Int J Artif Intell Educ (2017) 27:37–64
DOI 10.1007/s40593-015-0070-z

ARTICLE

Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor

Kelly Rivers[1] · **Kenneth R. Koedinger**[1]

Published online: 22 October 2015

Abstract To provide personalized help to students who are working on code-writing problems, we introduce a data-driven tutoring system, ITAP (Intelligent Teaching Assistant for Programming). ITAP uses state abstraction, path construction, and state reification to automatically generate personalized hints for students, even when given states that have not occurred in the data before. We provide a detailed description of the system's implementation and perform a technical evaluation on a small set of data to determine the effectiveness of the component algorithms and ITAP's potential for self-improvement. The results show that ITAP is capable of producing hints for almost any given state after being given only a single reference solution, and that it can improve its performance by collecting data over time.

Keywords Data-driven tutoring · Automatic hint generation · Programming tutors · Solution space

Introduction

The ability to write code accurately and fluently is a core competency for every computer scientist. However, learning to program is prohibitively difficult for many students, and this limits diversity in students that pursue computer science or gain programming skills to enhance their other careers. Novice programmers tend to make many mistakes on the path to becoming expert programmers, with the hardest-to-resolve mistakes occurring at the algorithmic level (Altadmri and Brown 2015). Students who receive help while programming do better in their courses, but it takes time to find a teacher or TA and have them look over code, and students have expressed

✉ Kelly Rivers
krivers@andrew.cmu.edu

Kenneth R. Koedinger
koedinger@cmu.edu

[1] Carnegie Mellon University, Pittsburgh, PA, USA

a need for more immediate help during their work process (Carter et al. 2015). Since teachers are limited in the amount of time they can spend helping students, an easy-to-access, automated help source would be a great benefit to students.

Intelligent Tutoring Systems (ITSs) are a natural solution to this need, as they are designed to give individualized feedback and assistance to students who are working on problems. In fact, ITSs have been designed for programming in structured editors in the past, and have been experimentally demonstrated to produce large positive effects on students learning (Anderson et al. 1989). However, traditional ITSs have a major drawback: they take much time and expert knowledge to create. A survey of several different ITS builders found that the ratio of tutor construction time to student interaction time is very high. Even systems which are simple to build have a 23:1 cost ratio, and most systems have ratios on the order of 100:1 or more (Corbett et al. 1997; Folsom-Kovarik et al. 2010). Building tutors the traditional way only takes more time as the domains being covered grow more complex. This complexity is especially evident for programming, where there are dozens of syntactic and semantic knowledge components and hundreds of algorithmic plans, as well as potentially infinite numbers of possible code states per problem (when the student input is not heavily structured). Even simple programming problems can have multiple separate correct solutions and hundreds of intermediate states, and each of these states can be rewritten in hundreds of ways by varying the code's ordering or adding extra code. In a domain with such vast solution spaces, a less effort-intensive tutor authoring technique is needed.

Instead of having experts provide the knowledge used within tutors, we can assemble that knowledge through the use of student data, particularly the solutions students write when they solve open-ended problems. In the programing domain, these solutions are the code written to solve a given programming problem. Student solutions can be used to construct a solution space consisting of all of the different states that students have created while working in the past, with paths between the states showing how to get from any given state to a correct state. The idea of data-driven tutor authoring is not new, but past efforts, such as Bootstrapping Novice Data (McLaren et al. 2004) and the Hint Factory (Barnes and Stamper 2008), rely entirely on the collected data to fill out the possible paths, giving help to students based on what others before them have done. This approach is severely limited for domains with large solution spaces, as it cannot provide help to students in states that have not been seen before and it does not guarantee that every student will always be able to get a hint.

In this paper, we address this limitation on hint generation and extend the concept of data-driven tutoring to more generally address the challenge of building ITSs in open-ended problem solving domains where *concrete* state coverage is impossible to achieve (i.e., there are infinitely many states, so it is impossible to anticipate what all of them will be). To do so, we have developed a process for tutor construction that goes beyond data reuse by automatically computing more *abstract* versions of observed states and constructing new paths in addition to the ones seen before. Our system, the Intelligent Teaching Assistant for Programming (ITAP) combines algorithms for *state abstraction*, *path construction*, and *state reification* to fully automate the process of hint generation. ITAP makes it possible to generate a full chain of hints from any new code state to the closest goal state. Further, ITAP is an instance of a self-improving ITS (Kimball 1982), a tutor that continually improves its ability to provide hints that are personalized to each students' individual solution to a problem. It does so by updating the solution space every time a student attempts a solution and recomputing optimal paths. ITAP currently recognizes correct solutions and provides hints on semantically or algorithmically incorrect programs; it currently is not implemented to handle syntactically incorrect programs.

In past work, we've introduced the concepts of canonicalization and path construction, and we've demonstrated how solution space reduction and goal individualization can be achieved using a diverse set of programming problem solutions (Rivers and Koedinger 2012, 2014). In this paper, we provide a more thorough description of all the sub-components of ITAP, including how the whole system ties together, and we perform a technical analysis using data collected from students working on programming problems in real time. This data makes it possible for us to test the system in a more realistic setting by using intermediate code at varying levels of completion, instead of relying on final submissions, which are usually almost correct. We also discuss the algorithm's potential for making self-improving tutor development easily accessible to teachers and researchers. Though we have not yet been able to perform an authentic evaluation of ITAP with students, we believe that the technical and theoretical contributions made by this paper (which are backed up by the technical evaluation) are supported by the past literature on the effectiveness of other automatic hint generation systems.

Background

A variety of systems have been built to provide tutoring services for programming problems. Many of these use approaches traditional to ITS design, such as providing examples, simulation, or scaffolding dialogue; a review of such systems is provided by Le et al. (2013). These services accomplish many of the goals of tutoring systems, but none of them provide next-step hints for code-writing problems.

Having personalized feedback such as next-step hints is an integral and highly beneficial aspect of intelligent tutoring system design, as was originally demonstrated with the Lisp tutor, which showed that greater control over feedback options led to more efficient learning (Corbett and Anderson 2001). Other studies have demonstrated that struggling students benefit from higher levels of tutor interaction (Razzaq et al. 2007) and that bottom-out hints can be treated by students as helpful worked examples (Shih et al. 2011), Hints generated from data have been evaluated in classroom contexts with positive results; for example, the Hint Factory was shown to help students complete more problems (Stamper et al. 2011) and help students complete problems more efficiently (Eagle and Barnes 2013). Therefore, it is important for us to design techniques for generating these valuable next-step hints.

Hint-enabled tutors for code-writing are rare, as they take much time and effort to create. However, there are a few examples outside of data-driven tutors which have been made. Constraint-based tutors provide one approach, due to the ease of using test cases as constraints; one example even uses Prolog patterns to model a solution space (Le and Menzel 2007). Another example is an interactive functional programming tutor for Haskell, which uses model solutions and teacher-annotated feedback to provide tutored support for various programming states. This tutor uses a library of strategies and rewrite rules to support a variety of solution types, which is very similar to our own state abstraction technique (Gerdes et al. 2012).

Data-Driven Hint Generation for Programming

Data-driven tutoring is a subfield of intelligent tutoring that bases decision-making within the tutor on old student work instead of a knowledge base built by experts or an author-mapped graph of all possible paths. The first iteration of data-driven tutoring with a focus on hint generation was started by the Hint Factory, a system for logic tutors which uses Markov decision processes to collapse student action paths into a single graph, a solution space (Barnes and 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 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 and Koedinger 2014). This makes it possible to generate hints for never-seen-before states, which the original Hint Factory could not do. This method 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 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 Bratko (2014) used student-generated text edits to correct programs over time, an algorithm that could sometimes support even non-parseable programs. Perelman et al. (2014) used all common expressions that occurred in code to create a database that 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 which 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 ues 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.

Implementation

In the following section we describe in detail how ITAP works by covering the algorithm for constructing a solution space and the methodology for representing solution states. We follow this description with a step-by-step demonstration of how the algorithm comes together to generate a hint for a single example.

