An Exploration of Data-Driven Hint Generation in an Open-Ended Programming Problem

Thomas W. Price
North Carolina State University
890 Oval Drive
Raleigh, NC 27606
twprice@ncsu.edu

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

ABSTRACT

Data-driven systems can provide automated feedback in the form of hints to students working in problem solving environments. Programming problems present a unique challenge to these systems, in part because of the many ways in which a single program can be written. This paper reviews current strategies for generating data-driven hints for programming problems and examines their applicability to larger, more open-ended problems, with multiple, loosely ordered goals. We use this analysis to suggest directions for future work to generate hints for these problems.

1. INTRODUCTION

Adaptive feedback is one of the hallmarks of an Intelligent Tutoring System. This feedback often takes the form of hints, pointing a student to the next step in solving a problem. While hints can be authored by experts or generated by a solver, more recent data-driven approaches have shown that this feedback can be automatically generated from previous students' solutions to a problem. The Hint Factory [18] has successfully generated data-driven hints in a number of problem solving domains, including logic proofs [2], linked list problems [5] and a programming game [12]. The Hint Factory operates on a representation of a problem called an interaction network [4], a directed graph where each vertex represents a student's state at some point in the problem solving process, and each edge represents a student's action that alters that state. A solution is represented as a path from the initial state to a goal state. A student requesting a hint is matched to a previously observed state and directed on a path to a goal state. The Hint Factory takes advantage of the intuition that students with the same initial state and objective will follow similar paths, producing a well-connected interaction network. When this occurs, even a relatively small sample of student solutions can be enough to provide hints to the majority of new students [1].

While problems in many domains result in well-connected networks, this is not always the case. Programming problems have a large, often infinite, space of possible states. Even a relatively simple programming problem may have many unique goal states, each with multiple possible solution paths, leaving little overlap among student solutions. Despite this challenge, a number of attempts have been made to adapt the Hint Factory to programming problems [9, 12, 16]. While these approaches have been generally successful, they are most effective on small, well structured programming problems, where the state space cannot grow too large. This paper explores the opposite type of problem: one that has a large state space, multiple loosely ordered goals, unstructured output and involves creative design. Each of these attributes poses a challenge to current data-driven hint generation techniques, but they are also the attributes that make such problems interesting, useful and realistic. In this paper, we will refer to these as open-ended programming problems. We investigate the applicability of current techniques to these problems and suggest areas for future research.

The primary contributions of this paper are 1) a review of current data-driven hint generation methods for programming problems, 2) an analysis of those methods' applicability to an open-ended problem and 3) a discussion of the challenges that need to be addressed before we can expect to generate hints for similar problems.

2. CURRENT APPROACHES

Current approaches to generating data-driven programming hints, including alternatives to the Hint Factory [10, 13, 17], can be broken down into three primary components:

1. A representation of a student's state and a method for determining when one state can be reached from another, meaning they are connected in the network
2. An algorithm that, given a student's current state, constructs an optimal path to a goal state. Often we simplify this to the problem of picking the first state on that path
3. A method to present this path, or next state, to the student in the form of a hint

For the purposes of this paper, we limit our discussion to the first step in this process. (For a good discussion of the second step, see an analysis by Piech et al. [13], comparing path selection algorithms.) While in some domains this first step is straightforward, in programming tasks, especially open-ended problems, it is likely the most challenging. The simplest approach is to take periodic snapshots of a student's code and treat these as states, connecting consecutive snapshots in the network. However, because two students' programs are unlikely to match *exactly*, this approach is likely to produce a very sparse, poorly connected network, making it difficult to match new students to prior solution attempts. A variety of techniques have been presented to address this problem, which can be grouped into three main strategies: canonicalization, connecting states and alternative state definitions.

