The Thirty-First International Florida
Artificial Intelligence Research Society Conference (FLAIRS-31)

Linkage Objects for Generalized Instruction in Coding (LOGIC)

Ted Carmichael,[1,2] Mary Jean Blink,[2] John Stamper,[2,3] Elizabeth Gieske[2,4]

[1]University of North Carolina at Charlotte, Charlotte, NC, USA
[2]TutorGen, Inc., Fort Thomas, KY, USA
[3]Carnegie Mellon University, Pittsburgh, PA, USA
[4]Northern Kentucky University, Highland Heights, KY, USA
tedsaid@gmail.com, mjblink@tutorgen.com, john@stamper.org, gieskee1@nku.edu

Abstract

Linkage Objects for Generalized Instruction in Coding (LOGIC) is an intelligent system for online tutoring which detects errors among programming exercises to improve understanding of student progress. This system represents an implementation of the Hint Factory method for automated hint generation. In this approach, variables and their dependencies are abstracted from correct coding solutions to determine all the possible paths towards a solution, regardless of the programming language or variable names. Incomplete programs can be compared to these unique paths after code normalization, and the next best line can be supplied in the form of a hint. Errors are recorded based on discrepancy between best-match and the student's code. The final report categorizing all errors is compiled to benefit the teacher's effectiveness, highlighting common errors made by students.

Introduction

The project Linkage Objects for Generalized Instruction in Coding (LOGIC) represents a revolutionary advance in adaptive educational systems for computer programming by using data collected from previous exercises to automatically generate hints and feedback for students coding in an integrated development environment (IDE). This work aims to help students succeed in developing solid coding skills and best coding practices. Previous Intelligent Tutoring Systems (ITSs) for computer programming have been shown to be extremely effective, yet have not gained widespread use due to the cost of rebuilding the tutors when the current programming languages change. LOGIC is designed to address this issue with a generalizable model that uses data to create the tutor, so that any programming language can be used.

Background

Computer science education is increasingly important to the country's and the world's prosperity and advancement. Congress recently passed a sweeping update of US education policy, designating computer science as a core contributor to a well-rounded education. Further, President Obama highlighted the importance of computer science education in his final State of the Union address. In 2017 it was announced there would be a $200 million per year commitment for computer science education in America's schools. Tech firms, such as Google, Amazon, Salesforce, Microsoft, and Facebook have pledged an additional $300 million dollars towards this effort. Computer literacy is quickly becoming one of the most important skills for interesting and well-paying careers.

Decades of previous research has shown that static, one-size-fits-all training is not as efficient or as effective as adaptive instruction that changes based on the student's current understanding of the material. This is also true of adaptive tutors for programming (Koedinger et al. 1995; Corbett and Anderson, 1995). An intelligent tutor for programming is not a new idea. The LISP tutor was the first ITS for programming (Anderson and Skwarecki 1986) and was a cognitive tutor using production rules (Corbett and Anderson, 1992). Students using the LISP tutor completed coding assignments 30% faster and scored 43% better on a post-test than students doing coding assignments without a tutor (Anderson, Conrad, and Corbett 1989). Help-seeking behaviors were studied with the LISP tutor (Anderson and Reiser 1985), and this research showed that tutors containing intelligent feedback exhibited time improvements with no loss in accuracy. Using the same framework, a later version called the ACT Programming tutor was implemented in LISP, Prolog, and Pascal; however, these tutors are no longer in use today. Experts were required to create the production rules used in these tutors, which made it challenging to port the methodology to new languages.

Although SQL is not a true programming language, there have been tutors created to help teach SQL in the context of database queries. One such tutor is the SQL Tutor (Mitrovic and Martin 2002), which is a constraint-based tutor. These tutors are only concerned with the student's current state and problem-constraint violations within that state. The hints provided, therefore, are "one size fits all,"

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

based on the current state rather than the type of learner. These tutors generally require less time to construct and tend to work well for less procedural problems. SQL Tutor has been used extensively and today is supplied as a free web-based supplement to Addison Wesley's SQL textbooks (since 2006). Studies have shown that students using the tutor scored 11.5% points higher than those not (Mitrovic and Ohlsson 1999). The constraints are created by experts for the set of commands used in SQL.

One ITS for coding, specific for C++, uses a method termed Pre-programming Analysis Guided Programming or PAGP (Jin 2008). This tutor presents the student with a series of drop down options to choose the correct next line from a group of available options. As such it requires experts to manually input the options for each programming step, making it generally cost-prohibitive to implement.