ITAP requires a two pieces of expert knowledge to run independently, though this knowledge is kept to a minimum. The needed data is:

- At least one reference solution to the problem (e.g. a teacher exemplar).
- A test method that can automatically score code (e.g. pairs of expected input and output).

Both reference solutions and test methods are already commonly created by teachers in the process of preparing assignments, so the burden of knowledge generation is not too large.

Solution Space Construction

Once the exemplar solution has been added, ITAP can begin processing new data to construct a solution space for the problem. We define a *solution space* to be a graph of intermediate states that students pass through as they work on individual problems, where each state is represented by the student's current code. Directed edges are added to the graph to provide 'next steps' for every incorrect state, with edges eventually leading to correct states (where 'correct' states score perfectly on the provided test methods). These edges determine the optimal path of actions that a student can take to solve the problem.

In the next section, we describe how the path construction algorithm adds new states that have not been seen before to the solution space. We also explain how the edges between states can be broken down into hint-sized chunks with token-by-token hint chaining.

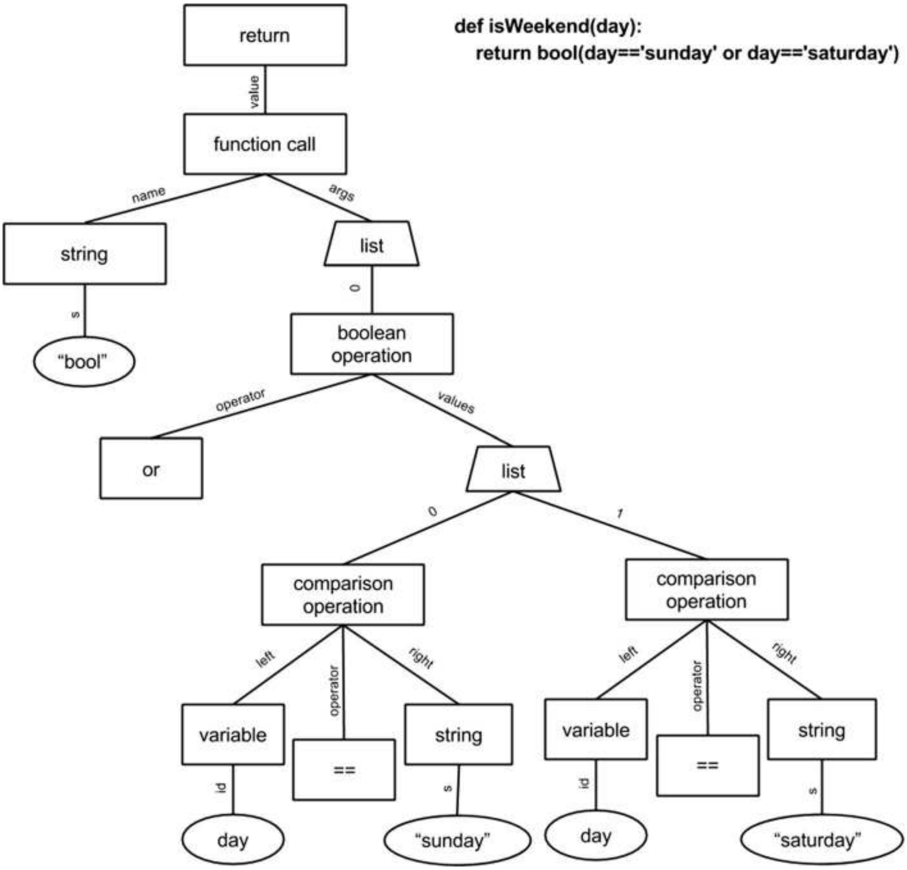
Path Construction Algorithm

Path construction (Rivers and 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 can identify new and personalized correct solutions for students. Generated hints take the form of edits between states, where edits are applied in order to change the current state into the 'next state'.

To define edits between program states, we represent program code with *abstract syntax trees* (ASTs). An AST is an intermediate representation of a program created by the compiler during the process of converting the program from text into binary code; an example is shown in Fig. 1. Since ASTs are stored as data structures (trees), 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.

We call the edits generated from AST comparison *change vectors* (see Fig. 2 for an example). A change vector has several properties:

- **Path:** a list of nodes and indices which can be traversed through an AST to get to the place where the edit occurs. This serves as a location marker for the change vector.



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

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

- **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 values.
- **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 which can be swapped with each other, and *Move* vectors identify a location that should be moved to a different place.

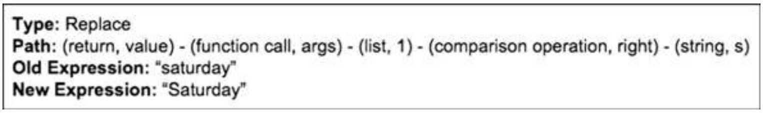
With the comparison method and change vectors, we can define a distance metric between ASTs which computes the percentage of nodes that occur in both trees. This will directly correspond to the number of edits needed to change one AST into another. With the comparison function and distance function, we can define the path construction algorithm.

First, we determine the *optimal goal state* for the given state. This 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). Figure 3 shows this comparison for a simple solution space.

Sometimes this first goal state perfectly represents the best goal state for the student. However, there are often differences between the current state and the goal state which do not change the results of the test cases. It is inefficient to give students hints for these types of edits, since they aren't 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.

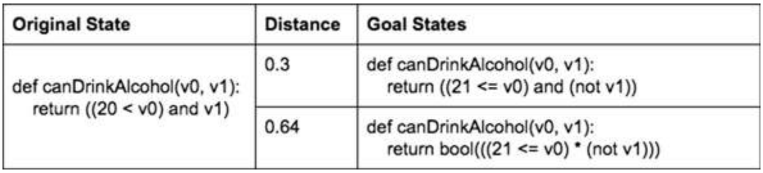
To reduce the edit set, we compute the power set of the set of edits, i.e., the set of all possible subsets of edits. Each subset of edits is then applied to the given state, to determine whether any of them lead to a correct solution. If there are some subsets which do lead to correct solutions, we compute their distances from the given state and choose the set which is closest to the given state as the final goal state (see Fig. 4 for an example). This new, personalized goal state is the closest current solution for the given student. Due to the exponential nature of power sets, we institute a cutoff of 15 edits for this process; states with more than 15 edits are given a simple path where the edge leads from the current state directly to the original 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



Type: Replace
Path: (return, value) - (function call, args) - (list, 1) - (comparison operation, right) - (string, s)
Old Expression: "saturday"
New Expression: "Saturday"

Fig. 2 An example change vector for the program shown in Fig. 1



Original State	Distance	Goal States

def canDrinkAlcohol(v0, v1):
return ((20 < v0) and v1)
	0.3	def canDrinkAlcohol(v0, v1):
		return ((21 <= v0) and (not v1))

	0.64	def canDrinkAlcohol(v0, v1):
		return bool(((21 <= v0) * (not v1)))

Fig. 3 An original state being compared to all the goal states in the solution space, and the resulting distances (which all fall between 0 and 1). In this case, the first goal is chosen

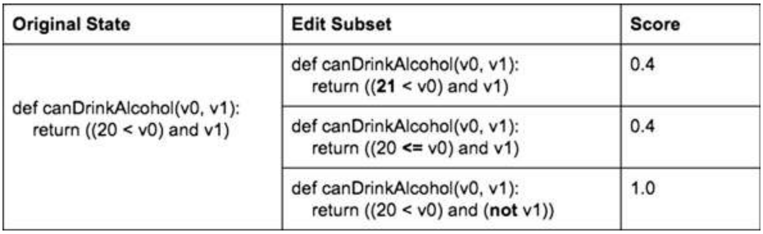
do much worse than the original state, and others might produce awkward code. We identify three properties that are required to identify an intermediate state as ‘valid’:

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, so we retain this property for the initial version of the algorithm.

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 which has been seen before is a state which we know is fathomable; otherwise, it would not have been submitted by a student in the past. This 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 ensures that the student will not need to make too many token edits based on the hint.



