice problem system. We designed a study where participants would be randomly assigned to one of two conditions, *control* or *hint*. Participants were given 15 minutes to complete up to four practice problems online (the pretest), had 30 minutes to practice with a set of seven problems, then spent 15 minutes on four problems (the posttest). The pre and post tests were counterbalanced, with problems designed to only cover material taught in the first week of a programming course, and in the middle problem set the hint condition students had a hint button added to their interface. We recruited 15 students from one of CMU's introductory programming courses, 15-112, during the first week of the course. 8 students were randomly assigned to the hint condition and 7 were assigned to control.

Initially we planned to use the pre-post differences to determine whether hint access had an effect on student learning or programming processes. However, perhaps due to the small sample size, the subjects in the two conditions were not comparable: students in the control group were almost all at ceiling (completing all problems) at the pretest, while students in the hint condition had much more trouble with the pretest problems (half at ceiling). Therefore, we were not able to compute any statistical results. However, we did still use the collected data and the observation notes of the main researcher to look for useful observations.

First, in examining the logged data from student work, we were able to confirm that students did generate many erroneous states while working. 294/585 (50.3%) of the logged states were syntactically correct but semantically incorrect, demonstrating that there was a primary need for semantic assistance; an additional 120/585 (20.5%) of the states were syntactically incorrect, demonstrating a smaller but still existing need for syntax assistance (which was implemented after the study).

The researcher's observations noted many trends that influenced the future design of ITAP. First, we found that students would first go to Google or the course notes when they encountered a problem, where they would attempt to learn more about the programming constructs they were using or what the error messages they received meant. Hints tended to be used after the student could not figure out the problem using other resources. In this study, ITAP presented three levels of hints (location only, location and change type, and location, change type, and changed value); students tended to skip over the second level, so it was removed in later iterations. Students did not always make the change that the hint recommended; sometimes, they would instead use it as a pointer for where they should be looking in their code, where they would go on to make their own changes (which could be unrelated to the hinted next step). The best use case for the hint system occurred when a student had a typo in their program that they were unable to find on their own; ITAP identified it and showed the correct variable name, and the student was able to fix their program immediately.

## Classroom Study 0

In Fall 2015, we ran our first real classroom study with college students in 15-112, an introductory programming course at Carnegie Mellon taken primarily by engineering majors. In this study, all students were given access to an online practice problem site called CloudCoder (Papancea et al. 2013), where they could log in with an anonymous id to work on basic programming problems, which had been designed to align with the material they were working on in class. A new set of problems was released at the beginning of each week, to encourage students to visit the site on multiple occasions. Student participation was entirely optional, and use of the system was anonymous and had no direct effect on their grades.

Students were randomly assigned to two conditions at the start of the semester. The first condition had a hint button visible in the problem-solving interface that, when clicked, would generate a hint for them based on their current code using the ITAP algorithm. The second condition had a normal problem-solving interface, which gave test case feedback but no hints. After three weeks, the conditions switched, so that the second condition had access to the hint button; after six weeks, both conditions were given access to the hint button. With this design we planned to test the effect that access to hints had on student learning in the class, by comparing student performance in the first quiz of the class to their performance on the following quizzes (with quizzes given at the end of every week).

Out of 419 students in the class, 97 logged into the system at any given point, and 62 started at least one problem (opening a problem and seeing the programming interface). Of these 62 students, five requested that we not use their data for research purposes; in the remaining analysis, we only refer to the remaining 57 students.

First, we examined the effect of student starting performance (score on quiz1, which was administered before students had access to the system) on interaction with the practice problem system, ignoring students who dropped out of the course. We found that pretest scores of students who logged into the system were significantly lower than scores of students who did not (82.85 vs. 86.36, $p < 0.03$). There seemed to be a higher ratio of students in the 70-85 score range using the hint system, and a lower ratio of students in the 95-100 range. It seemed likely that students in the lower score range were probably more aware that they needed practice, and were more motivated to practice in the hope that it would improve their scores. We also looked for correlations between different student behaviors in the interface and their pretest scores, and found that there was a moderate negative correlation between pretest score and student pausing (where we define pausing as taking at least one minute between two submissions within a problem) and a moderate negative correlation between pretest score and student submission rate (the number of submissions given per problem). In other words, students who did worse on the pretest paused and submitted more often within individual problems.

