A Survey of Intelligent Programming Tutors

Michael Neary

March 16, 2017

Abstract

I survey the field of Intelligent Tutoring Systems as applied to introductory programming, Intelligent Programming Tutors (IPT). I introduce the concept of an IPT along with motivation for continued development. I discuss the three components of an IPT: domain knowledge, student knowledge, and constructive feedback. I focus particularly on the generation of constructive feedback and open problems therein.

1 Introduction

An Intelligent Tutoring System (ITS) provides students with one-on-one guided instruction in a particular subject, typically after they had prior exposure to that subject in a traditional classroom setting. A typical ITS mimics the interaction between a human tutor and a student. The goal of this interaction is to bring the student from a state of confusion in a subject to mastery of that subject. It develops an internal model of what the student understands, providing the student with feedback based on that model to help guide the student towards mastery. The successful implementation of a tutoring system can be rewarding for both student and teacher, relieving some of the burden of teaching and learning in particularly difficult courses.

A majority of the content in introductory programming courses is difficult to grasp for the novice student, especially considering that the material is almost en-

tirely novel. Those who fall behind in an introductory course are likely to suffer learned helplessness because each new concept builds upon previous concepts [3]. Students also find it difficult to combine the basic structures of programming to form solutions to larger problems. If a student is able to think of a programmatic solution, they may still have trouble fixing bugs within that solution [6]. It is also difficult for the instructor to account for these varying issues when teaching an introductory course. Students learn different concepts at different rates, which makes deciding what programming concepts to teach in what sequence particularly hard [9]. The difficulty of programming for first-time learners and their instructors justifies the application of ITS to introductory programming.

An Intelligent Programming Tutor (IPT) is a specific implementation of an ITS for introductory programming. There are three important components of an IPT: domain knowledge, student knowledge, and tutoring knowledge. An IPT needs *domain knowledge*, or a method of representing concepts related to programming (e.g., language syntax), in order to identify define content mastery. It needs a model of *student knowledge*, or a method of encoding student understanding based on observation (e.g., code samples, multiple choice answers). An IPT also needs to know *how to tutor*, or how to provide meaningful feedback to a student in a timely manner. Exceptional IPTs must have all three components, as constructive feedback is best generated from the combination of an accurate student model and sound domain knowledge. I explore each of these components in detail in the coming sections, focusing particularly on feedback. I plan to focus on the problem of producing meaningful feedback in an IPT in my thesis work.

2 Encoding of Domain Knowledge

It is imperative that an ITS has expert-level domain knowledge in the subject area it is applied to. An IPT needs knowledge of programming concepts in order to model what a student knows and to give feedback to that student. The necessary programming concepts for an IPT are: how to write programs, how to identify and

describe errors, and declarative information on programming structures [8]. These concepts should be encoded in an easily accessible manner, such that the student model and feedback generator can leverage the information. Programming domain knowledge need not be a separate entity in an IPT, although there are methods that rely on domain knowledge as a separate entity. One can easily incorporate domain knowledge into a student model or feedback generation method.

One direct method of domain knowledge representation utilizes first-order logic to create a knowledge base of programming concepts. This method views a solved programming problem as a conjunction of predicates that describe the expected constructs in an ideal solution. Weregame and Reye translated a set of student code examples into a knowledge base of correct solutions. They leverage this method of domain knowledge encoding to administer tutoring through simple planning algorithms. This method creates an accurate tutoring system, and is suited for handling the differences in solutions to problems [11].

Some IPTs encode domain knowledge as a graph to capture the relationship between programming concepts. Understanding the relationship among programming concepts is crucial for navigating the tutoring process in an IPT. If the system understands what the student has not yet mastered, and it knows the similar concepts that they do understand, it can generate feedback to bridge that gap. Kumar extends the graph of programming concepts, adding learning objectives to each concept [5].

Most intelligent tutors that I have come across mold domain knowledge into either the student model or the method by which they give student feedback. The advantage to this strategy is clear in either case: it increases the efficiency of the system.

