Generating Hints for Programming Problems Using Intermediate Output

Barry Peddycord III
North Carolina State University
890 Oval Drive
Raleigh, NC, 27695
bwpeddyc@ncsu.edu

Andrew Hicks
North Carolina State University
890 Oval Drive
Raleigh, NC, 27695
aghicks3@ncsu.edu

Tiffany Barnes
North Carolina State University
890 Oval Drive
Raleigh, NC, 27695
tmbarnes@ncsu.edu

ABSTRACT

In this work, we compare two representations of student interactions within the context of a simple programming game. We refer to these representations as Worldstates and Codestates. Worldstates, which are representations of the output of the program, are generalizations of Codestates, snapshots of the source code taken when the program is run. Our goal is to incorporate intelligent data-driven feedback into a system, such as generating hints to guide students through problems. Using Worldstates simplifies this task by making it easier to compare student approaches, even for previously unseen problems, without requiring expert analysis. In the context of the educational programming game, BOTS, we find that worldstates require less prior data to generate hints in a majority of cases, without sacrificing quality or interpretability.

Keywords

Hint Generation, Programming Tutor, Educational Game

1. INTRODUCTION

One key benefit of Intelligent Tutoring Systems over other computer-aided instruction is the ability to provide intelligent adaptive feedback. A popular way of providing this feedback is through the generation of hints. Hints can help students who are stuggling by suggesting a next step or providing a clue about what the next step might be. While some of the earliest work in this area focuses on building models of the learner [7], recent work shows that quality hints can be generated in certain domains using data-driven methods, informed by the type and frequency of actions taken by students in the system [1].

Programming has been a domain of interest for tutoring systems as far back as the Lisp Tutor [3]. There has been recent interest in trying to apply hint generation techniques such as Stamper and Barnes' Hint Factory [9] to programming languages [6, 8], but this still remains an open problem given the complexity associated with learning programming languages. One of the challenges associated with handling programming tutors comes from the diversity of possible programs that a student can write.

2. PROBLEM STATEMENT

Our work is an effort to add techniques from Intelligent Tutoring Systems to an educational game called BOTS. BOTS is an educational programming game designed to teach middle school students the principles of programming, and also allows students to create their own puzzles for other students to solve. Due to the rapid creation of new puzzles, it is necessary that hints can be generated with relatively little data, since expert authoring is infeasible.

Like Rivers, our work is based on Hint Factory, but rather than attempting to analyze the student source code, our work looks entirely at the *output* of the programs. We hypthesize that using the output of programs for hint generation allows us to deal with the challenge of source code diversity and generate more hints with less data than using source code alone, without diminishing the quality of hints.

Though the hints have not yet been integrated into the game, this work shows that our technique is promising and could feasibly be integrated into the game for future studies.

3. PRIOR WORK

One popular technique for automatic hint generation that has enjoyed success is Stamper and Barnes' Hint Factory [9]. Hint Factory can be applied in any context where student interactions with a tutor can be defined as a graph of transitions between states [2]. In the Deep Thought propositional logic tutor, students are asked to solve logic proofs by deriving a conclusion from a set of premises [1]. Each time a student applies a logical rule, a snapshot of the current state of the proof is recorded in a graph called the *interaction network* as a node, with an edge following the states along the path they follow to get to their proof. At any point during the proof, a student can ask for a hint, and Hint Factory selects the edge from the state they are currently in that takes them closest to the solution.

The iList Linked List tutor is another example where a Hint Factory-based approach to generating feedback has been successful [4]. In Fossati's work on the iList Linked List

tutor, the developers used a graph representation of the program's state as the input to their system. In order to give hints for semantically equivalent states, rather than for only precisely identical states, the developers searched for isomorphisms between the program state and the previously observed states. This is a non-trivial comparison, and requires significant knowledge of the domain in order to assess the best match when multiple such relations exist.

Rivers et al also use Hint Factory to generate hints specifically for programming tutors. Due to the complexity of programming problems, a simple snapshot of the code will not suffice [8]. There are many varied approaches that can be taken to programming problems, and if a direct comparison is being used, it is rare that a student will independently have a precisely identical solution to another student. Therefore, some way of reducing the size of the state space is needed. During a programming problem, each time a student saves his or her work, their code is put through a process of canonicalization to normalize the naming conventions, whitespace, and other surface-level programming variations that differ widely across students. When two programs have the same canonical form, they are considered the same state in the interaction network. In both cases, the generation of a hint comes from selecting the appropriate edge and then articulating to the student some way of getting from their current state to the state connecting by that edge.