Original State	Edit Subset	Score

def canDrinkAlcohol(v0, v1):
return ((20 < v0) and v1)
	def canDrinkAlcohol(v0, v1):
	return ((**21** < v0) and v1)
		0.4

	def canDrinkAlcohol(v0, v1):
	return ((20 **<=** v0) and v1)
		0.4

	def canDrinkAlcohol(v0, v1):
	return ((20 < v0) and (**not** v1))
		1.0

Fig. 4 There are three edits between the original state and the original goal chosen for it, but testing subsets of these edits reveals that only one of the edits is needed (the third). By applying just this edit, we create a new goal that is personalized for the student

- **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.

We combine these four desirable properties to create the desirability metric shown in Eq. 1. This metric is used to rank the list of all valid intermediate states, to determine which will serve best as a 'next state'. The weights in the formula can be adjusted to reflect how important each of the properties are to the teacher. In our data set, we found that some of the properties were modestly correlated (such as closeness to solution and score), and adjusted the weights to account for this. Test_score is given the lowest weight, since it varies widely across intermediate states; times_visited and dist_goal are given equal weights, to serve as a baseline; and dist_current is given the strongest weight, to emphasize nearby states.

$$\begin{aligned} &0.2*\mathrm{times\_visited(s)} + 0.4*(\mathrm{dist\_current(s,n)}) + 0.1*\mathrm{test\_score(n)} \\ &\quad + 0.2*(\mathrm{dist\_goal(n,g)}) \end{aligned} \tag{1}$$

After ranking all of the states with this 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 which have change vectors containing the vectors in the first edit) and iterating on this step. This will eventually generate an entire solution path that extends from the original state to the goal.

The resulting path is a set of macro-edits that can be thought of as high-level hints. The states within these macro-edit paths can be added to the solution space, where they may possibly be used for future students as well.

Token-by-Token Hint Chaining

Though path construction can be used to generate macro-level steps through the solution space, these large steps give away too much information to be useful in a tutoring context, as a macro-level step often encompass multiple knowledge components. To address this, we further break up macro-level steps into *token edits*, where each token edit only gives the user one new token to change or add to their program. Delete and Super vectors, which only tell the user to remove code, remain unchanged, as do Move and Swap vectors. Chaining these token edits together can result in the macro-level step generated by the path construction algorithm.

Generating token edits presents two major challenges. First: which token in a complex program expression should be shown first? Many complex code expressions, such as boolean operations between two comparisons, require multiple knowledge components at different levels of hierarchy. Is it best to show the student the first token they would need to type? Or is it better to give them a token that represents the primary concept of the edit?

For now, we use the second approach, choosing to show a token from the top-level node of the new expression. In our example of a boolean operation, we would want to show just the operator, *and* or *or*. The resulting expression varies based on the type of change vector used; for example, for Sub vectors, we know that the student's current code is part of the new expression, so we can keep the current code visible while obfuscating the rest (as is shown in Fig. 5).

The second challenge involves displaying token edits. In order to make these token edits 'chainable' (capable of being applied one by one automatically), we need to express them such that they can be directly applied to the program without breaking the syntax. This also means that a confused student could treat the message as a bottom-out hint by directly copying the edit into their code without needing to interpret its meaning. However, this can be complicated for some code expressions. For example, going back to our earlier boolean operation example, it is not enough to tell a student that they need to add the expression *or* to their code; this will not compile, and does not make sense.

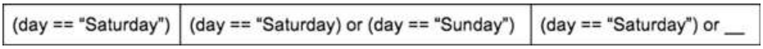
To address this problem, we replace all obfuscated tokens within the code expression with filler strings. These are strings which says things like '[put code here]' or '[extra stuff]', to make it clear that they serve only as placeholders. Python will parse most expressions successfully regardless of whether the types used make sense, so an expression such as *'[put code here]' and '[extra stuff]'* will parse just as well as *x>0 and x % 2 == 0*. With filler strings a student can see the structure of the code they need to write without being given all of the content inside of it.

Once the edits have been reduced to single tokens and obfuscated with filler strings, they can be successfully used as micro-level edits. And by iteratively generating macro- and micro-level edits, we can create solution spaces for new problems by automatically generating paths between states.

Student State Representation

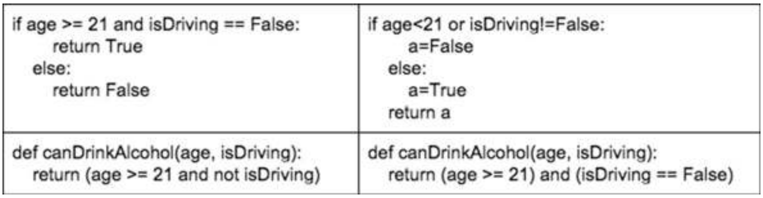
The construction of a solution space makes it possible for us to find paths from a student's state to a correct solution, even if we've never seen their state before. However, the hint paths generated during this process are not always optimal, as students have a tendency to code inefficiently. When examining student solutions it is common to see a variety of approaches that are identical semantically but incredibly different in their syntax; Fig. 6 has an example of this.

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 semantics of a problem. Therefore, to focus on semantics, it would be beneficial 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 using a set of normalizing functions to map student code into semantic equivalence classes. This process is called *canonicalization*, as it creates canonical examples of different code strategies (Rivers and Koedinger 2012).



(day == "Saturday")	(day == "Saturday) or (day == "Sunday")	(day == "Saturday") or __

Fig. 5 From left to right, the old expression, original new expression, and token-edit new expression (for a Sub Vector). The modified new expression shows the old expression and top-level token, but hides the rest



```
if age >= 21 and isDriving == False:
      return True
   else:
      return False
```

```
if age<21 or isDriving!=False:
      a=False
   else:
      a=True
   return a
```

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

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

Fig. 6 All four of these code examples are the same semantically (as they perform the same way algorithmically) despite their vastly different syntactic approaches

ITAP applies canonicalization at the beginning of the hint generation process, to simplify the path construction process, then reverses the abstraction at the end, so that the student's hint will still be personalized to their own code. In the following sections we describe how this is done.

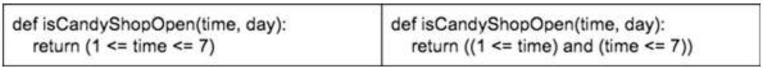
Canonicalization: an Abstraction Process for Solution States

We again use ASTs to represent code with data structures, as was done for path construction. This makes it possible to apply semantics-preserving program transformations to code. These transformations 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. In this section we describe the suite of transformations we currently implement.

We are not the first researchers to use program transformations in a computer science education context. 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 which were used to verify program correctness without using test cases. Several of our transformations are similar to the ones developed by these researchers.

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. This mainly involves simplifying multiple-target assignment (*a=b=4* to *b=4*; *a=b*) and augmented assignments (*a+=4* to *a=a+4*), and turning multiple-operator comparisons (*1<a<4*) into combinations of single-operator comparisons (*1<a and a<4*). Another example is shown in Fig. 7.

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 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.



```
def isCandyShopOpen(time, day):
   return (1 <= time <= 7)
```

```
def isCandyShopOpen(time, day):
   return ((1 <= time) and (time <= 7))
```

Fig. 7 An example of code before and after preprocessing

The final and most important step of preprocessing involves anonymizing all variable names used within the code, to standardize names across student solutions. To do this, we give each variable a new name as soon as we first encounter it. 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. An example is shown in Fig. 8.

Outside of these preprocessing transformations, the canonicalizing functions we use fall into three main categories: simplifying, ordering, and domain-specific functions. Each category is discussed below.

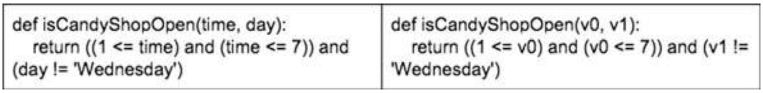
Simplification functions reduce the size of a program by collapsing values in the function and removing code that isn't being used. Four different functions are currently included in this category: constant folding, copy propagation, helper function inlining, and dead code removal. All four are staple transformations commonly used in compiler optimization.

