Creating Data-driven Feedback for Novices in Goal-driven Programming Projects

Thomas W. Price and Tiffany Barnes

North Carolina State University, Raleigh NC 27606, USA,
{twprice, tmbarnes}@ncsu.edu

Abstract. Programming environments that afford the creation of media-rich, goal-driven projects, such as games, stories and simulations, are effective at engaging novice users. However, the open-ended nature of these projects makes it difficult to generate ITS-style guidance for students in need of help. In domains where students produce similar, overlapping solutions, data-driven techniques can leverage the work of previous students to provide feedback. However, our data suggest that solutions to these projects have insufficient overlap to apply current data-driven methods. We propose a novel subtree-based state matching technique that will find partially overlapping solutions to generate feedback across diverse student programs. We will build a system to generate this feedback, test the technique on historical data, and evaluate the generated feedback in a study of goal-driven programming projects. If successful, this approach will provide insight into how to leverage structural similarities across complex, creative problem solutions to provide data-driven feedback for intelligent tutoring.

1 Introduction

Novice programming environments such as Scratch, Snap, Greenfoot, Alice and App Inventor have shown promise in classrooms and informal learning settings, with evaluations demonstrating improved student grades, retention and interest in computing [2]. These environments motivate students by connecting with their interests, such as games, stories and simulations [6]. However, even in environments designed for novices, programming can still be very difficult for students to learn. Tutorials can guide students, and teachers can help them when they get stuck, but at home, in informal learning environments, or in overcrowded classrooms, teacher help may not always be available. An Intelligent Tutoring System (ITS) for programming could bridge this gap by providing feedback and guidance, helping students continue when they are unsure how to proceed. Such advice would be quite valuable in these project-driven programming environments, especially for novices whose teachers may not be expert programmers.

While feedback could be hand-authored for a programming ITS, the process is very time consuming. Data-driven feedback is an alternative approach that uses programs written by previous students to provide guidance to new students with similar programs. The Hint Factory [5] and similar data-driven techniques have been successfully applied to the domain of programming [1, 4]. However, its application is generally restricted to small, well-structured problems with easily evaluated goals, for example converting the index of a playing card (0-51) to a string representation (e.g. "7H" for the 7 of hearts).

By contrast, goal-driven projects, such as programming a game, story or simulation, require solving multiple, loosely ordered and interdependent subgoals. Such projects may be defined by a set of instructions (e.g. a tutorial), which outline these goals, but the student is given room to make design choices within the framework of the instructions. Further, evaluation of these goals may be a somewhat subjective process. The ill-defined and open-ended nature of these projects pose a challenge for current data-driven methods. Our goal is to adapt current methods to overcome these challenges.

Our research questions are as follows. Can data-driven feedback in goal-driven programming projects:

1. Be generated such that feedback is available in the majority of situations
2. Increase student performance in the environment
3. Improve students' self-efficacy with regards to programming

2 Subtree-based State Matching

Hint-Factory-style data-driven techniques represent a problem as a space of possible states [5]. All students start in the same state and attempt to find a path to some goal state. Solution paths from previous, successful students are combined to form a network, and each state is assigned a value, approximating the likelihood of a student reaching a goal from that state. When a new student requests help, the system matches the current state of that student to a state in the network and suggests a path that will lead to a goal state.

In programming, the most straightforward representation of a student's state is the program itself, though other representations have been proposed (e.g. [1]). Two students are unlikely to have exactly matching programs, however, making it difficult to generate feedback. This problem was recognized by Rivers and Koedinger [3], who addressed it by transforming programs into Abstract Syntax Trees (ASTs) and performing canonicalization to increase the probability of overlapping states. They also connected states in the network which had at least 90% similarity, creating additional paths to the goals states. If no existing path is found, path construction can be applied to generate a new path to a goal state [4]. These techniques have been effective with small, well-structured problems. However, our preliminary results on goal-driven projects suggest that the resulting state space is much more sparse and resistant to canonicalization.

We therefore propose a subtree-based state matching approach that will isolate and match relevant sections of code, rather than entire programs. Like Rivers and Koedinger, we represent a program as an AST. When a student requests help, we isolate the subtree that encompasses their most recent actions. For instance, a student might be working only on code within a specific for-loop, and so we identify the subtree that is the body of that loop. Rather than finding an entire state which matches the student, we look for any states with a matching subtree and find advice that applies within that subtree. If multiple matches are found, we can look outside the subtree to determine which state is the closest match. If no exact matches are found, we can still apply previous techniques (e.g. path construction) to the subtrees.