In Jin's work on programming tutors [6] a similar approach is taken. Rather than using canonical forms generated using abstract syntax trees, the authors generate "Linkage Graphs" which define the relationships between variables in the program, then condense those graphs by removing intermediate variables, creating abstract variables that represent multiple approaches. To take a very simple example, if the goal is to multiply A by B and somehow output the result, a programmer may write `A = A * B` or `C = A * B`. After building the CLG, these approaches would be the same. If the condensed linkage graphs (CLGs) derived from two different programs are isomorphic to eachother, then those programs are said to be similar. Additional analysis is then needed to identify which concepts the abstracted variables in the CLGs represent; in this case, k-means clustering was used to find the most similar abstracted variable, then choosing the most common naming convention for that variable.

We propose that another way of canonicalizing student code is to use the intermediate output of programs. In a study of 30,000 student source code submissions to an assignment in the Stanford Massive Open Online Course on Machine Learning, it was found that there were only 200 unique outputs to the instructor test cases [5]. In the Machine Learning class, there was only one correct output, but despite there being an infinite number of ways to get the problem wrong, Huang et al observed a long tail effect where most students who got the exercises wrong had the same kinds of errors.

4. CONTEXT

4.1 BOTS

BOTS is an educational programming game that teaches the basic concepts of programming through block-moving puzzles. Each problem in BOTS, called a "puzzle", contains buttons, boxes, and a robot. The goal for the player is to write a program for the robot, using it to move boxes so that all of the buttons are held down. The BOTS programming language is quite simple, having elementary robot motion commands ("forward", "turn", "pickup/ putdown") and basic programming constructs (loops, functions, variables). Most programs are fairly small, and students who solve puzzles using the fewest number of blocks overall (fewer "lines of code") are shown on the leaderboard.

One important element of BOTS is that there is a built-in level editor for students to develop their own puzzles. There is a tutorial sequence that contains puzzles only authored by the developers of the system, but students are also free to attempt peer-authored puzzles which may only be played by a few students. Due to the constant addition of new puzzles, expert generation of hints is infeasible, and due to the limited number of times levels will be played, it must be possible to generate hints with very little data.

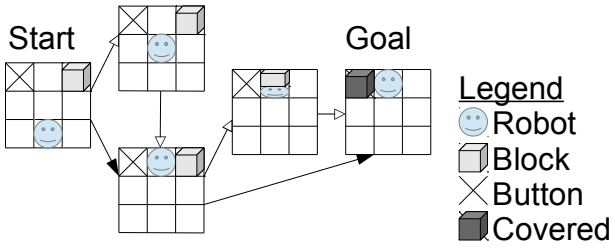

Figure 1: An example of the interaction network of a BOTS puzzle solved two ways by two students. One student (empty arrow) wrote four programs to arrive at the solution, and the other (filled arrow) only wrote two.

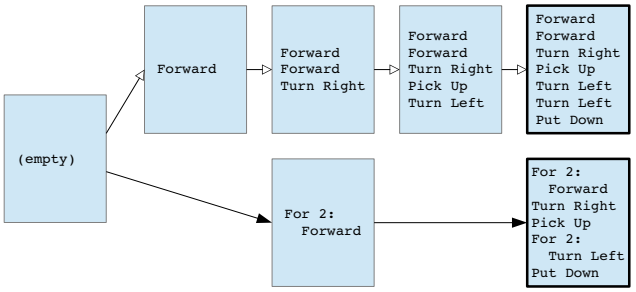

Figure 2: This figure shows the same two students with the same solutions. Notice how in this example, these two paths that get to the same goal have no overlapping states - including the goal state.

We use a data structure called an *interaction network* to represent the student interactions with the game [2]. An interaction network is a graph where the nodes represent the states that a student is in, and the edges represent transitions between states. In this research, we compare the effects of using two different types of states in the interaction network, which we refer to as *codestates* and *worldstates*.