Constant Folding is an optimization which reduces expressions that entirely involve constant values into their known results. For example, if (*x=2 * 5*) is included in code, it can be folded into (*x=10*). This also removes unnecessary boolean values in boolean operations (*x and True* is equivalent to *x*), simplifies various operations involving constants (*x * 1.0* can be restated as *float(x)*), and directly applies other constant operations whenever they occur, such as in comparisons or if expressions.

Copy Propagation is an optimization which propagates the values assigned to variables throughout the AST, replacing the variables entirely. This optimization effectively removes the unnecessary variables created by students by locating variable uses where the values used within the variable assignment do not change between the assignment and the use. As a result, code like (*x=5*; *y=6*; *print x * y*) can be reduced to (*print 5 * 6*).

Helper Function Inlining is used to collapse helper functions into the main function. This is especially helpful when students use helper functions for small expressions, which many students do as they are learning how to break up code into appropriate portions. Every time a helper method is called, we can determine whether it can be moved safely, which is true if there is a direct control flow line to the return statement, it doesn't change the state of the program, and it isn't recursive. Then the body of the function can be moved into the code, with the parameters replaced by variable assignments. This is demonstrated in Fig. 9. Additionally, we can move global variables into the main function if they are only referenced, and not modified.

Finally, *Dead Code Removal* is used to remove any part of the code which is not actually utilized during any call to the function. This includes any lines of code that follow a return statement, any assignment to a variable where the variable is not used in



def isCandyShopOpen(time, day):
return ((1 <= time) and (time <= 7)) and (day != 'Wednesday')
	def isCandyShopOpen(v0, v1):
	return ((1 <= v0) and (v0 <= 7)) and (v1 != 'Wednesday')

Fig. 8 Code before and after variable anonymization

the following lines of the function, and print statements which do not causes crashes or mutation in code.

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 is mainly accomplished through ordering commutative operations and adjusting certain nested values throughout the code. All ordering is done with a strict comparison function that is capable of comparing all types of AST nodes.

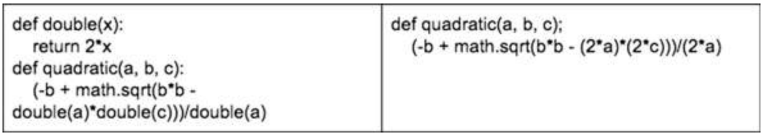
There are several commutative operations that exist within code which can be safely ordered without changing the effect of the code. This includes the obvious math expressions (addition, multiplication, bit operations), but can also be extended to less obvious types (boolean operations whose values won't crash and if statement branches). Additionally, we can impose order on comparisons by flipping all greater than/greater than or equal to expressions into their equivalent less than/less than or equal to forms. Before doing ordering, it's necessary to ensure that the values in the expressions used won't crash, as changing their positions could then change the results.

Several functions can also be used to change the ordering of nested statements. One common case is that of the *not* operator, which can be propagated into many kinds of statements. With boolean operations, we can use de Morgan's law to turn (not (a or b)) into (not a and not b); we can also reverse comparison operators (turning < to > =). Another common source of changed ordering is negations; expressions such as (−(a–b)) can become (−a+b), which can become (b - a). Negations can also be propagated within multiplications, and we can move unary subtractions into expressions. Finally, we can simplify nested boolean expressions by combining like values within them; for example, the expression *(a and b) or (a and c)* can become *a and (b or c)*.

Finally, there are several **domain-specific** transformation functions which are designed explicitly with novices in mind, functions which 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.

Four of these transformations are related to conditional statements, which novices have a tendency to write in awkward ways. All four are demonstrated in Table 1. The first checks for redundant lines within the if statement, lines that occur in both if and else branches, and moves them out into the main body of the code. The second looks for adjacent but disjoint if statements and combines them into if-elif statements. The third function seeks to combine conditional branches which do not need to be separate into a single test. Finally, the fourth transformation seeks to collapse if statements that can be rewritten as boolean statements. Novice programmers have an unfortunate tendency to use these entirely unnecessary if statements, and collapsing them makes it much easier to compare different solutions.

There are many smaller transformations outside of the conditional examples which can be used to target specific novice code oddities. We briefly describe three that we've



```
def double(x):
    return 2*x
def quadratic(a, b, c):
    (-b + math.sqrt(b*b -
double(a)*double(c)))/double(a)
```

```
def quadratic(a, b, c);
    (-b + math.sqrt(b*b - (2*a)*(2*c)))/(2*a)
```

Fig. 9 An example code before and after helper function inlining

Table 1 Examples of each of the four domain-specific conditional transformations

Transformation	Pre-transformation	Post-transformation

Redundant Lines	if (x == “Monday”):
	print “here”
	a = True
	else:
	print “there”
	a = True
		if (x == “Monday”):
		print “here”
		else:
		print “there”
		a = True

Disjoint Statements	if(x == “Monday”):
	return 1
	if (x == “Tuesday”):
	return 2
		if(x == “Monday”):
		return 1
		elif (x == “Tuesday”):
		return 2

Combined Tests	if (x > 5):
	return True
	elif (x < 0):
	return True
		if (x > 5 or x < 0):
		return True

Collapse to Return/Assign	if (x == “Monday”):
	return True
	else:
	return False
		return (x == “Monday”)

designed here. First, novices have a tendency to check *if (x == True)* instead of *if x*; this can easily be reduced to the latter form. Second, Python allows default values to be excluded in two of its commonly-used expressions, ranges and slices; we can remove the default values if novices have included them anyway. In other words, novice code that looks like *s[0:5:1]* is equivalent to *s[:5]*, and *range(0, len(s), 1)* is equivalent to *range(len(s))*. Finally, we remove unnecessary type mappings (that is, casts to a type where the value inside the cast is already of that type); *int(4)* can be simplified to 4.

It is almost certain that there are many other transformations that could be written, especially in the domain-specific category; the ones that have been developed so far were based mostly on study of relatively simple student programs, and many quirks will only show up in certain types of problems. However, the simplifying and ordering transformations transfer across all problems, and prove quite useful in collapsing different student code into the same semantic state.

Reversing Abstraction with State Reification

After the entire path construction process has finished, it is necessary to move the edit back into the student’s original context, to specify the actual location of the needed change. 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 [1]to undo the canonicalization changes in the code which 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- we can just replace the 'old expression' part of the change vector with the equivalent node in the original program.

There are some canonicalizing transformations which cannot be undone just with this mapping process, which need a small bit of additional transformation. This is generally due to transformations that will affect the new expressions as well as the old expressions. For example, code which has had its order reversed or has been negated during the canonicalization process (usually by the ordering functions) needs to be undone by hand, so that the new expression can be reversed/negated appropriately too. This is accomplished by adding metadata to nodes which have been reversed or negated during the transformations, tagging them with the appropriate change type. 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.

The other main category of special reification cases involve the reordering of subtrees within the AST. This happens to boolean operations which 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's easiest to move up in the tree until there are no more ordering differences between the original and new expressions, then create a Change Vector between those nodes. Though this technically makes the edit look larger, it is still encompassing the same idea of what needs to be changed overall.

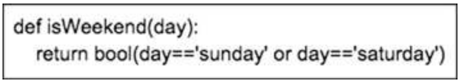
With the global id mapping and these few special undoing functions, state reification can appropriately map the edit back into the student's context. The edit can then be used in hint templates to show the student what they should do next within their problem-solving process.

Full Process

In the previous section, we explained how the path construction and state representation processes are used to make hint generation possible. Here, we demonstrate the full process of how ITAP creates hints with an example from a real student.

Consider the problem isWeekend, which provides a day (a string) and asks the student to determine if the given day is a weekend. In the solution state shown in

[1] Reification: making something real and/or concrete. This is the opposite of abstraction.



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

Fig. 10 The original solution state

Fig. 10, the student has gotten the general structure of the solution right, with one conceptual error; they're using lowercase strings instead of uppercase. This makes their code fail several of the test cases.

Once ITAP determines that the state is incorrect, we run the code through the canonicalization suite in order to reach an abstracted state. This ends up anonymizing, simplifying, and reordering the code, which results in the abstract state shown in Fig. 11.

