BOTS: Selecting Next-Steps from Player Traces in a Puzzle Game

Drew Hicks
North Carolina State University
911 Oval Drive
Raleigh, NC 27606
aghicks3@ncsu.edu

Yihuan Dong
North Carolina State University
911 Oval Drive
Raleigh, NC 27606
ydong2@ncsu.edu

Rui Zhi
North Carolina State University
911 Oval Drive
Raleigh, NC 27606
rzhi@ncsu.edu

Veronica Cateté
North Carolina State University
911 Oval Drive
Raleigh, NC 27606
vmcatete@ncsu.edu

Tiffany Barnes
North Carolina State University
911 Oval Drive
Raleigh, NC 27606
tmbarnes@ncsu.edu

ABSTRACT

In the field of Intelligent Tutoring Systems, data-driven methods for providing hints and feedback are becoming increasingly popular. One such method, Hint Factory, builds an interaction network out of observed player traces. This data structure is used to select the most appropriate next step from any previously observed state, which can then be used to provide guidance to future players. However, this method has previously been employed in systems in which each action a player may take requires roughly similar effort; that is, the "step cost" is constant no matter what action is taken. We hope to apply similar methods to an interaction network built from player traces in our game, BOTS; However, each edge can represent a varied amount of effort on the part of the student. Therefore, a different hint selection policy may be needed. In this paper, we discuss the problems with our current hint policy, assuming all edges are the same cost. Then, we discuss potential alternative hint selection policies we have considered.

Keywords

Hint Generation, Serious Games, Data Mining

1. INTRODUCTION

Data-driven methods for providing hints and feedback are becoming increasingly popular, and are especially useful for environments with user- or procedurally-generated content. One such method, Hint Factory, builds an interaction network out of observed player traces. An Interaction Network is a complex network of student-tutor interactions, used to model student behavior in tutors, and provide insight into problem-solving strategies and misconceptions. This data structure can be used to provide hints, by treating the Interaction Network similarly to a Markov Decision Process and selecting the most appropriate next step from the requesting user's current state. This method has successfully been employed in systems in which each action a player may take is of similar cost; for example in the Deep Thought logic tutor each action is an application of a particular axiom. Applying this method to an environment where actions are of different costs, or outcomes are of varying value will require some adaptations to be made. In this work, we discuss how we will apply Hint Factory methods to an interaction network built from player traces in a puzzle game, BOTS. In BOTS, each "Action" is the set of changes made to the program between each run. Therefore, using the current hint selection policy would result in very high-level hints comprising a great number of changes to the student's program. Since this is undesirable, a different hint selection policy may be needed.

2. DATA-DRIVEN HINTS AND FEEDBACK

In the ITS community, several methods have been proposed for generating hints/feedback from previous observations of users' solutions or behavior. Rivers et al propose a data-driven method to generate hints automatically for novice programmers based on Hint Factory[8]. They present a domain-independent algorithm, which automates hint generation. Their method relies on solution space, which utilizes graph to represent the solution states. In solution space, each node represents a candidate solution and each edge represents the action used to transfer from one state to another. Due to the existence of multiple ways to solve a programming problem, the size of the solution space is huge and thus it is impractical to use. A Canonicalizing model is used to reduce the size of the solution space. All states are transformed to canonicalized abstract syntax trees (ASTs). If the canonical form of two different states are identical, they can be combined together. After simplifying the solution space, hint generation is implemented. If the current state is in-

correct and not in the solution space, the path construction algorithm will find an optimal goal state in the solution space which is closest to current state. This algorithm uses change vectors to denote the change between current state and goal state. Once a better goal state is found during enumerating all possible changes, it returns the current combination of change vectors. Each change vector can be applied to current state and then form an intermediate state. The intermediate states are measured by desirability score, which represents the value of the state. And then the path construction algorithm generates optimal next states based on the rank of the desirability scores of all the intermediate states. Thus a new path can be formed and added to the solution space, and appropriate hints can be generated.