Codestates are snapshots of the source code of a student's program, ignoring user generated elements like function and variable names. Worldstates, inspired by Jin and Rivers, use the configuration of the entities in the puzzle as the definition of the state. In other words, codestates represent student source code while worldstates represent the output of student source code. In both cases, student interactions can be encoded as an interaction network, which enables the application of techniques like Hint Factory. Figures 1 and 2 demonstrate how the interaction networks differ between codestates and worldstates.

4.2 Hint Generation

When generating a hint, the goal is to help a student who is currently in one state transition to another state that is closer to the goal, without necessarily giving them the answer. In this work, rather than attempting to identify these states *a priori* using expert knowledge, we instead use the solutions from other students to estimate the distance to a solution. It is a fair assumption that students who solve the same problem will use similar approaches to solve it [1]. This assumption implies that there will be a small number of states that many students visit when they solve a problem. If several students take the same path from one state to another, it is a candidate for a hint. Hint Factory formalizes this process with an algorithm.

"Generating" a hint is a two-part problem. First, we must devise a *hint policy* to select one of the edges from the user's current state in the interaction network. Then, we must *articulate* the resultant state into a student-readable hint. For example, if we applied hint factory to Figure 1 when the student is in the start state, the edge selected would be the one going down to the bottom-most worldstate. The "hint" might be articulated to the student as "write a program that moves the robot north by two spaces". An example is provided in Figure 3

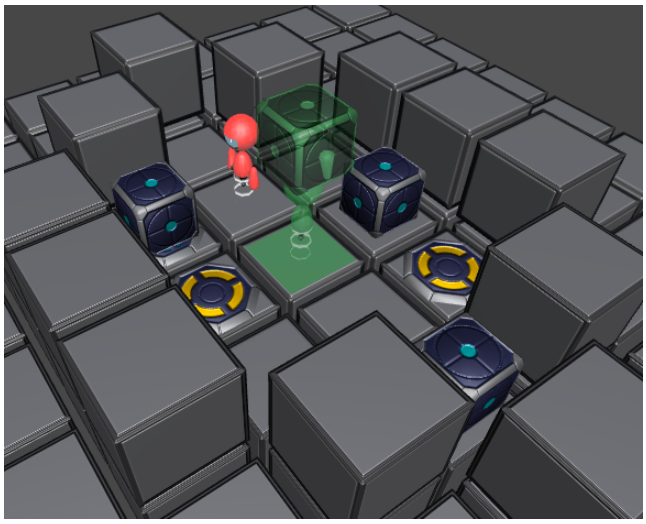

Figure 3: A mock-up of how a high-level hint might be presented in BOTS. The green "hologram" of the robot indicates the next worldstate the player should attempt to recreate.

In order to give a student a relevant hint, another student must have reached the solution from the same state that the one requesting a hint is currently in. If no other student has ever been in that state before, we can not generate an exact hint. Using source code to determine the state is challenging, since students can write the same program in many different ways, which makes the number of states across all students highly sparse, reducing the chance that a match (and by extension, a hint) will be available. Previous work attempts to find ways to canonicalize student program [8], but even then the variability is still very high. Using worldstates, however, serves to "canonicalize" student submissions without having to do complicated source code analysis.

When it comes to the articulation of hints, we can use whatever information is available in the interaction network to provide a hint to the student. Using codestates, a diff between the source code of the current state and the hint state can be used. For worldstates, an animation of the path of the robot can be played to show what the hint state's configuration would look like. In tutoring systems like Deep Thought, hints are given at multiple levels, with the first hint being very high-level, and the last essentially spelling out the answer to the student (a "bottom-out" hint) [1]. In BOTS, we can progressively reveal more information as necessary - hints about the worldstate help a student see how to solve the puzzle without telling them exactly how to code it. If a student is having trouble with the code itself, then lower-level hints might suggest which kinds of operations to use or show them snippets of code that other students used to solve the puzzle.

It is important to note that while worldstates are a generalization, we do not necessarily lose any information when using them for hint generation. BOTS programs are deterministic, and there is no situation where the same code in the same puzzle produces a different output. Therefore, using worldstates not only allows us to articulate high-level hints, it also provides a fallback when a student's source code snapshot is not yet in the interaction network.