2.1 Canonicalization

Canonicalization is the process of putting code states into a standardized form, often by removing semantically unimportant information, so that trivial difference do not prevent two states from matching. Rivers and Koedinger [15] present a method for canonicalizing student code by first representing it as an Abstract Syntax Tree (AST). Once in this form, they apply a number of functions to canonicalize the code, including normalizing arithmetic and boolean operators, removing unreachable and unused code, and inlining helper functions. After performing this canonicalization on a set of introductory programming problems, they found that a median 70% of states had at least one match in the network. Jin et al. [9] represent a program's state as a Linkage Graph, where each vertex is a code statement, and each directed edge represents an ordering dependency, determined by which variables are read and assigned to in each statement. This state representation allows the Hint Factory to ignore statement orderings which are not important to the execution of the program. Lazar and Bratko [10] use the actual text of Prolog code to represent a student's state, and then canonicalize the code by removing whitespace and normalizing variable names.

2.2 Connecting States

Even with canonicalization, a student requesting a hint may not match any existing state in the network. In this case, we can look for a similar state and create a connection between them. Rivers and Koedinger [16] use normalized string edit distance as a similarity metric between two program states. They connect any two states in the network which have at least 90% similarity, even if no historical data connect these states. Additionally, they use a technique called path construction to generate new solution paths from a given state to a nearby, unconnected goal state by searching for a series of insertions, deletions and edits to their AST that will transform it into the goal state [17]. They also use this method to discover new goal states which may be closer to the student's current state. Jin et al. [9] use a similar technique to transform their Linkage Graphs to better match the current state of a student when no direct matches can be found in the interaction network. Piech et al. [13] use path construction to interpolate between two consecutive states on a solution path which differ by more than one edit. This is useful to smooth data when student code is recorded in shapshots that are too far apart.

2.3 Alternate State Definitions

Another approach is to forego the traditional code-based representation of a student's state, and use an alternate definition. Hicks and Peddycord [7, 12] used the Hint Factory to generate hints for a programming game called Bots, in which the player writes a program to direct a robot through various tasks in a 3D level. They chose to represent the state of a player's program as the final state of the game world after the program was executed. They compared the availability of hints when using this "world state" model with a traditional "code state" model, and found that using world states significantly reduced the total number of states and increased the availability of hints. The challenge with this approach, as noted by the authors, is the generation of actionable hints. A student may be more capable of making a specific change to her code than determining how to effect a specific change in the code's output.

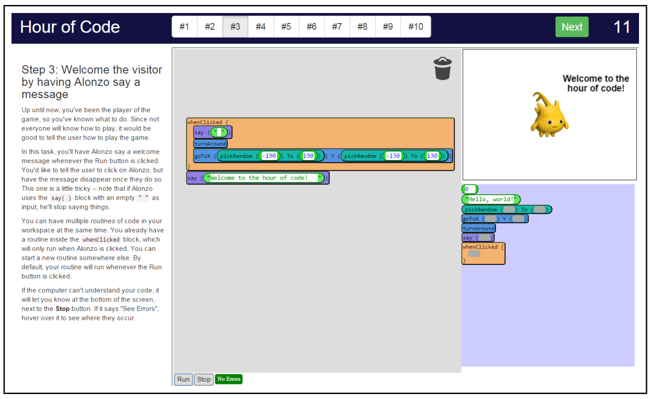
3. AN OPEN-ENDED PROBLEM

The above techniques have all shown success on smaller, well-structured problems, with ample data. We want to investigate their applicability to an open-ended problem, as described in Section 1, where this is not the case. The purpose of this paper is not to create actionable hints, nor are we attempting to show the failures of current methods by applying them to an overly challenging task. Rather, our purpose is exploratory, using a small dataset to identify areas of possible future work, and challenges of which to be mindful when moving forward with hint generation research.

We collected data from a programming activity completed by 6th grade students in a STEM outreach program called SPARCS [3]. The program, which meets for half-day sessions approximately once a month during the school year, consists of lessons designed and taught by undergraduate and graduate students to promote technical literacy. The class consisted of 17 students, 12 male and 5 female.