Jin et al propose linkage graph to generate hints for programming courses[4]. Linkage graph uses nodes to represent program statements and direct edges to indicate ordered dependencies of those statements. Jin's approach applies matrix to store linkage graph for computation. To generate linkage matrix, first, they normalize variables in programs by using instructor-provided variable specification file. After variable normalization, they sort the statement with 3 steps: (i) preprocessing, which breaks a single declaration for multiple variables (e.g. int a, b, c) into multiple declaration statements (e.g. int a; int b; int c;); (ii) creating statement sets according to variable dependencies, which put independent statements into first set, put statements depend only on statements in the first set into second set, put statements depends only on statements in the first and second set into third set, and so on; (iii) in-set statement sorting, during which the statements are sorted in decreasing order within set using their variable signatures. In hint generation, they first generate linkage graphs with a set of correct solutions, as the sources for hint generation. They also compose the intermediate steps during program development into a large linkage graph, and assign a reward value to each state and the correct solution. Then, they apply value iteration to create a Markov Decision Process (MDP). When a student requires hint, tutor will generate a linkage graph for the partial program and try to find the closest match in MDP. If a match is found in MDP, the tutor would generate hint with the next best state based on highest assigned value. If a match is not found in current MDP, which means the student is taking a different approach from existing correct solutions, the tutor will try to modify those correct solutions to fit student's program and then provide hints.

Hint Factory[9] is an automatic hint generation technique which uses Markov decision processes (MDPs) to generate contextualized hints from past student data. It mainly consists of two parts - Markov Decision Process (MDP) generator and hint provider. The MDP generator runs a process to generate MDP values for all states seen in previous students' solutions. In this process, all the students' solutions are combined together to form a single graph. Each node of the graph represents a state, and each edge represents an action one student takes to transform from current state to another state. Once the graph is built, the MDP generator uses Bellman backup to assign values for all nodes. After updating all values, a hint file is generated. The hint provider uses hint file to provide hint. When a student asks for a hint at a existing state, hint provider will retrieve current state

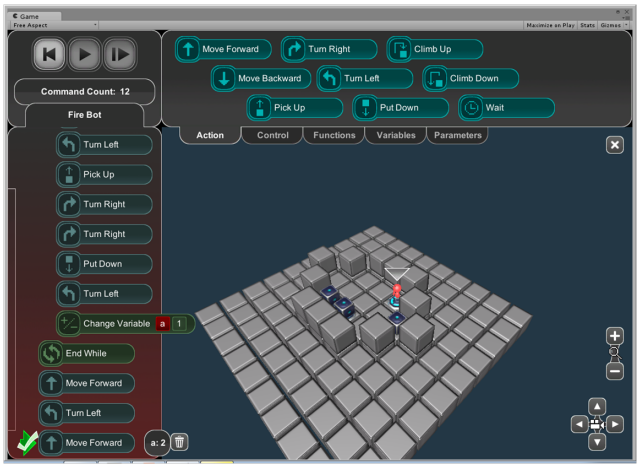

Figure 1: The BOTS interface. The robot's program is along the left side of the screen. The "toolbox" of available commands is along the top of the screen.

information and check if hints are available for the state. The action that leads to subsequent state with the highest value is used to generate a hint sequence. A hint sequence consists of four types of hints and are ordered from general hint to detailed hint. Hint provider will then show hint from top of the sequence to the student. Hint Factory has been applied in logic tutors which helps students learn logic proof. The result shows that the hint-generating function could provide hints over 80% of the time.