5. METHODS

5.1 Data set

The data for this study comes from the 16-level tutorial sequence of BOTS. These levels are divided into three categories: demos, tutorials, and challenges. Demos are puzzles where the solution is pre-coded for the student. Tutorials are puzzles where instructions are provided to solve the puzzle, but the students build the program themselves. Finally, challenges require the student to solve a puzzle without any assistance. Challenge levels are interspersed throughout the tutorial sequence, and students must complete the tutorial puzzles in order - they can not skip puzzles.

For the purposes of this evaluation, we exclude demos from our results, and run our analysis on the remaining 8 tutorials and 5 challenges. The data comes from a total of 125 students, coming from technology-related camps for middle school students as well as an introductory CS course for non-majors. Not all students complete the entire tutorial sequence, as only 34 of the students attempted to solve the final challenge in the sequence. A total of 2917 unique code submissions are collected over the 13 puzzles, though it is important to note that the number of submissions spike

when students are presented with the first challenge puzzle. This information is summarized for each puzzle in Table 1.

Table 1: A breakdown of the tutorial puzzles in BOTS, listed in the order that students have to complete them. Hint states are states that are an ancestor of a goal state in the interaction network, and are the states from which hints can be generated.

		Codestates		Worldstatates	
Name	#Students	Hint	All	Hint	All
Tutorial 1	125	89	162	22	25
Tutorial 2	118	36	50	12	14
Tutorial 3	117	130	210	22	24
Tutorial 4	114	137	225	33	41
Tutorial 5	109	75	106	25	29
Challenge 1	107	348	560	143	191
Challenge 2	98	201	431	86	133
Tutorial 6	90	107	143	33	36
Challenge 3	89	192	278	28	30
Challenge 4	86	137	208	40	45
Tutorial 7	76	206	383	43	57
Tutorial 8	68	112	134	29	30
Challenge 5	34	17	27	13	17

In order to demonstrate the effectiveness of using worldstates to generate hints, we apply a technique similar to the one used to evaluate the "cold start" problem used in Barnes and Stamper's work with Deep Thought [1]. The cold start problem is an attempt to model the situation when a new problem is used in a tutor with no historical data from which to draw hints. The evaluation method described in Barnes and Stamper's previous work uses existing data to simulate the process of students using the tutor, and provides an estimate as to how much data is necessary before hints can be generated reliably. As such, it is an appropriate method for determining how much earlier - if at all - worldstates generate hints as opposed to codestates.

We break the student data for each puzzle into a training and a validation set, and iteratively train the Hint Factory by adding one student at a time. We chart how the number of hints available to the students in the validation set grows as a function of how much data is in the interaction network, and average the results over 1000 folds to avoid ordering effects. The specific algorithm is as follows:

Step 1 Let the Validation set = 10 random students, and the training set = the n-10 remaining students

Step 2 Randomly select a single student attempt from the training set

Step 3 Add states from the student to the interaction network and recalculate the Hint Factory MDP

Step 4 Determine the number of hints that can be generated for the validation set

Step 5 While the training set is not empty, repeat from step 2

Step 6 Repeat from step 1 for 1000 folds and average the results

This approach simulates the a cohort of students asking for hints at the same states as a function of how much data is already in the system, and provides a rough estimate as to how many students need to solve a puzzle before hints can be generated reliably. However, this approach is still highly vulnerable to ordering effects, so to verify a hypothesis that n students are sufficient to generate hints reliably, we do cross validation with training sets of n students to further establish confidence in the hint generation.

6. RESULTS

6.1 State-space reduction

At a glance, Table 1 shows how using worldstates as opposed to codestates reduces the state space in the interaction network. Challenge 1, for example, has 560 unique code submissions across the 107 students who attempted the puzzle. These 560 code submissions can be generalized to 191 unique outputs.

We see a significant reduction in the number of states containing only a single student. Intuitively, students who get correct answers will overlap more in terms of their solutions, while the infinite numbers of ways to get things wrong will result in several code submissions that only a single student will ever encounter. In the Table 2, we look at the number of frequency-one states that students encounter for challenge puzzles 1 through 4.

When using worldstates, in all three cases, more than half of the states in the interaction network are only observed one time. In particular, Challenge 3 is particularly interesting, considering that out of 278 unique code submissions from 89 students, only 8 of the 30 worldstates are observed more than once. The sheer degree of overlap demonstrates that worldstates in particular meet the assumptions necessary to apply hint factory.

Table 2: This table highlights the number of code and worldstates that are only ever visited one time over the course of a problem solution.