With the techniques developed during this project, we can add such adaptability automatically. Using an extension of the Hint Factory methods (Stamper, Barnes, and Croy 2007), LOGIC learns the content and the context that previous students used to successfully solve a programming exercise, and uses this knowledge to provide next-step hints. This technique can be applied to any type of programming language provided an interface exists to capture student solutions.

While most IDEs provide syntactical assistance, there are no products that have an integrated solution for assistance on higher-level development strategies, which represents a clear drawback of current technology. To address this need, this work introduces the idea of linkage objects, which are objects that allow abstraction of different code segments of computer programs into comparable objects. These objects can be linked into a graph of known possible paths. Applying Markov models to this path allows the technology to present the learner with context-specific hints and feedback as they are completing the exercise.

By using these methods to improve educational software, we can dramatically reduce the costs of adding hint capabilities, expanding the market for computer-based training in programming without necessitating a complete redevelopment for every new programming language (or updated existing language).

This effort builds on previous successful implementation of the Hint Factory in a tutor used to teach deductive logic through solving logic proofs. When a student is stuck, the Hint Factory can automatically generate a hint by suggesting the next best step, based on the graph of solutions created from previous student attempts. Previous results have shown that students using a logic tutor augmented with the Hint Factory attempted and completed significantly more logic proof problems, were significantly less likely to abandon the tutor, and performed better on a post-test implemented within the tutor (Stamper, Barnes, and Croy 2010).

Traditional intelligent tutors work by giving students hints and feedback adapted to the individual student. Marking student work as right or wrong is a simple form of feedback that can often be automated, but automatically generating *formative* feedback is a much more complex problem. Shute's review of the literature suggests that effective formative feedback should be multidimensional and credible, specific but not evaluative, and infrequent but timely (Shute 2008). Determining the timing and frequency of hints is a particular challenge, but studies suggest that offering hints based on student choice can have positive effects on learning (Razzaq and Heffernan 2010; Aleven et al. 2004; Shih, Koedinger, and Scheines 2008). Previous research has shown that students often know how to execute the steps needed to solve multi-step problems but may have trouble choosing what to do next (Stamper, Barnes, Croy 2008). Our on-demand, context-specific system addresses this issue, and can be applied in many domains.

Methodology

The foundation of the LOGIC framework consists of the Markov decision process (MDP) generator and the hint provider. The MDP generator is an offline process that assigns values to the states that have occurred in student problem attempts once they have been made into a graph. These values are then used by the hint provider to select the next "best" state or action at any point in the problem space. An MDP is defined by its state set S, action set A, transition probabilities T, and a reward function R. The goal of using an MDP is to determine the best policy, or set of actions students have taken at each state *s* that maximize its expected cumulative utility. The expected cumulative value function can be calculated recursively using value iteration. For a particular point in a student's problem attempt, a state consists of the list of the features the student has generated so far. In our previous work for a logic tutor, the features of the states were simply the premises and statements that had been derived at that point in the student's solution. The actions in the logic tutor implementation were the rules used at each step. Actions are directed arcs that connect consecutive states.

Therefore, each problem attempt can be seen as a graph with a sequence of states connected by actions. We combine all student solution graphs into a single graph, by taking the union of all states and actions, and mapping identical states to one another. Once this graph is constructed, it represents all of the paths students have taken in working a problem. Next, value iteration is used to find an optimal solution to the MDP. We apply value iteration using a Bellman backup to iteratively assign values to all states in the MDP until the values converge. By identifying a new student's state in the graph we can point them to the next state in the graph with the highest state value.

When the MDP generator is applied to example data, a Markov chain consisting of six states is created. This Markov chain can be seen in Figure 1. Each state is con-

nected via the action that was used to reach that state. Combining the Markov chain seen in Figure 1 with all other student attempts for this problem creates a large graph of all “known” student solutions.

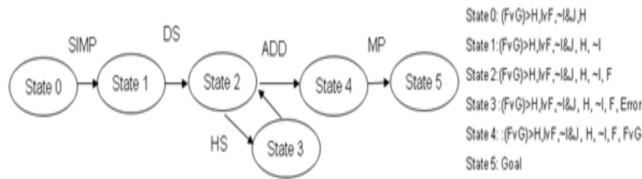

Figure 1. Markov Chain showing the states and actions created from the example data.

To make the abstraction of state feature for computer programming we introduce the concept of “linkages.” A linkage is a graph following the use of a distinct variable in a program. Linkages can provide one abstraction for state features in coding and may be enough to provide students with useful hints. Normalizing the variables in each linkage helps identify identical linkages across many students’ programs. A code snippet of a simple example program and its linkages can be seen in Figure 2.