3. BOTS

BOTS is a programming puzzle game designed to teach fundamental ideas of programming and problem-solving to novice computer users. The goal of the BOTS project is to investigate how to best use community-authored content within serious games and educational games. BOTS was inspired by games like LightBot [10] and RoboRally [2], as well as the success of Scratch and it's online community [1] [5]. In BOTS, players take on the role of programmers writing code to navigate a simple robot around a grid-based 3D environment, as seen in Figure 1. The goal of each puzzle is to press several switches within the environment, which can be done by placing an object or the robot on them. To program the robots, players will use simple graphical pseudo-code, allowing them to move the robot, repeat sets of commands using "for" or "while" loops, and re-use chunks of code using functions. Within each puzzle, players' scores depend on the number of commands used, with lower scores being preferable. In addition, each puzzle limits the maximum number of commands, as well as the number of times each command can be used. For example, in the tutorial levels, a user may only use the "Move Forward" instruction 10 times. Therefore, if a player wants to make the robot walk down a long hallway, it will be more efficient to use a loop to repeat a single "Move Forward" instruction, rather than to simply use several "Move Forward" instructions one after the other. These constraints are meant to encourage players to re-use code and optimize their solutions.

In addition to the guided tutorial mode, BOTS also con-

tains an extensive "Free Play" mode, with a wide selection of puzzles created by other players. The game, in line with the "Flow of Inspiration" principles outlined by Alexander Repenning [7], provides multiple ways for players to share knowledge through authoring and modifying content. Players are able to create their own puzzles to share with their peers, and can play and evaluate friends' puzzles, improving on past solutions. Features such as peer-authored hints for difficult puzzles, and a collaborative filtering approach to rating are planned next steps for the game's online element. We hope to create an environment where players can continually challenge their peers to find consistently better solutions for increasingly difficult problems.

User-generated content supports replayability and a sense of a community for a serious game. We believe that user-created puzzles could improve interest, encouraging students to return to the game to solidify their mastery of old skills and potentially helping them pick up new ones.

4. ANALYSIS

4.1 Dataset

Data for the BOTS studies has come from a middle school computer science enrichment program called SPARCS. In this program, the students attend class on Saturday for 4 hours where computer science undergraduates teach them about computational thinking and programming. Students attend a total of 7 sessions, each on a different topic, ranging from security and encryption to game design. The students all attend the same magnet middle school. The demographics for this club are 74.2% male, 25.8% female, 36.7% African American, and 23.3% Hispanic. The student's grade distribution is 58% 6th grade, 36% 7th grade and 6% 8th grade.

From these sessions, we collected gameplay data for 20 tutorial puzzles as well as 13 user-created puzzles, With this data, we created an Interaction Network in order to be able to provide hints and feedback for future students [3]. However, using program edits as states, the interaction networks produced were very sparse. In order to be better able to relate similar actions, we produced another interaction network using program output as our state definition [6].

4.2 States and Transitions

Based on the data collected, we can divide the set of observed states into classes. First among these is the *start state* in which the problem begins. By definition, every player's path must begin at this state. Next is the set of *goal states* in which all buttons on the stage are pressed. These are reported by the game as correct solutions. Any complete solution, by definition, ends at one such state. Among states which are neither start nor goal states, there are three important classifications: *Intermediate states* (states a robot moves through during a correct solution), *mistake states* (states a robot does not move through during a correct solution), and *error states* (states which result from illegal output, like attempting to move the robot out-of-bounds). Based on these types of states, we classified our hints based on the transitions they represented.

4.2.1 Subgoal Transition

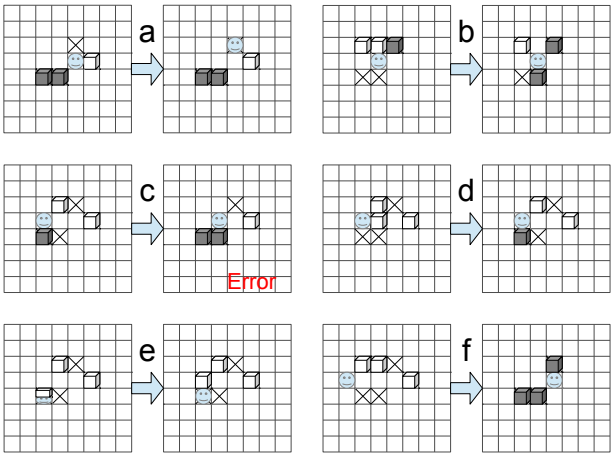

Figure 2: Several generated hints for a simple puzzle. The blue icon represents the robot. The 'X' icon represents a goal. Shaded boxes are boxes placed on goals, while unshaded boxes are not on goals. Hint F in this figure depicts the start and most common goal state of the puzzle.