		Codestates		Worldstatates	
Name	#Students	All	Freq1	All	Freq1
Challenge 1	107	560	146	191	112
Challenge 2	98	431	127	133	84
Challenge 3	89	278	91	30	22
Challenge 4	86	208	65	45	36

6.2 Cold Start

The graphs in Figure 4 show the result of the cold start evaluation for all 13 of the non-demo puzzles. Looking at the two graphs side-by-side, the worldstates have more area under the curves, demonstrating the ability to generate hints earlier than codestates. The effect is particularly pronounced when looking at the challenge puzzles (represented with the solid blue line). In tutorials, code snippets are given to the students, so there is more uniformity in the code being written. When the guidance is taken away, the code becomes more variable, and this is where the worldstates show a demonstrable improvement.

It is important to note that because of the way worldstates are defined, the ability to generate more hints is trivially

guaranteed. The contribution is the *scale* at which the new hints are generated. For the same amount of student data in the interaction network, the percentage of hints available is anywhere from two to four times larger when using worldstates than codestates.

Figure 5 summarizes these results by averaging the hints available for the first four challenge puzzles. Challenge puzzles are chosen as they represent a puzzle being solved without any prompts on how to solve it, and would be the environment where these automatically generated hints are deployed in practice. Challenge 5 was only attempted by 34 students, so it was left out of the average.

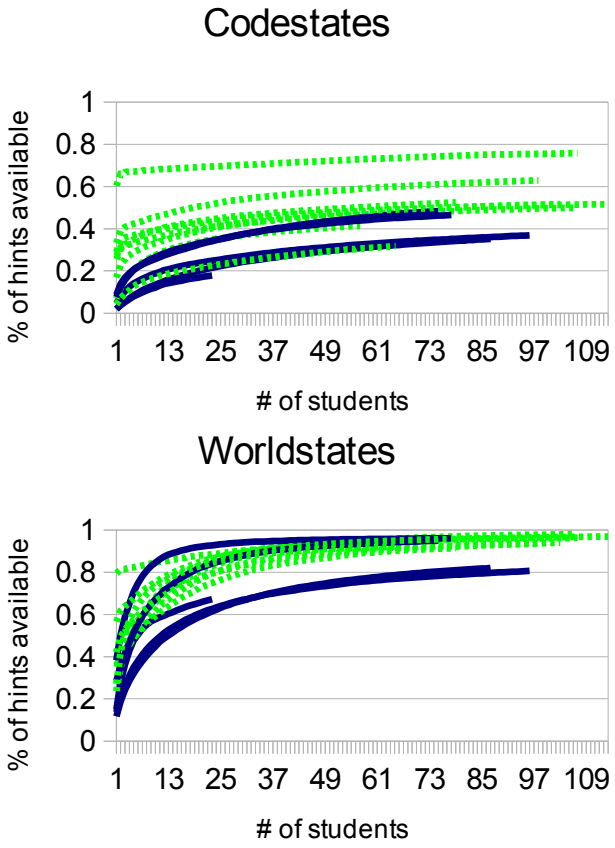

Figure 4: These graphs show the overall performance of hint generation for codestates and worldstates. The solid blue lines are the challenge puzzles, and the dotted light-green lines are the tutorial puzzles.

6.3 Validation

Table 3: This table shows the percentage of hints available using worldstates when 30 students worth of data are in the interaction network.

Name	Average	Median	Min	Max
Challenge 1	0.67	0.66	0.48	0.81
Challenge 2	0.67	0.66	0.44	0.84
Challenge 3	0.94	0.93	0.83	0.99
Challenge 4	0.88	0.87	0.69	0.96

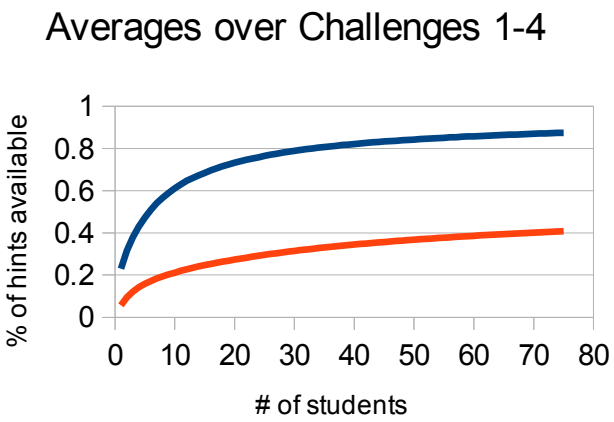

Figure 5: This graph summarizes Figure 4 by averaging the results of the analysis over the first four challenge puzzles. For these challenge levels (where students do not have guidance from the system) we are able to consistently generate hints with less ata when using world states rather than code states.