3 Model of Student Knowledge

It is important for the interaction between an intelligent tutor and a human to be as personalized as possible, since such a system must have an understanding of what the current user is capable of. It should reason from what it observes of the student's

actions to determine the content that student has learned. This is a difficult task due to the of uncertainty entwined in the observation of a student. A variety of methods for the representation of student knowledge that handle this uncertainty currently exist. A few of these methods utilize classic techniques such as Bayesian networks and Markov Decision Processes, while others employ novel methods. I first discuss the classic techniques applied to this problem, then more recent methods.

A Bayesian network is a directed acyclic graph used to encode conditional dependencies among random variables. These networks are the most utilized method for the modeling of student knowledge because they are easy to implement and they can determine the probability that some concept is understood given evidence, or lack thereof, for that hypothesis. Butz et al. use a Bayesian network to model the student who is using the system, and update their model based on the student's self-assessed understanding [1]. Chang et al. go a step further by describing the modeling problem in relation to time to apply a dynamic Bayesian network, achieving a better model overall at the cost of training time [2].

A Markov Decision Process can also be used as a model for student behavior in a tutoring system. This technique determines an optimal policy for the intelligent tutor given a set of actions it can take (feedback to the student) and the states that result from taking those actions (student understanding of the material), with some probability that action A results in a transition from state S to S′. The problem of student modeling can be constructed as a Partially Observable Markov Decision Process (POMDP) because there is a level of uncertainty in the result of an action taken by a student. The model must do what it can with unreliable evidence, since a piece of feedback the student receives may either help or hurt their understanding of the topic. A student's response to a problem can vary, and it is not certain what state of understanding they are in until after their response in analyzed. A POMDP is a powerful modeling method, but can become intractably complex. Folsom-Kovarik et al. propose two variations—state queues and observation chains—on a POMDP to solve the problem of increased complexity in modeling student knowledge. Both

methods use properties of tutoring tasks to compress the information needed to reliably model a student with a POMDP. Their results show that their compression methods do not have any negative effect on the student model, but there was no substantial improvement over existing modeling methods.

4 Method of Constructive Feedback

Quality feedback is important to facilitate the acquisition of new knowledge. The majority of feedback that a new programmer receives comes from the compiler or interpreter of the language they are using, which can be hard to decipher. This type of feedback is useless to the novice beyond highlighting syntax errors. It is therefore up to the instructor to provide meaningful feedback on student programming work, ranging from proper syntax use to walking through flawed logic. An IPT removes the need for the instructor to intervene and correct a flaw in student thinking. It should provide meaningful hints towards valid solutions and correct explanations of errors that arise when programming.

Stamper et al. laid the groundwork for constructive feedback in an intelligent tutor with the Hint Factory [10]. They created a system for a logic tutor that could generate step by step hints as a student was attempting to solve a problem. This system used data from past student problem solving attempts to create a Markov Decision Process, effectively creating a student model that was not individualized. Using this model the system would figure out what the next best problem solving state was, and generate a hint to get the student into that state. Through the use of this hint generation method, they found that students increased the number of times they attempted to answer questions, and increased the number of questions they solved overall. The students who used the hint generation system achieved higher scores on post tests in their experiments and received higher course grades overall. The Hint Factory was meant to be generalizable to all sorts of intelligent tutors, but there are limitations to using this system in a tutoring environment for programming.

Generating hints for student trying to solve a programming problem can be difficult to compute. The solution space for a single programming problem can be infinitely large unless a bound is placed on the syntax or standard library functions that are allowed in a solution. Searching through a solution space to find the most similar solution to what a student has is therefore unreasonable. Current methods for hint generation in this space either circumvent the need for looking at the solution space, or take steps to pare down its size.

There have been a few approaches for improving hint generation and programming feedback in an IPT. Recently, there has been a push towards the use of programming data to generate feedback. One method was developed by Lazar and Bratko without the need for a state-space representation of the problem solving process, instead analyzing student programs by looking for common "edits" between them [7]. Once these edits were found, they would apply them to a student submission until it resulted in a solution. From the sequence of applied edits, they derived hints they would show to the student. Using this approach they were able to fix 70% of student submissions, independent of language choice without the need to manually enter common mistakes for the system to be aware of.