We then ran a linear regression using various features that described student use of the system: whether the student logged in, whether they ever opened a problem, whether they ever submitted code to a problem, whether they completed problems (and how many they

completed), and whether they ever used hints. Since most use of the system use occurred within the first two weeks of the course, we examined the effect that these variables had on student learning between the first quiz (before they started using the system) and the second. We found that logging into the system had a significant positive effect on student learning between quiz1 and quiz2, as is shown in Table 1. In other words, choosing to use the practice problem system resulted in significantly more learning. Most importantly, this happened in a population that was previously under-performing! This effect is illustrated in Figure 5. Of course, we must consider that this effect may not be due to the actual use of the system; it is possible that it was the internal factors of the students that led to them choosing to log in, and these same internal factors might have led to greater learning. This caveat is especially important as we did not have enough statistical power to demonstrate an increased learning effect due to student attempts or completion of practice problems. We hope to investigate this question further in the proposed thesis work.

| Q2 Score | Estimate | Std. Error | P |
|---|---|---|---|
| Intercept | -0.06211 | 0.05163 | 0.230 |
| Q1 Score | 0.46285 | 0.05081 | < 2e-16 |
| loggedIn | 0.24488 | 0.11942 | 0.041 |

Table 1: A linear regression demonstrating the effect of logging into the online system on quiz score (with quiz scores normalized over the entire population).

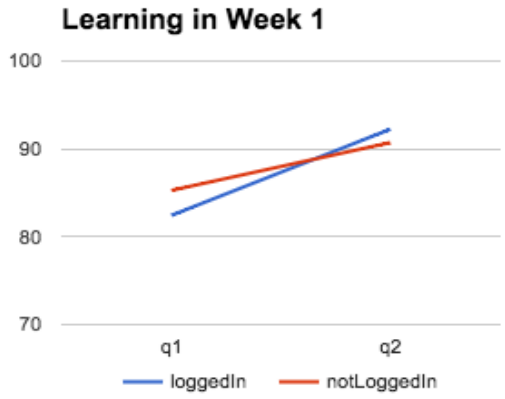


Figure 5: The effect of logging into CloudCoder on learning between quizzes 1 and 2.

Finally, to see if there was an effect of condition on student work, we compared the two conditions over weeks 1-3 (only nine students started at least one problem in the second phase, so we needed to restrict our analysis to the first phase). 24 students were in the control condition, and 23 were in the hint condition. We compared a range of different variables, but the

only significant difference between the two conditions appeared in the problem dropout rate. In this case, we define *problem dropout* as a student opening a problem in the interface, then never clicking the Submit button (to test their code). In the control condition, 1/24 students dropped out; in the hint condition, however, 7/23 students dropped out (significantly different by a fisher exact test, $p < 0.05$).

Closer investigation revealed that one of these dropout students typed in the editor (without ever submitting or asking for a hint), two asked for hints but never typed in the editor, one asked for a hint, typed in the editor, asked for another hint, and then dropped out, and the remaining three did not interact with the problem at all. The one student in the control condition who dropped out also did not interact with the problem. If we only compare dropout rates for students who did not interact with the problem, the difference is no longer significant ($p = 0.3475$). Therefore, it seems possible that the existence of the hint button might have made the interface more confusing, resulting in fewer students clicking Submit. We plan to examine this effect further in future studies by including a brief introduction to the system when a student first opens the website, and by including an inactive hint button in control condition interfaces.