Next, ITAP uses the path construction algorithm to find a macro-level edit path from the current state to the solution. The algorithm identifies the closest correct state currently within the space; it attempts to simplify the correct state, but both change vectors (shown in Fig. 12) between the two states are required to pass all the test cases.

In generating the path, the algorithm separates the two edits, leading to a two-step macro-level path. The first edit (and the one we'll provide a hint on) is shown in Fig. 13.

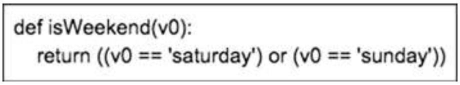
Next, ITAP uses state reification to reverse the abstraction process. For this change vector, there are no special functions that need to be undone; the system only needs to use the global ID of the old expression in the edit (the string 'saturday') to locate it in the original code's AST. This gives us a new edit (shown in Fig. 14) which points to the correct location, in the right value of the comparison operation instead of the left.

At this point, ITAP tokenizes the new expression in the edit, to reduce the amount of information given to the user. But for this example, the new expression is only one token to start with, so it isn't changed at all. Now, with a fully prepared edit, the system can fill out a hint message template using the change vector's data.

Currently, ITAP provides two levels of hints: a 'point' hint and a bottom-out hint. The first only tells the student where the change should be made and what type of change it is (in this case, that the string 'saturday' needs to be replaced), while the second provides all of the information needed to make the edit. The template for the bottom-out hint is:

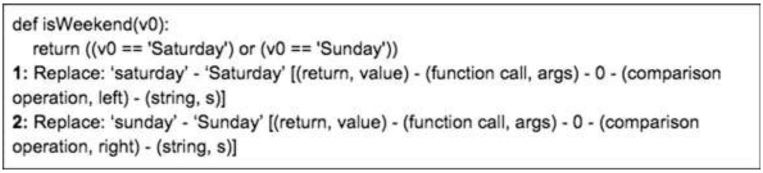
[Location info] + [action verb 1] + [old val] + [action verb 2] + [new val] + [context].

An example showing how this template is filled out is shown in Fig. 15. *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. Finally, the *old and new values* come from the old and new expressions, to show the actual content



```
def isWeekend(v0):
    return ((v0 == 'saturday') or (v0 == 'sunday'))
```

Fig. 11 The abstract state of the incorrect code. Variables have been anonymized and expressions reordered, and the cast to bool has been removed



```
def isWeekend(v0):
   return ((v0 == 'Saturday') or (v0 == 'Sunday'))
1: Replace: 'saturday' - 'Saturday' [(return, value) - (function call, args) - 0 - (comparison
operation, left) - (string, s)]
2: Replace: 'sunday' - 'Sunday' [(return, value) - (function call, args) - 0 - (comparison
operation, right) - (string, s)]
```

Fig. 12 The goal state for this code, and the two change vectors that can be applied to the incorrect code to get there

that needs to be changed. The 'point' hint template simply replaces the *new value* with a filler expression, so that the student can try to figure it out on their own.

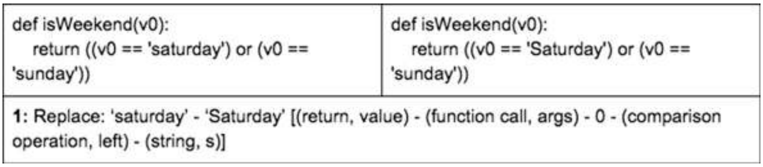
Evaluation

In this section we provide a technical evaluation of ITAP, with separate tests for the path construction algorithm, the state representation effectiveness, and the system's potential for self-improvement in its ability to provide personalized hints. We define a personalized hint as a hint that is adaptive to the students' individual strategy as revealed by their work so far. In operational terms, a hint is more personalized if it guides a student towards a nearer correct solution (i.e., one that is consistent with a student's strategy) than a more distant one (i.e., one that uses a different strategy).

Methodology

The data set we use was collected from a set of 15 students enrolled in an introductory programming course in January 2015. These students were completing code-writing practice problems in an online environment. They were able to select problems from 15 code-writing problems included in this environment (details shown in Table 2). Overall, the students generated 585 program states, where states were created whenever a student compiled their code, excluding adjacent states with identical code. 294 (50.3 %) of these states were syntactically correct but semantically incorrect, thus providing substantial data to evaluate ITAP on the target code state type.

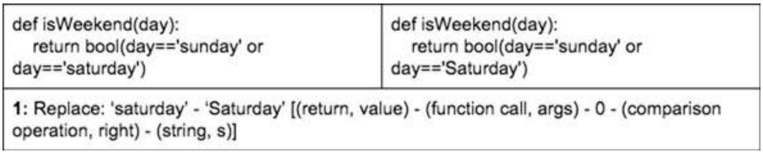
This data set contains programs that only use fairly simple code; students mainly used boolean and comparison operations, as well as occasional conditional statements. We have demonstrated in past work that the components of our system can function on more complex problems that include loops and multiple functions (Rivers and Koedinger 2012, 2014). For this paper, we chose to use this simpler data set because it contained a greater proportion of intermediate student work states that were incorrect or incomplete.



def isWeekend(v0):
return ((v0 == 'saturday') or (v0 ==
'sunday'))
	def isWeekend(v0):
	return ((v0 == 'Saturday') or (v0 ==
	'sunday'))

```
1: Replace: 'saturday' - 'Saturday' [(return, value) - (function call, args) - 0 - (comparison
operation, left) - (string, s)]
```

Fig. 13 The edit that will be hinted first



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

1: Replace: 'saturday' - 'Saturday' [(return, value) - (function call, args) - 0 - (comparison operation, right) - (string, s)]	

Fig. 14 The edit after state reification, located using the global ID of the old expression

As can be seen in Table 2, the normalization functions reduced the number of unique states in most of the problems, both for incorrect and correct states. However, this effect is much more pronounced among the correct states, with a 60 % reduction rate in comparison to the average incorrect state reduction rate of 26 %. In general, there is more semantic variation in the mistakes that students make than there is in their correct solutions. In other words, there is a more limited set of semantically distinguishable correct strategies than there are semantically distinguishable incorrect solution attempts.

Evaluating Path Construction

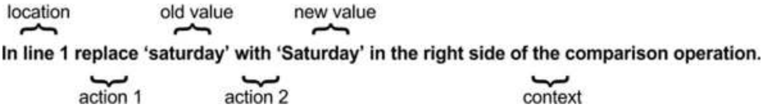
To evaluate the path construction algorithm, we consider two main questions. First: how often can the algorithm successfully generate a chain of hints that reaches a correct state? And second: how long are these hint chains, on average?

Path Existence

We can test how often ITAP is able to generate a chain of hints that, when applied in order, lead from the given state to a correct state. Each edit in the chain needs to be generated anew after the one before has been applied to simulate how well students at any point in the chain would be directed if they followed each hint, or followed the expected path of edits on their own.

In the basic version of this test, we ran the code states from the dataset in the order that they were originally submitted by students, in an attempt to replicate a realistic learning situation. For 98.3 % of the solution states (289/294), ITAP was able to generate a full token-edit hint chain to a correct final solution. All five of the chains which could not generate a full path to a correct solution (usually due to loops occurring in the solution space, which we will discuss later) came from one problem, onlyTwoTrue; for that problem, the success rate was 89.8 % (44/49). All other problems had a 100 % success rate.

In the advanced version of the test, we look at different orderings of the states within problems to determine the robustness of the system. The states for each problem were fully randomized in order, with student traces split up into individual states. We ran ITAP 20 times with randomized-order data sets, clearing the solution space in between each iteration. 99.3 % (5840/5880) of the states were able to generate complete paths to solution. Again, almost all of the broken paths came from a single problem,



In line 1 replace 'saturday' with 'Saturday' in the right side of the comparison operation.