The activity was a programming exercise based on an Hour of Code activity from the Beauty and Joy of Computing curriculum [6]. It was a tutorial designed to introduce novices to programming for the first time. The exercise had users create a simple web-based game, similar to whack-a-mole, in which players attempt to click on a sprite as it jumps around the screen to win points. The exercise was split into 9 objectives, with tutorial text at each stage. Students were not required to finish an objective before proceeding. A finished project required the use of various programming concepts, including events, loops, variables and conditionals. The students used a drag-and-drop, block-based programming language called Tiled Grace [8], which is similar to Scratch [14]. The user writes a program by arranging code blocks, which correspond directly to constructs in the Grace programming language. The editor also supports switching to textual coding, but this feature was disabled. A screenshot of the activity can be seen in Figure 1.

During the activity, the students were allowed to go through the exercise at their own pace. If they had questions, the students were allowed to ask for help from the student volunteers. Students were stopped after 45 minutes of work. Snapshots of a student's code were saved each time it was run and periodically throughout the session. Occasional technical issues did occur in both groups. One student had severe technical issues, and this student's data was not analyzed (and is not reflected in the counts above). Students



Step 3: Welcome the visitor by having Alonzo say a message

Up until now, you've been the player of the game, so you've known what to do. Since not everyone will know how to play, it would be good to tell the user how to play the game.

In this task, you'll have Alonzo say a welcome message whenever the Run button is clicked. You'd like to tell the user to click on Alonzo, but have the message disappear once they do so. This one is a little tricky -- note that if Alonzo uses the `say( )` block with an empty `" "` as input, he'll stop saying things.

You can have multiple *routines* of code in your workspace at the same time. You already have a routine inside the `whenClicked` block, which will only run when Alonzo is clicked. You can start a new routine somewhere else. By default, your routine will run whenever the Run button is clicked.

If the computer can't understand your code, it will let you know at the bottom of the screen, next to the **Stop** button. If it says "See Errors", hover over it to see where they occur.

```
whenClicked {
  say (" ")
  turnAround
  goToX (pickRandom (-190) To (190)) Y (pickRandom (-130) To (130))
}
say ("Welcome to the hour of code! ")
```

Figure 1: A screenshot of the Hour of Code activity that students completed. Students received instructions on the left panel. The center panel was a work area, and students could drag blocks in from the bottom-right panel. The top-right panel allowed students to test their games.

produced on average 148.5 unique code states and accomplished between 1 and 6 of the activity's objectives, averaging 3.2 objectives per student.

4. ANALYSIS

We attempted to understand the applicability of each of the techniques discussed in Section 2 to our dataset. However, since the output of our program was a game that involved nondeterminism, we felt it would be inappropriate to attempt to represent a program's state as the result of its execution. We therefore focused on the first two strategies, canonicalization and connecting states.

4.1 Canonicalization

Our initial representation of a student's code state was a tree, where each code block was a node, and its children included any blocks that were nested inside of it. In this way, our representation was similar to Rivers and Koedinger's ASTs. To get a baseline for the sparsity of our dataset, we first analyzed the code states without performing any canonicalization. We calculated the total number of states in the interaction network and the percentage which were only reached by one student. Of those states reached by multiple students, we calculated the mean and median number of students who reached them. We also calculated the percentage of each student's states that were unreached by any other student in the dataset.

We then canonicalized the data by removing variable names and the values of number and string literals. Our problem featured very few arithmetic or logical operators, and these were generally not nested, so we did not normalize them, as suggested by Rivers and Koedinger [15]. We reran our analyses on the canonicalized data. To ensure that we had effectively removed all unimportant ordering information, we recursively sorted the children of each node in the tree. This effectively removed any ordering information from a student's code state, and kept only hierarchical information. This is somewhat more extreme than the Linkage Graphs of Jin et al. [9], and it does allow two meaningfully different code states to be merged in the process. We therefore see this as an upper bound on the value of removing unimportant orderings from a code state. We recomputed our metrics for the ordered-canonicalized interaction network as well. The results can be seen in Table 1. Our later analyses use the unsorted, canonicalized code representation.

	Raw	Canonical	Ordered
Total States	2380	1781	1656
% Unique	97.5%	94.8%	92.8%
Mean NU Count	3.44	3.95	2.82
Median NU Count	2	2	2
Mean % Path Unique Standard Deviation	89.9% (6.67)	83.0% (10.5)	78.9% (13.3)

Table 1: Various measures of the sparseness of the interaction network for the raw, canonicalized, and ordered-canonicalized state representations. Mean and median NU counts refer to the number of students who reached each non-unique state.