# Proposed Work

In past sections I have described the work we have done in developing ITAP and pilot testing it with students. Though this system shares many traits with traditional tutoring systems of the past, it is also novel in several ways, especially in its lack of cognitive modeling and the open-ended student input it parses. Additionally, findings from the pilot indicated that students in introductory programming had diverse reactions to intelligently tutored practice problems, which could affect studies run on these tutors in the future. Therefore, I need to more thoroughly investigate how students think about and interact with programming tutors, and how those tutors affect their learning, to inform how future data-driven intelligent tutoring systems should be designed for open problem solving work.

For my thesis research, I want to determine whether ITAP is meeting its primary goal of assisting students in learning how to program. To answer this question, I plan to first investigate how *internal motivation and help-seeking activities* affect how students practice programming; once that has been established, I will study how *performance and learning* are affected by access to different types of feedback, especially hints. Taken together, these research questions will inform how ITAP is used by programming students in the future.

## Research Questions

I plan to address four research questions in my thesis work. The first is an extension of my previous technical work, while the other three are extensions of the system evaluation.

**RQ1:** *How much does ITAP automatically improve its performance as it accumulates student data, and at which points does it reach reasonable quality and essentially stop improving?* This

question extends the technical evaluation already done on ITAP by addressing an essential feasibility question of data-driven tutors: how efficiently can they be produced, and how much data is required to make the tutor perform well?

**RQ2:** *How do internal motivational factors and help-seeking practices affect how students interact with programming problems and differing feedback types?* This question addresses whether the different interaction patterns students exhibit when using programming tutors are due to set beliefs about learning programming and how feedback should be used. The results may help us understand how feedback, hints and practice need to be framed so that students will both use and benefit from them.

**RQ3:** *Does access to hints improve students' programming performance?* This question tests a basic premise of whether receiving hint assistance while programming results in students completing more problems or completing problems faster (thus improving their performance).

**RQ4:** *Does access to hints improve students' programming learning?* The final research question tests the central question of whether receiving hints while programming results in better learning outside of the immediate programming activity.

To address these research questions, I plan to conduct an extended technical evaluation and run three studies (two classroom studies and one usability analysis). The results from each evaluation should provide a foundation for the following evaluation. In the following sections I describe each of the planned evaluations.

## Technical Evaluation: ITAP as a Self-Improving System

So far, the technical evaluations performed on ITAP have sought to determine whether the system can function appropriately at the primary goal of generating hints. A secondary goal of the system has always been to generate tutored problems with very little human input and observation, to make intelligent tutoring easier to develop and more approachable for teachers. I have shown that the tutors can be generated with very little data, with the caveat that the resulting hints may not always be targeted at students' intended solutions. However, theoretically, the generated hints should improve over time as more data is gathered, with hints getting consistently closer to what a teacher would provide. If this is the case, then ITAP would be a self-improving tutoring system, which would be very valuable in a learning ecosystem.

To see if ITAP can improve generated hints based on student-submitted solutions (RQ1), I plan to test whether the distance between a submitted programming state and an ITAP-chosen goal decreases over time as ITAP gathers more data, and if there is a point at which this reduction plateaus. A similar evaluation was run with the Hint Factory (Barnes & Stamper 2008) to determine how many student attempts were required to achieve coverage of the solution space. Barnes and Stamper found that their coverage fit a power curve, covering a large portion of the solution space at the beginning but plateauing towards the end (only eight student attempts were needed for 50% coverage, but eighty were needed for 80% coverage). Since ITAP can

generate hints for states that have not been seen before, the Hint Factory coverage evaluation does not directly apply; instead; I plan to investigate how the distance between student state and ITAP-chosen goal state changes over time, to see if it fits a similar power curve.

To run this evaluation, I will develop a test harness system that uses sets of student-written solution attempt sequences collected in previous evaluations to evaluate ITAP's performance as data accumulates. To better model the variety of attempt sequences in real ITAP use, the attempt sequences will be randomized by student (so that a student's work is kept in order) to create many different test set sequences. Each test set will be run through ITAP sequentially, starting with a single correct solution as the solution space, with ITAP updating the solution space after each solution attempt is added. For each new solution attempt, the test harness will log the number of edits between each provided state and the chosen goal state. The edit count at each position will then be averaged over all attempt sequences to see what effect the number of previously-seen states has on the number of edits between state and goal. I expect to find that the distance decreases in a slow curve and then plateaus as the main variations on the final solution are all found, as was seen in the Hint Factory research.