Fig. 15 The resulting hint for our example program

Table 2 The distribution across our 15-problem data set of syntactically incorrect, semantically incorrect, and correct programs. For programs that are syntactically correct, we also include the percent reduction in states due to canonical abstraction, where reduction is measured as (original number - canonical number)/original number (e.g., 50 original states going to 40 canonical states is a 20 % reduction)

Problem	% of all states			% Canonical reduction		Total # states
	Incorrect syntax	Incorrect semantics	Correct	Incorrect semantics	Correct	All
avocados	2 %	74 %	23 %	40 %	82 %	47
canDrinkAlcohol	14 %	46 %	39 %	38 %	64 %	28
firstHalf	8 %	75 %	18 %	10 %	44 %	51
hasTwoDigits	0 %	58 %	42 %	20 %	27 %	26
isCandyShopOpen	11 %	58 %	31 %	24 %	64 %	36
isLowerCase	9 %	56 %	35 %	16 %	58 %	34
isNegative	9 %	23 %	68 %	20 %	93 %	22
isNotConnected	28 %	55 %	17 %	24 %	67 %	69
isWeekend	31 %	48 %	21 %	25 %	64 %	52
onlyOneTrue	21 %	21 %	58 %	0 %	18 %	19
onlyTwoTrue	27 %	53 %	20 %	35 %	10 %	49
overNineThousand	25 %	36 %	39 %	15 %	93 %	36
teenagedNotPrime	42 %	38 %	20 %	38 %	45 %	55
willPass	50 %	12 %	38 %	0 %	85 %	34
withinFive	4 %	63 %	33 %	41 %	44 %	27
Total	21 %	50 %	29 %	26 %	60 %	585

onlyTwoTrue; 35 states here led to broken chains, leading to a success rate of 93.5 % (508/543). The remaining five broken paths came from three different problems, and all problems had success rates greater than 99.3 %.

Path Length

We can begin to evaluate how personalized ITAP's hints are by determining how many edits are required to move from a given incorrect state to a correct state. A comparison of the lengths of the hint chains (counted in token edits) to the number of tokens in the provided teacher solution (which serves as an approximate measure for the complexity of the problem) can demonstrate how effective the algorithm is at providing a short path to a correct state.

In this test, we use the randomized-order method described above to observe the behavior of ITAP on the same states in different circumstances. On average across all problems, ITAP took 5.62 token edits to arrive at a correct state (with a standard deviation of 1.16 on average). As a point of comparison, a system that only used a single solution to provide hints would produce, for a submitted empty solution, a number of edits equal to the token length of that original solution, which is 8.27 on average. Such a system would provide fewer hints for a submitted partial solution that was consistent (but incomplete) with this original solution, but potentially many more for a partial solution following a different strategy, as deletion edits would be needed. Therefore, an average hint chain length of less than 8.27 shows great promise.

There is a strong correlation between the average standard deviation of the chain lengths and the length of the teacher's solution (r=0.86), which seems to imply that more complex problems have more variation in the hint chains generated. There is also a strong correlation between average chain length and function length (r=0.80), which makes sense, as longer functions would require more edits.

Evaluating State Representation

To evaluate the state representation process, we consider the effectiveness of the state abstraction process and the canonicalizing transformations that produce the abstract states. We determine first if the canonical states improve the performance of the hint generation system, and second, which of the transformations contribute the most to any improvement that occurs.

State Space Size Reduction

One of the main goals of using a different representation for code states is to make the solution space more reasonably sized, to deal with all the variation that occurs naturally in student code. To test how effective this reduction is, we can determine how much the size of the solution space is reduced from its original size, in terms of distinct concrete states, to the reduced size, in terms of abstract states. When we examine states that have just had code 'cleaned' (by standardizing whitespace and comments), without transformations being applied, we find a 19.3 % reduction total; when transformations are included as well, there's a 39.6 % reduction over all states. Therefore, about 20 % of the reduction is due solely to the transformation functions.

The amount of reduction does not matter if it doesn't shorten the lengths of the hint paths (to reduce the time students need to spend per problem), so we can also test how many hints are needed in functions when code is represented with and without the normalizing functions. When testing with student states in their provided order, we found the average length of hint chains based on original states was 7.61 while the average length of chains based on canonical states was 6.00. However, for some functions (hasTwoDigits, isWeekend, and onlyOneTrue) the abstract chain length was longer (by 0.8, 0.56, and 0.5 edits, respectively). We discuss below why the abstract chains are sometimes longer and how ITAP could be improved to guarantee shorter paths.

Canonicalization Ablation Testing

To further analyze how the transformations affect the abstraction process, we perform ablation testing to determine which of the individual canonicalizing transformations have the largest effects on state space reduction. In this ablation testing, we compare the performance of the full transformation suite (in terms of canonical reduction) against the performance of 'all-but-one' suites, where each alternative has one of the transformations removed.

The results of this analysis are shown in Table 3. For each transformation, we investigate the percentage effect that the individual transformation had on the total

Table 3 Results of the ablation testing on individual transformation functions. Change size was computed by measuring 1 - (reduction with all-but-one suite) / (reduction with full suite). We do not include six functions (cleanupBoolOps, cleanupRanges, cleanupSlices, cleanupInverts, conditionalRedundancy, and combineConditionals) which had no effect on canonical reduction in this problem set

Transformation	Change in incorrect canonical reduction	Change in correct canonical reduction
deadCodeRemoval	51.16 %	27.27 %
collapseConditionals	25.58 %	43.64 %
cleanupTypes	18.60 %	21.82 %
orderCommutativeOperations	16.28 %	27.27 %
deMorganize	13.95 %	7.27 %
constantFolding	11.63 %	9.09 %
copyPropogation	6.98 %	23.64 %
cleanupNegations	2.33 %	1.82 %
cleanupEquals	0 %	3.64 %
helperFolding	0 %	1.82 %

reduction by computing 1 - (reduction of all-but-one-suite)/(reduction of total suite). This demonstrates how large of an effect the individual transformations have on the reduction as a whole. Our results showed that for this set of problems, the most effective transformations are mainly simplification functions (dead code removal, constant folding, copy propagation), with ordering transformations also contributing (orderCommutativeOperations, deMorganize) and a few domain-specific functions helping as well (collapse conditionals, cleanup types). However, there was a large set of domain-specific functions that had no effect on canonical reduction.

Further investigation will need to be done to see how much the individual functions contribute to space size reduction on a more diverse range of problems, but our results seem to imply that the simplifying and ordering functions will be highly effective on most problems, while the domain-specific transformations will be better suited to individual problems.

Testing for Self-Improvement

Data-driven tutoring has great potential to dynamically improve tutoring performance as students use the tutor and more solution data is collected. To test this potential, we evaluate changes in hint path length for every state of every problem at three different points of solution data accumulation. First, ITAP is run after only seeing the teacher's solution (at the beginning of solution space construction); second, it's run after half of the states have been run through the system (in the middle); and finally, it's run after every other state has been inserted into the solution space (at the end). All of these path lengths can be compared to a gold standard path length, which we define to be the minimum length path seen in the 20-random-iteration test done before. We use this random minimum length as a gold standard because randomizing the path lengths tends to create best possible cases for the solution spaces, resulting in a few low-length hint chains.

Table 4 Results from the solution space size test, with average path lengths at start, middle, and end

Problem	Function	Start	Middle	End	Gold standard
avocados	8	4.94	4.91	4.71	4.66
canDrinkAlcohol	7	3.62	3.54	3.54	3.54
firstHalf	4	2.39	2.42	2.58	2.29
hasTwoDigits	8	6.67	6.47	5.27	5.27
isCandyShopOpen	11	4.71	3.38	2.43	2.38
isLowerCase	8	7.26	6.84	6.74	5.11
isNegative	4	2.80	2.80	2.80	2.80
isNotConnected	8	5.87	5.73	5.45	5.45
isWeekend	8	6.76	7.04	6.76	5.80
onlyOneTrue	10	9.00	6.25	7.50	4.00
onlyTwoTrue	17	17.92	12.50	9.80	7.35
overNineThousand	4	3.46	3.38	3.46	2.85
teenagedNotPrime	11	9.48	8.57	8.43	7.62
willPass	4	3.50	4.00	4.00	3.50
withinFive	12	10.24	5.35	4.18	2.65
Average	8.27	6.57	5.55	5.18	4.35

The results from this evaluation are shown in Table 4. There is improvement in ITAP's performance once more states have been seen, as the average path length goes from 6.57 at the start to 5.55 at the middle. However, there is no large improvement between middle and end (5.55 to 5.18), which is unexpected. Furthermore, there are several problems which seem to get worse in performance the more states they are given (such as willPass and firstHalf).