These results indicate that canonicalization does little to reduce the sparsity of the state space, with students spending most of their time in states that no other student has seen. For comparison, recall Rivers and Koedinger found 70% of states in a simple programming problem had a match after canonicalization [15], though they were using a much larger dataset. In our dataset, it is unlikely that we would be able to find a direct path from a new student's state to a goal state in order to suggest a hint.

4.2 Connecting States

To address this, we explored the feasibility of connecting a new student's state to a similar, existing state in the network. It is unclear how close two code states should be before it is appropriate to connect them as in [16], or to generate a path between them as in [17]. It certainly depends on the state representation and distance metric used. Rather than identifying a cutoff and measuring how often these techniques could be applied, we chose to visualize the distance between two students and make qualitative observations. Because our code states were already represented as trees, we used Tree Edit Distance (TED) as a distance metric. While Rivers and Koedinger reported better success with Levenshtein distance [16], we believe that TED is the most appropriate distance metric for block code, where tree edit operations correspond directly to user actions.

For each pair of students, A and B, we created an N by M distance matrix, D, where N is the number of states in A's solution path, and M is the number of states in B's solution path. $D_{i,j} = d(A_i, B_j)$, where d is the TED distance function, A_i is the ith state of A and B_j is the jth state of B. We used the RTED algorithm [11] to calculate the distance function, putting a weight of 1.0 on insertions, deletions and replacements. We also omitted any state which was identical to its predecessor state. We normalized these values by dividing by the maximum value of $D_{i,j}$, and plotted the result as an image. Three such images can be seen in Figure 2.

We also calculated the "path" through this matrix that passes through the least total distance. This does not represent a

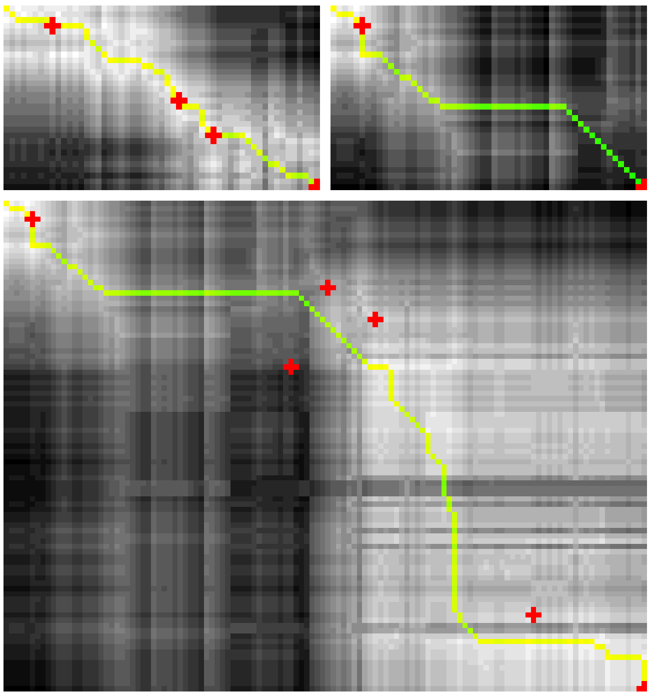


Figure 2: Three distance matrices, each comparing two students, where each pixel represents the TED between two states. Lighter shades of gray indicate smaller distances. The green/yellow line shows the path through the matrix with minimized total distance, with yellow shades indicating smaller distances. Red crosses indicate where both students met an objective. The top-left figure compares two students as they completed objectives 1-4 in the exercise. In this example, the minimum-distance line crosses each objective. The top right figure also depicts two students completing objective 4, but the darker colors and straighter line indicate less alignment. The bottom figure depicts two students completing objectives 1-6, with high alignment.

path through the interaction network, but rather an alignment between the states of student A and those of student B. Each pixel of the line represents a pairing of a state from A with a state from B, such that these pairings are contiguous and represent the smallest total distance. While we do not suggest applying this directly as a strategy for hint generation, it serves an a useful visual indicator of the compatibility of two students for hinting purposes.

