Matsuda, N., Cohen, W. W., & Koedinger, K. R. (2005). Building Cognitive Tutors with Programming by Demonstration. In S. Kramer & B. Pfahringer (Eds.), Technical report: TUM-I0510 (Proceedings of the International Conference on Inductive Logic Programming) (pp. 41-46): Institut fur Informatik, Technische Universitat Munchen.

Building Cognitive Tutors with Programming by Demonstration

Noboru Matsuda, William W. Cohen, Kenneth R. Koedinger

School of Computer Science, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh PA 15213
{mazda,wcohen,koedinger}@cs.cmu.edu

Abstract: The aim of this study is to incorporate the technique of programming by demonstration (PBD) into an authoring tool for Cognitive Tutors. The primary motivation of using PBD is to facilitate the authoring of Cognitive Tutors by educators, rather than AI programmers. That is, instead of asking authors to build a cognitive model representing a task to be taught, a machine-learning agent – called the *Simulated Student* – observes the author performing the target task and induces production rules that replicate the author's performance. FOIL is used to learn conditions appearing in the production rules. An evaluation in an example domain of algebra equation solving shows that observing 10 problems solved in 44 steps induced 9 correct and 1 wrong production rules. Two of the correctly induced rules were overly general hence produced redundant solutions.

1 Introduction

This study considers the application of *programming by demonstration* (PBD) to an unusual task domain: constructing *intelligent tutoring systems,* or *Cognitive Tutors.* Cognitive Tutors are known to be an effective means of tutoring students in various topics including algebra, chemistry, and physics [1]. However, building a Cognitive Tutor is difficult, as it requires knowledge of the subject matter, a good understanding of the prior abilities of the students who will use the system, and extensive programming skills. Our goal is to allow educators – i.e., people with knowledge of the subject matter and students' abilities, but little programming skills – to build Cognitive Tutors. To accomplish this, we wish to construct a system in which an author can construct a GUI for a Cognitive Tutor, and then use this GUI to present examples of how the human students should solve problems. A PBD learning system, called the *Simulated Student*, will then generalize these examples and build a set of production rules for solving problems in the task domain.

It is clearly desirable for the Simulated Student to learn a cognitive model that can be easily communicated with the authors. It is also clearly desirable for the Simulated Student to learn the "correct" generalizations (the ones intended by the authors) as quickly as possible. Less obviously, it is also useful for the Simulated Student to produce generalizations that are incorrect, but consistent with those that a human student might produce: such incorrect but *plausible* generalizations are often incorporated in the Cognitive Tutors to model possible human errors.

There have been a number of studies reported so far to integrate PBD into an authoring tool to build Cognitive Tutors. Jarvis *et al* [2] have successfully identified working memory elements and a sequence of operators that must be involved in a

production rule, but they have not addressed the issue of extracting conditions for the production rules to be fired, hence their production rules tended to be overly general. Blessing developed Demonstr8 [3] that also induces working memory elements and a sequence of operators for production rules from authors' demonstrations. It also has a tool for the authors to *manually* specify conditions of firing rules that must be embedded into the condition part. The interface to specify conditions apparently include predefined predicate symbols hard-coded into Demonstr8 hence the flexibility for authors to add new conditions is unclear.

In the current study all necessary conditions of firing rules are learned using FOIL [4] and embedded into production rules. Simulated Students are modular in both condition extraction and operator synthesis hence the background knowledge can be added and deleted easily. The resulted production rules would be just as good in quality as the ones hand-written by expert cognitive scientists, who are skillful in both cognitive task analysis and AI-programming.

2 Overview of Cognitive Tutors

2.1 Example domain: Algebra Equation Solving

As an example domain, we use algebra equation solving. **Fig. 1** shows an example tutor interface. It consists of two tables; one for left-hand side of the equation and the other one for right-hand side.