This technical evaluation will be conducted simultaneously with the other three stages of research. The same evaluation method may be re-run each time new data has been generated by students. Doing so tests the robustness of the original findings and whether results are specific to or generalize across different student samples drawn from different kinds of courses.

## Classroom Study 1: Motivation and Help-Seeking

Study 1 is a classroom study that began this spring (January to May 2016) in the two introductory programming courses at Carnegie Mellon University. This study primarily examines the effect of internal motivational factors and help-seeking practices on student interaction with practice problems and feedback types (RQ2). I use a randomized control trial experiment, a series of surveys, and in-depth interviews with the students themselves to gather data.

Study 0 revealed a variety of student responses to the online practice problem system (as described above). In Study 1, I plan to probe whether this variety is due to intrinsic factors that students bring to the course, to see if I can influence student participation in future practice opportunities. This question is important to investigate as previous research has shown that help-seeking behaviors may affect problem solving, but do not always change the end learning result, though hints themselves have been shown to improve student learning (Aleven et al. 2016). Therefore, I am investigating a set of motivational factors that have been found to affect student learning and performance in the past:

- **Grit**: the ability to be persistent and finish work (Duckworth et al. 2007).
- **Achievement goals**: the internal goal(s) that a student has with respect to what they want to get out of the course (Elliot & McGregor 2001).
- **Mindset**: how the student perceives their own intelligence and ability to learn (Blackwell et al. 2007).

I also plan to investigate how students perceive help-seeking and how they engage with it during their course of study. These factors will all be measured using the surveys and interviews described later in this section.

In the study, students in 15-110 (~200 students) and 15-112 (~500 students), the two introductory programming courses at Carnegie Mellon, were given access to an online version of ITAP where they can choose to complete as many problems as they would like at any time during the first 5-6 weeks of the course. These problems are separated into labeled categories that cover a variety of topics: Functions, Data and Expressions, Logic and Conditionals, Iteration, Strings, Collections, and Recursion. Students are able to work on the problems in any order that they choose, as all problems are available from the start of the course.

As in Study 0, ITAP will provide two different kinds of feedback to students. The control feedback type is the Test button, which runs test cases on a student's code to determine if it is correct and shows the results. There is also a Hint button, which uses ITAP and the student's current code to generate a next-step hint. At the start of the experiment, students will be randomly assigned (with stratification across courses) to one of two feedback conditions: Test-only (T) or Test-and-Hint (H). The assigned condition will only affect which of the feedback buttons are activated (clickable) for the student; otherwise, the interface will look the same for everyone. At the beginning of Week 4 (the halfway point), student conditions will be swapped, to allow all students the same opportunity. After Week 6, I will activate all feedback types for all students, so that problems are available for review with hints later on.

In an attempt to isolate student personality from the effect found in Study 0 of logging in on learning, students were randomly assigned to two different encouragement conditions, where half of the students received a control statement in the introductory email (which simply reminded them that ITAP did not substitute for coursework) and half got a statement encouraging them to use ITAP (which added that previous studies have suggested that working on practice problems increases learning). If students in the encouragement condition are more likely to access ITAP, I can compare the learning of the students across the two conditions to see whether the practice problems actually have an effect. If the encouragement condition has no effect, I will instead compare motivational factors from the first survey between students who did and did not complete practice problems, to see if there are any substantial differences.