An alternate approach would have been to pair each state in the interaction network with its closest pair from any other student, and use this as a measure of how sparse the network was. We chose to compare whole students, rather than individual states, because we felt that the former could lead to strange hinting behavior. Imagine, for instance, that a student requests a hint, which initially points to a state from student B, but at the very next step requests a hint that points instead to student C. Perhaps the attributes that make the student's state similar to that of B are different from those that make the state similar to C. The resulting hints would be at best confusing, and at worst conflicting.

	Mean	Median	Max	Farthest
1	0.25 (0.27)	0.15 (0.29)	0.76 (0.56)	2.23 (0.75)
2	4.88 (3.93)	4.95 (4.34)	9.18 (5.74)	12.73 (6.10)
4	4.92 (2.77)	4.83 (2.78)	10.11 (3.69)	14.67 (4.77)
5	7.79 (1.32)	7.75 (1.41)	13.17 (1.72)	18.17 (1.72)
6	7.49 (1.11)	7.76 (1.37)	13.17 (0.98)	18.67 (1.75)

Table 2: For each objective, average distances (and standard deviations) of minimum-distance student pairs, using the mean, median and max metrics. For reference, the final column represents the average maximum distance each student moved from the start state while completing the objective.

A visual inspection of these matrices reveals that while many student pairs are quite divergent, some show a notable closeness throughout the exercise. In order to quantify these results, we developed a set of distance metrics between *students*. First, the distance matrix and the minimum-distance "path" were calculated for the two students. The path is comprised of pairs of states, and for each pair, we recorded the tree edit distance between the states. From this list of distances, we calculated the mean, median and maximum distances between the two students. We looked at each objective in the exercise, and isolated the relevant subpath of each student who completed that objective. We paired each of these subpaths with the most similar subpath in the set, using the mean, median and max distance metrics. Table 2 shows the average values of these minimized pairs of students, using each metric. Objective 3, and objectives 7-9 were omitted, as too few students completed them.

5. DISCUSSION

We have attempted to apply a meaningful canonicalization to our state space, which did serve to reduce the number of states by 30.4%. However, as seen in Table 1, even after the strongest canonicalization, over 90% of the states in the interaction network had only been reached by one student, with an average 78.9% of the states in each student's solution path being unique to that student. It seems that our approach to canonicalization is insufficient to produce a meaningful reduction of the state space, though it is possible a more stringent canonicalization would be more effective.

Connecting existing states seems to be a more promising approach, giving us the ability to link new states to previously observed states, even when they do not match exactly. Our distance matrices indicate that some students take parallel, or slowly diverging solution paths, which suggests that they may be useful to each other in the context of hinting. As shown in Figure 2, students are often closest together when completing the same objective. This may seem self-evident, but it does indicate that our distance metric is meaningful. It is more difficult to put the actual TED values into context. Students get, on average, farther away from their closest paired student as they complete more objectives, but this average distance does not exceed 8 tree edits during the first 6 objectives. To put that number into context, during this same time students do not, on average, get more than 19 tree edits away from the start state. This suggest that there is certainly hint-relevant knowledge in these paired students, but that it may be difficult to harness this knowledge to generate a hint.

5.1 Limitations

It is important to note that this analysis is an exploratory case study, and makes no strong claims, only observations. We studied data from only 17 novice programmers, and the problem we analyzed was highly complex, involving multiple control structures, loosely ordered objectives, and unstructured output. This makes the problem quite dissimilar from previous problems that have been been studied in the context of hint generation, making it difficult to determine what observations should be generalized.

5.2 Future Work

Rivers and Koedinger [16], as well as Jin et al. [9] note the limitations of their methods for larger problems, and each suggest that breaking a problem down into subproblems would help to address this. Lazar and Bratko [10] attempted this in their Prolog tutor by constructing hints for individual lines of code, which were treated as independent subproblems. Similarly, we may be able to isolate the subsection of a student's code that is currently relevant, and treat this as an independent problem. The hierarchical nature of block-based coding environments lends itself to this practice, making it an appealing direction for future work.