Other approaches treat the problem solving environment of learning to program with a state-space representation method. Koedinger et al. generate a solution space using Abstract Syntax Trees created from correct student code samples, and they reduce the size of this solution space by applying certain transformations on these ASTs [4]. Using the reduced solution space they create the AST for a student's intermediate submission, and then compute the delta between the intermediate solution and the known solutions. The goal is then to reverse the differences to generate feedback from the closest solution found.

Expanding on their previous results, Rivers and Koedigner describe a path construction method, where instead of comparing all possible solutions, they develop a space of possible paths that a student may take in order to get to a solution, and generate hints based on the most likely path that a student could take given the pre-

vious attempts [9]. This method was more computationally feasible than generating the solution space, and they found most hints to be generated fairly quickly (in less than one minute). However, they are not certain how helpful the hints generated were for the students interacting with the system.

4.1 Thesis Motivation

Generating hints based on the difference between a decent, correct solution and the student's current attempt can be limiting in the types of hints that are generated. There are many solutions to the same problem, and trying to force a student into a particular solution may confuse the student. They may not have meant to go in the direction the system's hints is taking them. In fact, they could have lines of unnecessary code that confuse current methods of hint generation. I believe this to be an oversight in current hint generation systems: most methods simply ignore the potentially important information contained in unnecessary code. I have not found any literature that addresses this issue.

How can a hint generation take this into account? Can it identify lines that code that are unnecessary, ones that make no sense in the context of the problem? Given that it can identify this kind of code, what does it tell you about the student's train of thought and how can you give a hint to fix their misunderstanding(s)?

References

[1] C.J. Butz, S. Hua, and R. B. Maguire. Bits: A Bayesian Intelligent Tutoring System for Computer Programming. *Wccce '04*, pages 179–186, 2004.

[2] K-M. Chang, J Beck, J Mostow, and a Corbett. A Bayes Net toolkit for student modelling in intelligent tutoring systems. *Proceedings of the 8th International Conference on Intelligent Tutoring Systems (ITS 2006)*, pages 104–113, 2006.

[3] Tony Jenkins. On the Difficulty of Learning to Program. In *3rd Annual LTSN-ICS Conference*, pages 53–58, 2002.

[4] Kenneth R. Koedinger, Emma Brunskill, Ryan S.J.d. Baker, Elizabeth A. Mclaughlin, and John Stamper. New potentials for data-driven inteligent tutoring system development and optimization. *AI Magazine*, 34(3):27–41, 2013.

[5] Amruth N. Kumar. An Evaluation of Self-explanation in a Programming Tutor. *LNCS*, 8474:248–253, 2014.

[6] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, 37(3):14–18, 2005.

[7] Timotej Lazar and Ivan Bratko. Data-driven program synthesis for hint generation in programming tutors. In *Lecture Notes in Computer Science*, volume 8474 LNCS, pages 306–311, 2014.

[8] Nelishia Pillay. Developing Intelligent Programming Tutors for Novice Programmers. *inroads – The SIGCSE Bulletin*, 78(2), 2003.

[9] Kelly Rivers, Erik Harpstead, and Ken Koedinger. Learning Curve Analysis for Programming: Which Concepts do Students Struggle With? *Proceedings of the 12th International Computing Education Research Conference*, pages 143–151, 2016.

[10] John Stamper, Michael Eagle, Tiffany Barnes, and Marvin Croy. Experimental Evaluation of Automatic Hint Generation for a Logic Tutor. *International Journal of Artificial Intelligence in Education*, 22(1-2):3–17, 2013.

[11] Dinesha Weragama and Jim Reye. Analysing student programs in the PHP intelligent tutoring system. *International Journal of Artificial Intelligence in Education*, 24(2):162–188, jun 2014.