Further analysis revealed a strong correlation between the length of the original solution and the reduction in the path length between start and end states, where reduction was measured as (1 - end length/start length) (r=0.8182). There is also a good correlation between the start path length and the reduction size (r=0.6477). These correlations imply that personalized hint generation has more room for improvement on more complex problems; with simpler problems, hint generation can achieve high-level performance with only one provided solution.

Discussion

Data-driven hint generation in vast problem spaces is difficult due to the variety of possible paths through the solution space. We have been exploring whether it is possible to rapidly build a viable abstract solution space with enough paths to give students personalized next-step hints that are adaptive to their particular strategy. We are particularly interested in whether abstract solution spaces can be constructed without a large amount of student solution data for each programming problem given, as this could support rapid tutor development. We have made progress toward these ambitious goals by implementing a three step strategy that involves state abstraction (the process of reducing syntactic variability in code states), path construction (determining which steps a student should take to improve their solution), and state reification (re-individualizing the resulting edits into personalized hint messages).

In prior work, we demonstrated that state abstraction can reduce the space by around 50 % for a variety of reasonably complex programming problems (Rivers and Koedinger 2012). We also demonstrated that path construction can identify personalized goal states and generate macro-level edits for any given states (Rivers and Koedinger 2014). In this paper, we introduced the state reification algorithm that is critical for the system to personalize next-step hints which are computed in the abstract space, putting the edits into the concrete structure of the individual student's solution. We also presented an analytic evaluation of the combined system, ITAP, with respect to how well it can provide personalized hints, how much data is needed, and how quickly the system improves as data is added.

Through our evaluations we have demonstrated that ITAP has great potential for fast generation of solution spaces by showing the existence of hint chains even given single-state solution spaces. We demonstrated that ITAP can generate a path to a correct solution from almost any state. We also showed that the paths constructed become shorter and more efficient as more states are added to the solution space, due to better mapping of states to their individualized goal states. This led to more personalized hint chains which provided hints which led to the student needing to remove less of their original code. Together, these findings demonstrate that we have met our goal: we can create a data-driven tutor rapidly by starting with a small amount of data, and then watch as it improves over time as more data is collected.

Limitations

The current results are promising, but the evaluation revealed some limitations that, if addressed, could further improve ITAP. First, while our overall results for self-improvement show that improvement happens as more data is gathered, there are

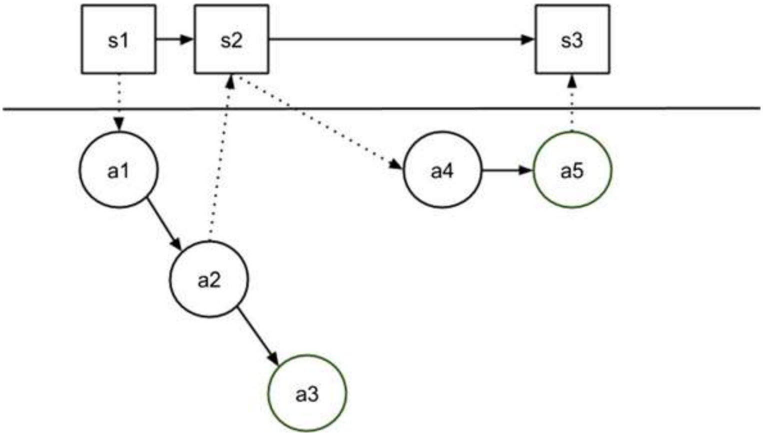
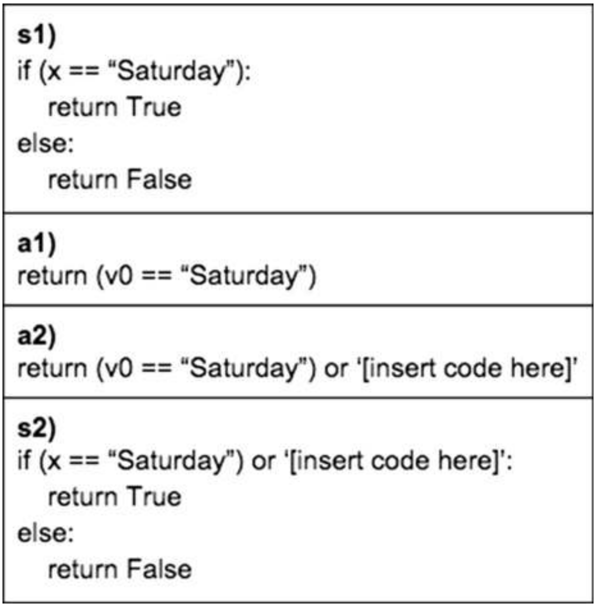


Fig. 16 A case where efficient path construction breaks down. The original state (s1) is mapped into the abstract space as a1, and given a path to follow to reach a3. When a2 is reified back into the original context, it becomes s2, and when s2 is canonicalized, it becomes a4. This results in the goal state changes from a3 to a5, resulting in some edits that are unnecessary

individual functions where hint chains actually grow longer when more data is available. Further exploration of the data has shown that the longer hint chains are primarily caused by state representation problems that arise while doing token-by-token hint chaining. Sometimes, token edits which occur in the abstract space are mapped back to individual states, then re-canonicalized into a different state than they inhabited before (see Fig. 16). This problem generally occurs whenever an edit changes the *type* of an expression, which can invalidate many canonicalizing transformations. As a result, the goal state for the code can change halfway through hint chaining, resulting in some edits which are not necessary for the final goal.

To demonstrate how this can happen, we provide the following example. A student solving the problem isWeekend asks for a hint on the program labeled *s1* in Fig. 17, which uses an unnecessary *if* statement. This program is canonicalized into a simpler version in the abstract solution space (program *a1*). Note that this transformation checks that the condition of the *if* statement, *v0* == "*Saturday*", is of the boolean type (which is *not* necessary in Python). The path construction algorithm identifies that the next step should change to program *a2* (to add another condition using an *or* statement). State reification creates a concrete version of *a2* that is otherwise like the student's original approach in *s1* (using an *if* statement). The result is program *s2*.

When program *s2* in Fig. 17 is canonicalized, the *if* statement can no longer be collapsed, because the test in the *if* statement does not have a guaranteed boolean type (Python *or* expressions return the first 'true-ish' value, which could be True, or could be '[insert code here]'). Therefore, the abstract version of this code statement would be mapped to a different location in the solution space, and could thus be assigned a different goal state.



s1)
```
if (x == "Saturday"):
   return True
else:
   return False
```

a1)
```
return (v0 == "Saturday")
```

a2)
```
return (v0 == "Saturday") or '[insert code here]'
```

s2)
```
if (x == "Saturday") or '[insert code here]':
   return True
else:
   return False
```

Fig. 17 Further demonstration of a potential breakdown in path construction between concrete and abstract spaces. Program s1 is the original concrete code, a1 is the canonicalized version, a2 is the next step in the abstract space, and s2 is the next step in the concrete space

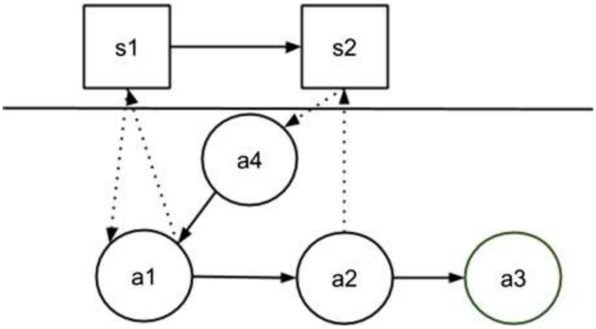


Fig. 18 A case where path construction fails due to an infinite loop between the two spaces, caused by the canonicalization and individualization of states

This breakdown is also the cause of the small number of cases where the system was not able to generate paths from the original state to the goal state, the problem that plagued onlyTwoTrue in our evaluation. Our investigations showed that all of these states were suffering from loops between the individual and abstract state spaces, which generated pairs of edits that kept undoing each other (see Fig. 18). Such looping is rare in the current evaluation (<1 % of states) and seems to only happen when expressions are reorderable in one piece of code but not reorderable in another. This only occurs in highly complex boolean expressions (the kind that commonly appear in solutions to onlyTwoTrue).