Our work makes the simplifying assumption that unweighted TED is a reliable distance metric for code, but future work should investigate alternative metrics. This might include a weighted TED metric, which assigns different costs to insertions, deletions and replacements, or even to different types of nodes (e.g. deleting a for-loop node might cost more than a function call node). Regardless of the metric used, once two proximate states are identified, it is still an open question how this information can be best used for hint generation. It is possible to construct a path between the states and direct a student along this path. However, future work might also investigate how to extract hint-relevant information from one state and apply it to a similar state directly.

Because of the nature of our problem's output, we did not explore non-code-based state representations, as described in Section 2.3. It would still be worth investigating how this might be applied to open-ended problems. For instance, a code state could be represented as a boolean vector, indicating whether the code has passed a series of Unit Tests, and hints could direct the student to the current flaw in their program. However, creating actionable hints from this information would pose a significant challenge.

6. REFERENCES

[1] T. Barnes and J. Stamper. Toward Automatic Hint Generation for Logic Proof Tutoring Using Historical Student Data. In *Intelligent Tutoring Systems (ITS)*, pages 373–382, 2008.

[2] T. Barnes, J. Stamper, L. Lehman, and M. Croy. A pilot study on logic proof tutoring using hints generated from historical student data. In *Proceedings of the 1st Annual International Conference on Educational Data Mining (EDM)*, pages 1–5, 2008.

[3] V. Catet́é, K. Wassell, and T. Barnes. Use and development of entertainment technologies in after school STEM program. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 163–168, 2014.

[4] M. Eagle, M. Johnson, and T. Barnes. Interaction Networks: Generating High Level Hints Based on Network Community Clustering. In *International Educational Data Mining Society*, pages 164–167, 2012.

[5] D. Fossati, B. D. Eugenio, and S. Ohlsson. I learn from you, you learn from me: How to make iList learn from students. In *Artificial Intelligence in Education (AIED)*, 2009.

[6] D. Garcia, B. Harvey, L. Segars, and C. How. AP CS Principles Pilot at University of California, Berkeley. *ACM Inroads*, 3(2), 2012.

[7] 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 (ITS)*, pages 312–317, 2014.

[8] M. Homer and J. Noble. Combining Tiled and Textual Views of Code. In *Proceedings of 2nd IEEE Working Conference on Software Visualization*, 2014.

[9] W. Jin, T. Barnes, and J. Stamper. Program representation for automatic hint generation for a data-driven novice programming tutor. In *Intelligent Tutoring Systems (ITS)*, 2012.

[10] T. Lazar and I. Bratko. Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In *Intelligent Tutoring Systems (ITS)*. Springer, 2014.

[11] M. Pawlik and N. Augsten. RTED: a robust algorithm for the tree edit distance. *Proceedings of the VLDB Endowment*, 5(4):334–345, 2011.

[12] B. Peddycord III, A. Hicks, and T. Barnes. Generating Hints for Programming Problems Using Intermediate Output. In *Proceedings of the 7th International Conference on Educational Data Mining (EDM 2014)*, pages 92–98, 2014.

[13] C. Piech, M. Sahami, J. Huang, and L. Guibas. Autonomously Generating Hints by Inferring Problem Solving Policies. In *Learning at Scale (LAS)*, 2015.

[14] M. Resnick, J. Maloney, H. Andrés, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[15] K. Rivers and K. Koedinger. A canonicalizing model for building programming tutors. In *Intelligent Tutoring Systems (ITS)*, 2012.

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

[17] K. Rivers and K. Koedinger. Automating Hint Generation with Solution Space Path Construction. In *Intelligent Tutoring Systems (ITS)*, pages 329–339, 2014.

[18] J. Stamper, M. Eagle, T. Barnes, and M. Croy. Experimental evaluation of automatic hint generation for a logic tutor. *Artificial Intelligence in Education (AIED)*, 22(1):3–17, 2013.