(start/intermediate) → (intermediate/goal) These transitions occur when a student moves the robot to an intermediate state rather than directly to the goal. Since players run their programs to produce output, we speculate that these may represent subgoals such asmoving a box onto a specific switch. After accomplishing that task, the user then appends to their program, moving towards a new objective, until they reach a goal state. Hint B in Figure 2 shows a hint generated from such a transition.

4.2.2 Correction Transition

(error/mistake) → (intermediate/goal) This transition occurs when a student makes and then corrects a mistake. These are especially useful because we can offer hints based on the type of mistake. Hints D and E in Figure 2 show hints built from this type of transition; however, hint E shows a case where a student resolved the mistake in a suboptimal way.

4.2.3 Simple Solution Transition

(start) → (goal) This occurs when a student enters an entire, correct program, and solves the puzzle in one attempt. This makes such transitions not particularly useful for generating hints, other than showing a potential solution state of the puzzle. Hint F in Figure 2 shows this type of transition.

4.2.4 Rethinking Transition

(intermediate) → (intermediate/goal) This transition occurs when rather than appending to the program as in a subgoal transition, the user deletes part or all of their program, then moving towards a new goal. As a result, the first state is unrelated to the next state the player reaches. Offering this state as a hint would likely not help guide a different user. Hint A in Figure 2 shows an example of this. Finding and recognizing these is an important direction for future work.

4.2.5 Error Transition

(start/intermediate) → (mistake/error) This corresponds to a program which walks the robot out of bounds, into an object, or other similar errors. While we disregarded these as hints, this type of transition may still be useful. In such a case, the last *legal* output before the error could be a valuable state. Hint C in Figure 2 is one such case.

4.3 Next Steps

While this approach was able to help us identify interesting transitions, as well as significantly reduce the sparseness of the Interaction Network by merging states with similar output, we violate several assumptions of the Hint Factory technique by using user compilation as an action. Essentially, the cost of an action can vary widely. In the most extreme examples, the best next state selected by Hint Factory will simply be the goal state.

4.4 Current Hint Policy

Our current hint selection policy is the same as the one used in the logic tutor Deep Thought with a few exceptions [9]. We combine all student solution paths into a single graph, mapping identical states to one another (comparing either the programs or the output). Then, we calculate a fitness value for each node. We assign a large positive value (100) to each goal state, a low value for dead-end states (0) and a step cost for each step taken (1). Setting a non-zero cost on actions biases us towards shorter solutions. We then calculate fitness values $V(s)$ for each state s, where $R(s)$ is the initial fitness value for the state, γ is a discount factor, and $P(s, s')$ is the observed frequency with which users in state s go to state s' next, via taking the action a. The equation for determining the fitness value of a state is as follows:

$$V(s) := R(s) + \gamma \max_a \sum_{s'} P_a(s, s')V(s') \quad (1)$$

However, in our current representation there is only one available action from any state: "run." Different players using this action will change their programs in different ways between runs, so it is not useful to aggregate across all the possible resulting states. Instead, we want to consider each resulting state on its own. As a result, we use a simplified version of the above, essentially considering each possible resulting state s' as the definite result of its own action:

$$V(s) := R(s) + \gamma \max_{s'} P(s, s')V(s') \quad (2)$$

Since the action "run" can encompass many changes, selecting the s' which maximizes the value may not always be the best choice for a hint. The difference between s and s' can be quite large, and this is usually the case when an expert user solves the problem in one try, forming an edge directly between the "start" state and "goal" state. These and other "short-circuits" make it difficult to assess which of the child nodes would be best to offer as a hint by simply using the calculated fitness value.