One of the possible ways to fix this problem is to run a version of path construction on the concrete states (without any canonicalizing transformations). Paths in the concrete space do not change due to state representation as edits are made. ITAP can than select the shorter of the generated concrete or abstract path as the final path to use. Another solution can involve tracking the previous code states a student has submitted, and refusing to let the path construction algorithm change the goal state that was originally assigned.

One limitation of this evaluation is that it was done with a small set of students and only on relatively simple problems that mostly covered boolean logic and comparisons, with some if statements used as well. It is certainly possible that some of our results may differ when given problems that require more lengthy code, and further evaluation will need to be done to determine if this is the case. However, we have shown in past analyses that the base functionality of canonicalization and path construction work just as well for more complex problems (Rivers and Koedinger 2012, 2014), so there is promise for future analyses.

There are also a few more general limitations of the ITAP system. First, the current system only works for programs which are syntactically correct and parseable, so it cannot help students who are struggling with syntax. We are currently investigating potential extensions to ITAP that could support syntax hints as well. We focused on semantic errors first because they are arguably a bigger barrier for learners – an investigation into a large corpus of student code-writing data has shown that semantic errors are a more serious challenge to novices than syntax errors, since they take much longer to fix (Altadmri and Brown 2015).

Another major limitation of this method is that it relies on the existence of test cases that can measure the correctness of solutions. Though there are a large number of programming problems which can easily be tested using input/output sets, there are many other problems which are difficult to test; for example, graphical assignments, interactive programs, and programs using randomization. It would be possible to use a simplified version of the path construction algorithm to provide hints for these problems based on a handful of instructor-provided correct states, but such a system could not be self-improving (as it could not add new correct solutions on its own). And of course, we would not be able to provide hints for any problem where 'correctness' is an open-ended concept- for example, a task where students use a graphics package to draw their own names. Many creative tasks fall into this category, and supporting students who are working on them is still an open problem.

Finally, and most importantly, we have only provided a technical evaluation for the algorithms in this paper. Though this evaluation demonstrates the potential of ITAP, it cannot measure how students will react to automatically-generated hints, or whether hints will increase or expedite learning. Positive evaluation of the similar Hint Factory approach indicates a high likelihood that ITAP will improve learning (Stamper et al. 2011). Nevertheless, we intend to run an experimental analysis to determine how effective data-driven tutoring is at enhancing learning in real classrooms. We also plan to evaluate the appropriateness of the generated hint messages by contrasting them to messages generated by real TAs in a future study.

Future Work

Throughout this paper, we have described a set of algorithms which can provide hint generation for ITAP, a data-driven tutoring system. Next-step hints are highly valuable in ITS frameworks, but they are not the only required component. In his paper on the structure of tutoring systems, VanLehn (2006) identifies an 'outer loop' and an 'inner loop' which, when put together, create a tutoring system. While our system can handle the inner loop (using test cases for feedback), we do not yet support an outer loop which can create an overall student model and intelligently choose which problems to show to the student. Other researchers have demonstrated that data-driven outer loop creation can be done by automatically extracting knowledge components from programming problems (Hosseini and Brusilovsky 2013) and automatically improving knowledge component models (Koedinger et al. 2012). Such outer loop development could be combined with our inner loop to create a more completely adaptive system.

We are also interested in investigating how state abstraction, path construction, and hint reification can be used in different ways to give students different kinds of feedback. There is great potential in using canonicalizing functions to identify inefficient student code, and path construction could be modified to help students with already-correct solutions improve style, or efficiency, or anything else that the teacher thinks is valuable. We hope to eventually expand on these ideas, to increase the range of types of feedback students can get while learning to program.

In our upcoming work, we plan to run a large-scale randomized control trial to test ITAP with students who are working on practice problems that cover a range of different topics and complexities in order to determine whether having access to hints has beneficial effects on student performance. We plan to use this evaluation to more closely examine how students interact with feedback and hints, which will inform how we design the help systems of the future. We also want to determine whether we can make personalized intelligent tutoring more accessible to teachers by making it easy for them to build tutors quickly using their own problems. If this outreach to teachers succeeds, it may make intelligent tutoring more accessible to teachers, which could result in greater usage of personalized education across education as a whole.

Acknowledgments This work was supported in part by Graduate Training Grant awarded to Carnegie Mellon University by the Department of Education (# R305B090023) and by the Pittsburgh Science of Learning Center, which is supported by the National Science Foundation (#SBE-0836012).

References

Altadmri, A., & Brown, N. C. (2015). 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 522–527).

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP tutor. *Cognitive Science, 13*(4), 467–505.

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).

Carter, J., Dewan, P., & Pichiliani, M. (2015). Towards incremental separation of surmountable and insurmountable programming difficulties. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 241–246).

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 M. G. Helander, T. K. Landauer, & P. V. Prabhu (Eds.), *Handbook of human-computer interaction* (pp. 849–874). Amsterdam: Elsevier Science B. V.

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., 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).

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).

Gerdes, A., Jeuring, J., & Heeren, B. (2012). An interactive functional programming tutor. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education* (pp. 250–255).

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

Hosseini, R., & Brusilovsky, P. (2013). JavaParser: A fine-grain concept indexing tool for java problems. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)* (pp. 60-63).

Kimball, R. (1982). A self-improving tutor for symbolic integration. In D. Sleeman & J.S. Brown (Eds.), *Intelligent tutoring systems (Vol. 1)* (pp. 283-307). Academic Press.

Koedinger, K. R., McLaughlin, E. A., & Stamper, J. C. (2012). Automated Student Model Improvement. In *Proceedings of the 5th International Conference on Educational Data Mining* (pp. 17–24).

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., & Menzel, W. (2007). Using constraint-based modelling to describe the solution space of ill-defined problems in logic programming. In *Proceedings of the 6th international conference on Advances in web based learning* (pp. 367–379).

Le, N. T., Strickroth, S., Gross, S., & Pinkwart, N. (2013). A review of AI-supported tutoring approaches for learning programming. In N.T. Nguyen, T. van Do, & H. A. le Thi (Eds.), *Advanced Computational Methods for Knowledge Engineering* (pp. 267–279). Springer International Publishing.

McLaren, B.M., Koedinger, K.R., Schneider, M., Harrer, A., & Bollen, L. (2004). Bootstrapping novice data: semi-automated tutor authoring using student log files. In *The Proceedings of the Workshop on Analyzing Student-Tutor Interaction Logs to Improve Educational Outcomes, Seventh International Conference on Intelligent Tutoring Systems (ITS-2004).*

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).

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).*

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).

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. (2014). Automating hint generation with solution space path construction. In *Proceedings of the 12th international conference on Intelligent Tutoring Systems* (pp. 329–339).

Shih, B., Koedinger, K. R., & Scheines, R. (2011). A response time model for bottom-out hints as worked examples. In C. Romero, S. Venture, M. Pechenizkiy, & R. S.J.D. Baker (Eds.), *Handbook of Educational Data Mining* (pp. 201–212). CRC Press.

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).

Stamper, J. C., Eagle, M., Barnes, T., & Croy, M. (2011). Experimental evaluation of automatic hint generation for a logic tutor. In *Proceedings of the 15th international conference on Artificial intelligence in education* (pp. 345–352).

VanLehn, K. (2006). The behavior of tutoring systems. *International Journal of Artificial Intelligence in Education, 16*(3), 227–265.

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.