The program asks for the base and the height of a triangle and then outputs the area. In the right box of the figure, the three created linkages are seen. One linkage is created for each variable used in the program, normalizing variable names. Even with this small example, we could imagine many ways that students could calculate the area of a triangle. Instead of [A = .5(B*C)], one student might try [A = .5(C*B)], or another might try [A=(B*C)/2].

```
double triBase;
double triHeight;
double triArea;

 cout<< “Enter the base of the triangle";   //user enters the length
 cin>> triBase;                            //input  stored in memory

 cout<< “Enter the height of the triangle";   //user enters feet
 cin>> triHeight;                            //input stored

 //calculation of triangle area
 triArea=.5(triBase*triHeight);

// output Area
cout << “The area of the triangle is: " << triArea;
```

Linkages

A → cin>>A → C=.5(A*B)
A → cin>>A → C=.5(B*A)
A → A=.5(B*C) → cout<<A

Figure 2. Sample code snippet used to calculate the area of a triangle and the resulting linkages.

With the Hint Factory, all of these ways are fine, but if the student requests a hint on a path just before this equation, the Hint Factory will give them a hint along the path that is currently closest to their own that has the highest chance of leading them to a correct solution. More frequent linkages get a higher value, and are used more often.

For data collection, we used the open-source, web-based IDE CloudCoder (Figure 3), because it already includes a platform for creating and assigning short programming exercises for a variety of programming languages, and currently includes C/C++, Java, Python, and Ruby. CloudCoder is web-based, letting students write code directly in a web browser, click the "submit" button, and receive immediate feedback. Unlike other systems, which tend to be closed, commercial, or both, CloudCoder is a completely open platform. The code for the system is open-source, and exercises written for CloudCoder may be shared to a central repository under permissive licenses. And significantly, CloudCoder is designed to provide formative feedback for students, automatically grading each submission based on test-cases.

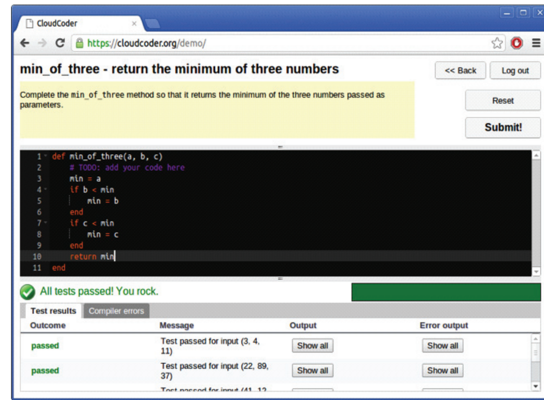

Figure 3. Screen shot of CloudCoder web-based IDE.

Results

We collected student data across multiple classrooms and two different programming languages (C++ and Java), we processed and analyzed this data, providing abstracted results (matrices), hint coverage, and error classification. Here we implemented the LOGIC method using two processes: one process works with correct submissions, while the second compares the incorrect solutions to the correct submissions to find a detrimental error. The first process determines unique paths by creating matrices representing the occurrence of variables on each line. By abstracting the variables in this way, we can create matrices that can be compared across programming languages. Further, this ensures that superficial differences, such as students using different variable names or declaring variables in a different order, won’t be used to wrongly classify solutions.

The second process dynamically compares the erroneous code’s matrix to each of the matrices representing correct submissions. After finding a similar match, the system normalizes the code and analyzes each line using text comparison. Once a difference is found, the system further analyzes the line to determine the line’s function and specific error. Occasionally, a line that is not identical to the matching code is actually performing the same function. For instance, the line “sum += i” is producing the same result at “sum = sum + i.” With string manipulation and comparisons, LOGIC is able to recognize when superficially different lines are essentially the same.

Each line type has specific errors associated with it. For example, the for-loop line type encompasses the following

errors: incorrect boundary variables, off-by-one error due to comparison symbol, failed to include for-loop, used an unneeded for-loop, iteration error, and comparison symbol in wrong direction. This list is ordered from most to least frequented mistakes in the C++ dataset. All of the errors are tallied and visually displayed for the instructor.

Four problems were selected for testing, ones that contained ample incorrect and correct student submissions. These four problems were: add elements in an array, determine if an array is ascending, determine the largest of three numbers, and compute the sum between two numbers. Analyzing the C++ coding language, LOGIC correctly identified an error in 234 out of 260 wrong submissions, resulting in 90% accuracy. (The IDE used here will identify correct solutions already; hence, there is no need to test for false positives.) The system also flagged 15 submissions containing hardcoded answers. LOGIC correctly identified errors in 88 of the 104 Java submissions and detected one hardcoded submission resulting in 84% accuracy. Interestingly, students among both languages carried similar error tendencies, with for-loops and return statement errors being the most popular, while errors in method headers and variable assignments were the least common.