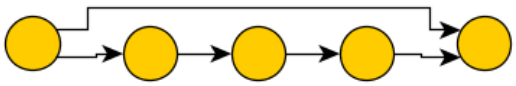

Figure 3: This subgraph shows a short-circuit where a player bypasses a chain of several steps.

Another problem which arises from this state representation is seen in Hints C and E above. These hints show states where a student traveled from a state to a worse state before ultimately solving the problem. Since we limit our search for hintable states to the immediate child states of s in s', we are unable to escape from such a situation if the path containing the error is the best or only observable path to the goal.

4.5 Proposed Hint Policies

One potential modification of the hint policy involves analyzing the programs/output on the nodes, using some distance metric $\delta(s, s')$. This measurement would be used in addition to the state's independent fitness value $R(s)$ which takes into account distance from a goal, but is irrespective of the distance from any previous state. For example in the short-circuit example above, using "difference in number of lines of code" as a distance metric we could take into account how far the "Goal" state is from the "Start" state, and potentially choose a nearer state as a hint. This also helps correct for small error-correction steps in player solutions; if the change between the current state and the target hint state is very small, we may want to consider hinting toward the next step instead, or a different solution path altogether.

$$V(s) := R(s) + \gamma \max_a \sum_{s'} \delta(s, s')P(s, s')V(s') \quad (3)$$

One potential downside to this approach is that it requires somewhat more knowledge of the domain to be built into the model. If the distance metric used is inaccurate or flawed, there may be cases where we choose a very suboptimal hint. using difference in lines of code as our distance metric, the change between a state where a player is using no functions and a state where the user writes existing code into a function may be very small. Hints selected in these cases might guide students away from desired outcomes in our game.

Another problem we need to resolve with our current hint policy, as discussed above, is the case where the best or only path to a goal from a given state s has an error as a direct child s'. One method of resolving this could be, instead of offering s' as a hint, continuing to ask for next-step hints from s' until some s' is a hintable, non-error state. This solution requires no additional knowledge of the game domain, however it's possible that the hint produced will be very far from s, or that we may skip over important information about how to resolve the error or misconception

that led the student into state s in the first place.

Other modifications to the hint selection policy may produce better results than these. We hope to look into as many possible modifications as we can, seeing which modifications produce the most suitable hints on our current dataset before settling on an implementation for the live version of the game.

5. ACKNOWLEDGMENTS

Thanks to the additional developers who have worked on this project or helped with our outreach activities so far, including Aaron Quidley, 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.

6. REFERENCES

[1] I. F. de Kereki. Scratch: Applications in computer science 1. In *Frontiers in Education Conference, 2008. FIE 2008. 38th Annual*, pages T3B–7. IEEE, 2008.

[2] R. Garfield. Roborally. [Board Game], 1994.

[3] A. Hicks, B. Peddycord III, and T. Barnes. Building games to learn from their players: Generating hints in a serious game. In *Intelligent Tutoring Systems*, pages 312–317. Springer, 2014.

[4] 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.

[5] D. J. Malan and H. H. Leitner. Scratch for budding computer scientists. *ACM SIGCSE Bulletin*, 39(1):223–227, 2007.

[6] B. Peddycord III, A. Hicks, and T. Barnes. Generating hints for programming problems using intermediate output.

[7] A. Repenning, A. Basawapatna, and K. H. Koh. Making university education more like middle school computer club: facilitating the flow of inspiration. In *Proceedings of the 14th Western Canadian Conference on Computing Education*, pages 9–16. ACM, 2009.

[8] K. Rivers and K. R. Koedinger. Automating hint generation with solution space path construction. In *Intelligent Tutoring Systems*, pages 329–339. Springer, 2014.

[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.

[10] D. Yaroslavski. LightBot. [Video Game], 2008.