Throughout the experiment, there will be three optional surveys provided to students to measure their personal motivation and help-seeking beliefs. The first survey was included in the initial email sent to students with ITAP login information, the second survey will be sent to students at the Week 4 mark, and the final survey will be sent to students at the end of the semester, after the final exam. All three of the surveys include questions on perceived programming knowledge, the motivational factors mentioned above, and help-seeking behaviors. The questions related to motivational factors have all been taken from the foundational studies relating to the concepts (Duckworth et al. 2007; Elliot & McGregor 2001; Blackwell et al. 2007); the other questions have

been written to best capture the information I wish to investigate. The survey questions and their response types are included in Table 2.

| | |
|---|---|
| Self-rated Knowledge | How would you rate your knowledge of programming [before taking intro programming/at this point in the course]? (no knowledge to expert knowledge) |
| Grit | Likert scale agreement: I try to finish whatever I begin. |
| Achievement Goal | Likert scale agreement: It is important for me to do better than other students. |
| | Likert scale agreement: I worry that I may not learn all that I possibly could in this class. |
| | Likert scale agreement: I just want to avoid doing poorly in this class. |
| | Likert scale agreement: I want to learn as much as possible from this class. |
| Mindset | Likert scale agreement: I have a certain amount of intelligence, and I really can't do much to change it. |
| | Likert scale agreement: I can always greatly change how intelligent I am. |
| Help-seeking | Likert scale agreement: [In other courses/In this course], I have completed practice/review problems to help myself learn the course material. |
| | Likert scale agreement: [In other courses/In this course], I have gone to office hours to get help from the TAs/teacher with learning the material. |

Table 2: Questions included in all three iterations of the survey

The first two surveys will also contain an item recruiting students to participate in the interview section of the study, and every survey will ask for the participant's student id number, to tie the survey results back to ITAP's log data.

All three surveys will also contain a few independent questions related to the time at which the survey is released. The first survey will include three questions to measure students' prior ability, the second survey will include two questions to gather feedback on ITAP, and the third survey will include a range of questions to measure student satisfaction and gather feedback. These questions are all included in Table 3.

| | |
|---|---|
| Survey 1: Prior Ability | Have you used any programming languages before this course? (Y/N) |
| | If you answered 'Yes', select all the languages you've used. (checkboxes) |
| | Approximately what is your Math SAT score? (short answer) |
| Survey 2: Feedback | Likert scale agreement: ITAP has helped me learn the programming material covered in class. |
| | Do you have any suggestions for how we can improve ITAP? (open text entry) |
| Survey 3: Satisfaction | How well do you think you met your course goal(s)? (Not at all to Very well) |
| | Did you ever use the practice problem system this year? (Y/N) |
| | Briefly describe why you did or did not choose to use the practice problem system. (open text entry) |
| Survey 3: Feedback (only given to students who did use the practice problem system) | How often did you use ITAP? (once a semester to multiple times a week) |
| | Likert scale agreement: The ITAP system has helped me learn the programming material covered in my class. |
| | Which types of feedback did you use while solving problems in ITAP? (checkboxes: Test, Hint) |
| | Were the feedback types you used helpful? For each feedback type, why or why not? (open text entry) |
| | Do you have any suggestions for how we can improve ITAP? (open text entry) |

Table 3: Additional questions included in the three surveys used in the study.

There will be an optional interview component conducted during the first six weeks of the course. Students who sign up to participate in the interviews will be brought into the lab for half an hour. The interviewer will attempt to cover the following questions to determine what the student thinks about help-seeking and practice problem use. These questions were generated to best reflect the questions I had after running Study 0 and are based on interview question design principles from usability research (Goodman et al. 2012).

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 will collect external measures of student learning by receiving quiz and exam scores from the class' teachers. Additionally, I will collect information on students' departments and years of study from the school's online directory (prior to data anonymization), to use as additional factors in testing.

In analysis, I plan to use the collected data to run linear regressions that determine which factors affect use of the online practice problem system (in terms of login rate and problem completion) and learning (as measured by quiz/exam scores). I also plan to compare students' self-reported motivational and help-seeking factors in the three surveys to see if they change over time; while I do not expect to see a large effect, there is a small possibility of finding interesting effects. The logged programming work will be broken up into multiple factors (such as loggedIn, openedProblem, submittedProblem, solvedProblem, etc.) that can be used to determine if practice work had an effect on learning. The interview notes will be reviewed in an attempt to glean understanding about how students view help-seeking and how that should change the design of future tools.