For example, consider a student trying to implement code that will move a game character to a random position on the screen. In the previous section, this student added some extra code, which makes her current program state dissimilar from any previously observed state. However, the random movement logic is encompassed within a subtree. By isolating this subtree, we can still match her to the movement logic of a student who completed that goal.

3 Current Progress

We have already collected log and survey data from middle school students completing an hour-long goal-driven project, in both a block-based and textual programming environment. Students employed variables, loops and conditionals to create a simple Whack-a-Mole style game, where the objective is to click on a sprite as it jumps around the screen to win points. Our results indicate that students perform significantly better using a block programming interface, spending a larger portion of their time on-task, and completing more goals in less time. Therefore, since students can do more work in less time, and blocks prevent some kinds of errors, we will focus on block-based environments.

We constructed a network from the collected program logs to analyze the potential for applying current data-driven techniques. We applied strong canonicalization to the programs, removing all program-specific variable names and standardizing statement ordering, to achieve as much overlap between programs as possible. Even then, over 92% of the observed states were unique, confirming the intuition that open-ended, goal-driven tasks naturally have sparse state spaces. Further, states from different students were very dissimilar, even when they were pursuing the same subgoal, making it difficult to construct additional paths, as in [4]. However, inspection of the data revealed that even very dissimilar program states still contained similar or identical sections of code. For example, some students who had different solutions to one subgoal of the activity, had a similar solution to another subgoal. This suggests that the subtree technique would be successful with this data, which we seek to verify in future work.

4 Future Work

We plan to use the data we have already collected to evaluate the feasibility of our approach, specifically our ability to generate on-demand feedback in the form of hints. We will implement the subtree state matching algorithm and adapt the Hint Factory to generate hints from matching subtrees. We will use a technique called a Cold Start analysis to predict how much data will be necessary to provide hints to students at least 50% of the time, which is our goal. If too much data is required, we can modify the algorithm to increase hint availability (at the cost of applicability) by matching subtrees containing only a student's most recent action, rather than a set of recent actions. We will then determine which programming constructs we want to focus on and create a goal-driven project that incorporates them.

Our evaluation will target middle school students, and will be in two parts. First we will collect data from a group of students as they complete the project, who will serve as a control group. We will then use this data to generate hints and modify the programming environment to provide them on-demand. We will perform a second data collection using the hint-enabled environment, otherwise identical to the first. We will collect log data to determine how often feedback was available when requested. We will evaluate how well each group performed, as measured by the number of subgoals completed and the time required for completion. We will also measure how students' self-efficacy with regards to computing changed after completing the exercise, using a validated pre- and post-survey. We hypothesize that the hint group will outperform the control group and demonstrate higher self-efficacy.

The work proposed here leaves some open questions, which will need to be addressed. While subtree-based state matching will allow us to generate feedback in more states, the quality of the feedback will need to be evaluated. What impact will statements outside of a matched subtree have on the applicability of the generated feedback? When evaluating the technique, how complex should the goal-driven project be, and what programming concepts should it cover? What other measures could be used to evaluate the impact of the hints?

The work proposed here will represent a significant advance for automated feedback generation, expanding its application to open-ended design tasks, such as goal-driven projects. It lays the groundwork for a programming ITS capable of aiding students with programming at a novice level and beyond.

References

1. Hicks, A., Peddycord III, B., Barnes, T.: Building Games to Learn from Their Players: Generating Hints in a Serious Game. In: Intelligent Tutoring Systems (ITS). pp. 312–317 (2014)
2. Moskal, B., Lurie, D., Cooper, S.: Evaluating the effectiveness of a new instructional approach. ACM SIGCSE Bulletin 36(1), 75–79 (2004)
3. Rivers, K., Koedinger, K.: Automatic generation of programming feedback: A data-driven approach. In: The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (2013)
4. Rivers, K., Koedinger, K.: Automating Hint Generation with Solution Space Path Construction. In: Intelligent Tutoring Systems (ITS). pp. 329–339 (2014)
5. Stamper, J., Eagle, M., Barnes, T., Croy, M.: Experimental evaluation of automatic hint generation for a logic tutor. Artificial Intelligence in Education (AIED) 22(1), 3–17 (2013)
6. Utting, I., Cooper, S., Kölling, M.: Alice, Greenfoot, and Scratch–a discussion. ACM Transactions on Computing Education (TOCE) 10(4) (2010)