Figure 5, suggests that 30 students of data should be able to generate hints using worldstates about 80% of the time. To validate this hypothesis, we do cross-validation with training sets of 30 students and validate on the remaining students for challenge puzzles one through four, once again, because they are an appropriate model for how hints would be deployed in practice. We find the average, median, maximum, and minimum of the results over another 1000 trials, and summarize them in Table 3. We find that for challenges 1 and 2, hint generation is below the 80% mark, but for challenges 3 and 4, it is well over.

7. DISCUSSION

Our results indicate that when using worldstates we are able to generate hints using less data than when using code states. The reduction in the state space and the number of hints available after only a few dozen students solve the puzzle is highly encouraging. We only test our results on instructor-authored puzzles for this study, but these results potentially make hint generation for user-generated levels feasible as well.

While on average, the number of hints available using worldstates is very high, it is interesting to look at numbers on a per level basis. For example, in the summary in Table 3, there are less hints available after 30 students have been through the puzzle than the second two, which are presumably harder. This could be an averaging effect due to the more advanced students remaining in the more advanced levels, but there are some structural differences to the levels that may also have an effect. Figures 6 and 7 show the puzzles for Challenges 1 and 4.

Challenge 1 is a much smaller puzzle, but can be solved many different ways. Any of the blocks can be put on any of the buttons, and the robot can also hold down a button, meaning that there are several goal states, especially considering that the superflous block can be located anywhere. Challenge 4 is substantially more difficult, but has

much less variation in how it can be completed. In this puzzle, the student has to place blocks so that the robot can scale the tower, and because of the way the tower is shaped, there are not nearly as many different ways to approach the problem as there are in Challenge 1.

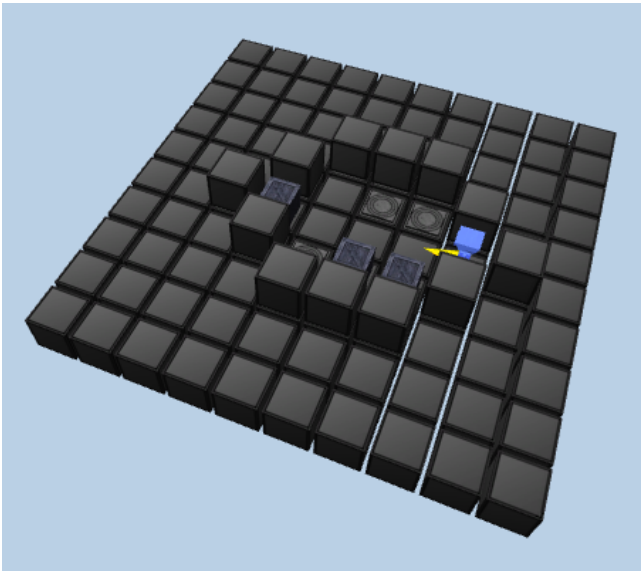

Figure 6: A screenshot of challenge 1. There are more blocks than necessary to press all the buttons, since the robot can press a button too.