Discussion

The process for abstracting out the variables, and basing each program's uniqueness on the variable matrix is largely successful here, representing a crude but effective method for abstraction and comparison across students and languages. Our further analysis, examining and classifying the types of errors, indicates that such errors tend to follow similar patterns, at least for these problems and programming languages. This indicates that it may be tractable to customize error-type classification for widespread use.

LOGIC's hint generation can be implemented by offering the next matching line to the student. However, as seen above, a replaced line is sometimes not what the student needs. Conversely, a student might need to ask for a hint more than once, if a single correction is not going to fix the program, and the student has trouble seeing where the first correction is leading.

However, for LOGIC to be able to give hints, it only requires one previously successful solution that matches a current student's path. Further, we can always back up a path on the graph until we find a match, which means we can always give a hint, though the hint will be in better context the closer it is to the student's current state.

Based on these initial results we believe the use of linkages as state features will be applicable to computer programming given enough data. An initial list of good state features will be derived using the linkages extracted from programming examples that are complete and correct. The combination of all linkages for one program makes a single graph of states and actions that can be used as a MDP. Further research across complex programs in our C++, Java, and Python datasets will provide a stronger basis for extending the Hint Factory framework for programming.

References

Koedinger, K. R., Anderson, J. R., Hadley, W. H., & Mark, M. A. (1995). Intelligent tutoring goes to school in the big city. In Proceedings of the 7th World Conference on Artificial Intelligence in Education.

Corbett, A.T. and Anderson, J.R. (1995). Knowledge decomposition and subgoal reification in the ACT programming tutor. Artificial Intelligence and Education, 1995: The Proceedings of AI-ED 95. Charlottesville, VA.

Anderson A., Skwarecki, (1986). The Automated Tutoring of Introductory Computer Programming. *Communications of the ACM* 29: pp. 842-849.

Corbett, A. T. & Anderson, J. R. (1992). The LISP intelligent tutoring system: Research in skill acquisition. In J. Larkin, R. Chabay, and C. Scheftic (Eds.), *Computer Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration*. Hillsdale, NJ.

Anderson, JR; Conrad, FG; & Corbett, AT (1989). Skill acquisition and the LISP Tutor. *Cog Science*, 13, 467-506.

Anderson, J.R., & Reiser, B.J. (1985). The LISP tutor. *Byte*, 10, 159-175.

Mitrovic, A., Martin, B., (2002). Evaluating the Effects of Open Student Models on Learning. *Second International Conference on Adaptive Hypermedia and Adaptive Web-Based Systems*. Berlin, Springer-Verlag: 296-305.

Mitrovic, A., Ohlsson, S. (1999). Evaluation of a Constraint-Based Tutor for Database Language. *Int. Journal of Artificial Intelligence in Education* 11(2): pp 238-256.

Jin, W. (2008). Pre-programming Analysis Tutors Help Students Learn Basic Programming Concepts. *In Proceedings of ACM SIGCSE Technical Symposium on Computer Science Education.* March 12-15, Portland, OR, USA.

Stamper, J. C., Barnes, T., & Croy, M. (2007). Extracting student models for intelligent tutoring systems. In *Proc of the Natl Conf on Artificial Intelligence* (22, 2, p.1900). AAAI Press; MIT Press.

Stamper J., Barnes T., Croy M. (2010) Enhancing the Automatic Generation of Hints with Expert Seeding. *In: Aleven V., Kay J., Mostow J. (eds) Intelligent Tutoring Systems. ITS 2010. LNCS,* vol 6095. Springer, Berlin.

Shute, V.J., Focus on formative feedback. *Review of Educational Research, 2008. 78(1): p. 153-189.*

Razzaq, L. & Heffernan, N. (2010). Hints: Is It Better to Give or Wait to be Asked? In *Aleven, V., Kay, J & Mostow, J. (Eds). Proc. 10th Intelligent Tutoring Systems (ITS2010) Part 1. Springer. Pages 349-358.*

Aleven, V., McLaren, B., Roll, I., & Koedinger, K. (2004). Toward tutoring help seeking: Applying cognitive modeling to metacognitive skills. *Proc. of the Seventh International Conference on Intelligent Tutoring Systems.*

Shih, B., Kenneth R. Koedinger, and Scheines, R. "A Response Time Model For Bottom-Out Hints as Worked Examples." In *Educ. Data Mining*, 2008, 117-126.