## Usability Analysis: Help-Seeking and Performance

In the summer of 2016 I plan to conduct a usability study using guided think-aloud methodology (Ericsson & Simon 1998) to determine how students react to different kinds of feedback and different help-seeking situations, and how these situations influence their ability to solve programming practice problems. This study will be informed by the interview results from Study 1, and will address RQ2 and RQ3 by having users actively report their reactions to different help-seeking situations and measuring how programming performance is changed by different types of feedback and assistance. The results will inform the final design of ITAP prior to classroom Study 2, and may provide more in-depth insight into the relationship between help-seeking beliefs and programming performance.

I plan to recruit subjects on campus during the summer of 2016 who have minimal programming experience (or moderate experience but no experience with Python). These subjects will be invited to participate in a one hour lab session where they will complete a short pre-survey to establish their previous experience and other relevant information, then work on different programming problems online while thinking aloud (with occasional prompts from the observing experimenter). The activities will be designed based on the observations made in Study 1, but will explore variations in available feedback types (test cases, hint messages, and worked examples, which have been shown to be just as effective as problem-solving activities (Mostafavi et al. 2015)), student vs. system control of feedback delivery, and level of detail/amount of information provided in hint messages. Audio will be recorded during the think-alouds for later analysis, in addition to the notes taken by the researcher.

Evaluation of this study will be conducted by comparing user comments during the different tasks to their performance in task (whether they completed problems and how long it took them). These results can also be connected to the data collected in initial surveys to see how a student's prior knowledge and help-seeking beliefs are connected to later performance. Additionally, I will examine the programs generated by users in the different tasks to see if the generated programs are different in terms of style, efficiency, or diversity.

## Classroom Study 2: Performance and Learning

The final study will seek to measure the impact of hints of performance and learning (RQ3 and RQ4) by conducting a randomized control trial classroom study with increased statistical power (via increased student exposure to hints). This study will serve as a strong indicator for whether hints actually affect how students perform and learn.

In this experiment I aim to have students taking introductory programming in Fall 2016 work on practice problems as a required assignment, to increase participant count. In this study, I plan to have hint-condition hints appear automatically instead of on request, to increase exposure. This plan may change based on the findings of the usability study, but the study design will still attempt to increase statistical power so that, if the hints have an effect on learning, it can be directly measured. As in previous studies, control and hint conditions will be counter-balanced across two assignments, to provide students with equal opportunity while still allowing measurement of learning across conditions.

In evaluation, I intend to look at the effect of access to hints on performance and learning. Performance will be measured by looking at whole-assignment measures (how many problems could the student complete, how much time did they take) as well as in-problem metrics (how often did the students follow the provided hints). Based on previous studies on the effect of data-driven hints on student work, I expect to at least find that seeing hints leads to faster problem completion (Eagle & Barnes 2013). Learning will be measured by comparing the students' quiz scores before the first assignment, between the two assignments, and after both are done, to see if there is a difference between the two conditions. I will also compare the code

generated by the two different groups in each assignment, to see if the hints affected the style and diversity of student solutions.

Since this evaluation depends on homework-associated use of ITAP, I will need to work with the teachers of involved courses to ensure that they are comfortable with the study, which should be facilitated by the fact that I have run studies with these teachers before. If necessary, we may also make the grading of the assignment participation-based, so that students are not penalized for their assignment to different conditions. If possible, I also hope to add questions to the quizzes that can be used as direct measures of learning (with questions using similar knowledge components across all three quizzes); if this plan does not work out, I will instead incorporate pre- and post-test questions into the assignments that can be used to provide more direct comparisons.

## Expected Contributions

Overall, I expect that my work will make contributions to the fields of human-computer interaction, learning science, and computer science education.