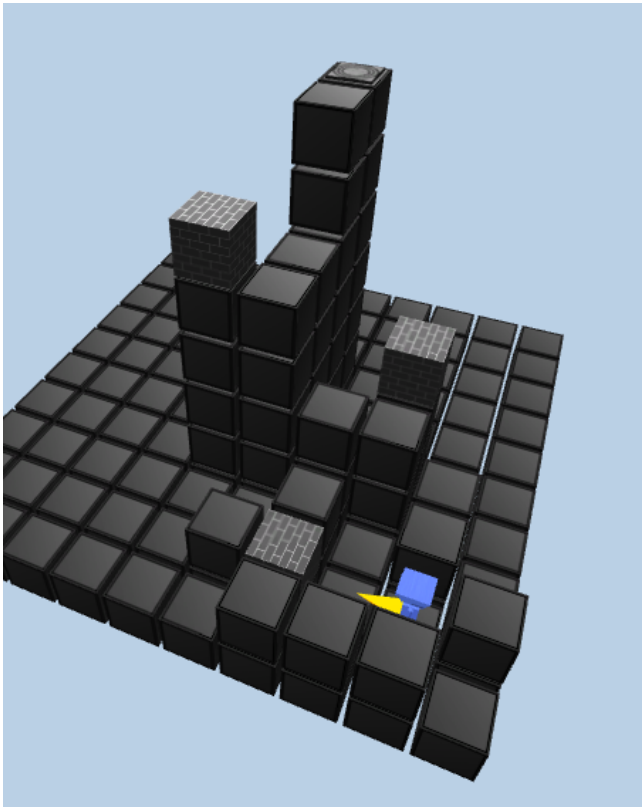

Figure 7: A screenshot of challenge 4. The puzzle is more complex, yet has a more linear solution.

The results for the earlier stages are more interesting for interpreting these results, because it shows how well world-states manage the variability even in the wake of open-ended problems. While we emphasize the ability for worldstates to generate hints, it is important to note that codestates still have utility. A hint can be generated from any of the data in an interaction network, and using a world-based state representation does not restrict us from comparing other collected data. When enough data is collected for the codestates to match, more specific, low-level hints can be generated as well, meaning that more data does not just mean more hints, but also more detailed hints that can be applied at the source code level.

8. CONCLUSIONS AND FUTURE WORK

In this work, we describe our how we added intelligent tutoring system techniques to an educational game. Rather than using knowledge engineering, we instead use the approach used in Deep Thought, a logic tutor built around Hint Factory, where we provide hints to students by drawing from what worked for other students [1]. In order to deal with the fact that student code submissions can be highly diverse, with many different inputs resulting in the same output, we use the output of the student code to represent a student's position in the problem solving process. In doing so, we generate hints much more quickly than if we had only analyzed the source code alone.

In future work, we will identify the ability to generate hints on student-authored puzzles and test the effectiveness of these hints implemented in actual gameplay. We predict that by including hints, we can improve the completion rate of the tutorial and - if our results transfer to student-authored puzzles - improve performance on puzzles generated by other students. We will also explore how well these techniques transfer to contexts beyond our programming game.

9. ACKNOWLEDGEMENTS

Thanks to the additional developers who have worked on this project or helped with our outreach activities so far including Aaron Quidley, Veronica Catete, Trevor Brennan, Irena Rindos, Vincent Bugica, Victoria Cooper, Dustin Culler, Shaun Pickford, Antoine Campbell, and Javier Olaya. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 0900860 and Grant No. 1252376.

10. REFERENCES

[1] T. Barnes and J. C. Stamper. Automatic hint generation for logic proof tutoring using historical data. *Educational Technology & Society*, 13(1):3–12, 2010.

[2] M. Eagle, M. Johnson, and T. Barnes. Interaction networks: Generating high level hints based on network community clusterings. In *EDM*, pages 164–167, 2012.

[3] R. G. Farrell, J. R. Anderson, and B. J. Reiser. An interactive computer-based tutor for lisp. In *AAAI*, pages 106–109, 1984.

[4] D. Fossati, B. Di Eugenio, S. Ohlsson, C. W. Brown, L. Chen, and D. G. Cosejo. I learn from you, you learn

from me: How to make ilist learn from students. In *AIED*, pages 491–498, 2009.

[5] J. Huang, C. Piech, A. Nguyen, and L. Guibas. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*, page 25, 2013.

[6] W. Jin, T. Barnes, J. Stamper, M. J. Eagle, M. W. Johnson, and L. Lehmann. Program representation for automatic hint generation for a data-driven novice programming tutor. In *Intelligent Tutoring Systems*, pages 304–309. Springer, 2012.

[7] B. J. Reiser, J. R. Anderson, and R. G. Farrell. Dynamic student modelling in an intelligent tutor for lisp programming. In *IJCAI*, pages 8–14, 1985.

[8] K. Rivers and K. R. Koedinger. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, page 50, 2013.

[9] J. Stamper, T. Barnes, L. Lehmann, and M. Croy. The hint factory: Automatic generation of contextualized help for existing computer aided instruction. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems Young Researchers Track*, pages 71–78, 2008.