In human-computer interaction, ITAP has already provided a conceptualization for how systems can shape user data to provide new users with assistance. If the proposed technical evaluation demonstrates that ITAP is a self-improving tutoring system, it may also provide insight into how systems can automatically upgrade themselves based on user interactions. The proposed work on help-seeking will also provide insight on how software designers can best support users in educational contexts by designing assistance that will actually be used.

In learning science, ITAP has extended the work of the Hint Factory to provide a new framework for data-driven hint generation that may be extended to many other domains in the future. My work on canonicalization also serves as a new concept for modeling student states, and may someday be extended to other educational fields as well. The classroom studies I plan to run will provide evidence for how effective this tool might be, and the surveys on motivational factors and help-seeking behaviors may provide insight into how a student's prior mindset affects their interactions with educational tools.

Finally, in computer science education, ITAP is a practical contribution that may soon see growing use in introductory programming classrooms. The path construction process can be adopted by all programming languages, and canonicalization has also provided a new framework for conceptualizing the representation of student programs, even though it is limited to Python programs at the moment. The interviews and surveys run in Study 1 may also provide insight into how students' motivational factors affect their interactions with programming classes, which could be used to inform course design in the future.

# Timeline

A detailed outline of my timeline is included below. I plan to defend my thesis in April 2017.

| | |
|---|---|
| January - May 2016 | Run Study 1 |
| February 2016 | Thesis proposal |
| February - March 2016 | Technical evaluation, Round 1 |
| June 2016 | Technical evaluation, Round 2 |
| June - August 2016 | Run Usability analysis |
| August - October 2016 | Run Study 2 |
| November 2016 | Technical evaluation, Round 3 |
| November 2016 - January 2017 | Write thesis document |
| February - March 2017 | Thesis revisions |
| April 2017 | **Thesis Defense** |
| May 2017 | More thesis revisions |

# References


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*, 1-19.

Barnes, T., & Stamper, J. (2010). Automatic hint generation for logic proof tutoring using historical data. *Journal of Educational Technology & Society*, 13(1), 3-12.

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).

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.

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. In Helander, M. G., Landauer, T. K., & Prabhu, P. V. (Ed.s) *Handbook of Human-Computer Interaction*, (pp. 849-874).

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*.

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).

Elliot, A. J., & McGregor, H. A. (2001). A 2x2 achievement goal framework. *Journal of personality and social psychology*, 80(3), 501-519.

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).

Gerdes, A., Jeuring, J. T., & Heeren, B. J. (2010). Using strategies for assessment of programming exercises. In *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 441-445).

Goodman, E., Kuniavsky, M., & Moed, A. (2012). *Observing the User Experience: A Practitioner's Guide to User Research.*

Gross, S., Mokbel, B., Paassen, B., Hammer, B., & Pinkwart, N. (2014). Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 10*, 9(3), 248-280.

Hicks, A., Peddycord III, B., & Barnes, T. (2014). Building Games to Learn from Their Players: Generating Hints in a Serious Game. In *Proceedings of the 12th international conference on Intelligent Tutoring Systems* (pp. 312-317).

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).

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).

Moghadam, J. B., Choudhury, R. R., Yin, H., & Fox, A. (2015). AutoStyle: Toward Coding Style Feedback at Scale. In *Proceedings of the Second (2015) ACM Conference on Learning@Scale* (pp. 261-266).

Mostafavi, B., Zhou, G., Lynch, C., Chi, M., & Barnes, T. (2015). Data-Driven Worked Examples Improve Retention and Completion in a Logic Tutor. In *Artificial Intelligence in Education* (pp. 726-729).

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).

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 *Data Mining for Educational Assessment and Feedback (ASSESS 2014)*.

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

Rivers, K. and Koedinger, K.R. (2015). Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 1-28.

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. (2012). A canonicalizing model for building programming tutors. In *Proceedings of the 11th international conference on Intelligent Tutoring Systems* (pp. 591-593).

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).

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).

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

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