An equation is supposed to be solved by filling a cell on the left- or right-hand side alternatively, one at a time, from top to bottom without skipping any cells. At the moment shown in **Fig. 1**, the term "4x" on the left-hand side has been just entered. The next desirable action is then to enter the term "2x+5+2" in the empty cell on the right hand side immediately below "2x+5." These two steps, entering "4x" and "2x+5+2," together complete a higher-level goal in transposing the term "-2" from one side of the equation to the other. In the following sections, the first step that enters "4x" is refereed to as `trans-lr-lhs`, and the second step to tern "2x+5+2" is called `trans-lr-rhs`.

2.2 Model Tracing

One of the distinguished features of Cognitive Tutors is the way it recognizes and

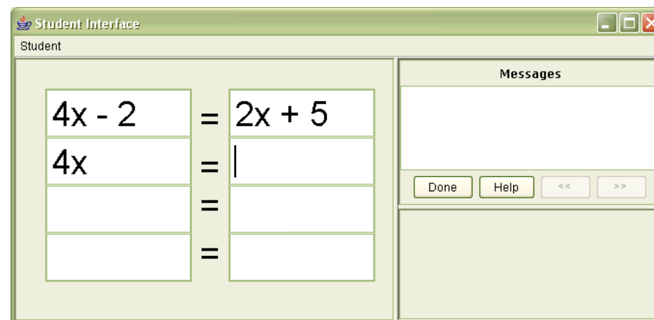

Fig. 1: Example of a Cognitive Tutor.

assesses students competent on each of the cognitive skills. For every problem-solving steps performed by a student, the Cognitive Tutor provides *flagged feedback*, which merely tells the student what he/she just did is correct or wrong. The judgment is based on a *cognitive model* that represents individual cognitive skills required to perform a target task. Given a cognitive model, one can generate a *solution graph* representing all possible solutions derived from the given cognitive model as a directed graph. Nodes in the graph represent an intermediate state reflecting a particular status of the GUI. Links in the graph represent a single production rule application.

The Cognitive Tutors monitor students' problem-solving activities by using a solution graph. For every action a student makes, the tutor attempts to identify a node in the solution graph that has the same GUI status as the one the student has just reached, and verify if it is an immediate successor of the previous state. If such state is successfully identified, then the student's action is considered to be correct, otherwise wrong.

2.3 Production Rules as a Cognitive Model

A production rule defines ways to manipulate objects (i.e., text fields, buttons, etc.) representing on the GUI. Each of those elements has a corresponding *working memory element* (WME) appearing in production rules.

A production rule consists of three major components: *WME-path* and *feature tests* in a condition part (or the left-hand side, LHS), and a set of *operators* in an action part (or the right-hand side, RHS).

A WME-path is a chain of WMEs from the WME representing a problem to a certain WME. A WME is structured like a frame. That is, it has *slots* and *slot-values*. For example, in the Equation Tutor, the first element in the `interface-elements` slot of the problem WME is a table that corresponds to the left table in the GUI shown in **Fig. 1**. The first element of the `column` slot of the table WME is a column WME. Finally, the cells in the column WME are associated to the `cell` slot, each of which corresponds to a cell in the left hand side in the tutor interface shown in **Fig. 1**.

A feature test specifies a condition in LHS as a relation that must be held among one or more of the WMEs.

3 Next Generation Authoring

3.1 Interaction between Authors and Simulated Students

Authors first build a GUI for their desired tutor, like a one shown in **Fig. 1**. To do this, they use the Cognitive Tutor Authoring Tools [5], which basically is a collection of tools, including a GUI builder, to build a Cognitive Tutor.

Next, authors need to specify all *predicate symbols* and *operator symbols* appearing in production rules. A predicate symbol represents a test for a specific feature. An operator takes one or more arguments and returns a single value. Both predicate symbols and operators are task dependent. They may be written by advanced authors.

The authors then use the GUI and solve a number of problems just in a way that the human students are supposed to perform. Those demonstrations would then be fed to the Simulated Student to induce production rules that are sufficient to replicate the demonstrations. Each step of problem solving must be annotated in such a way that the steps that can be done with the same production rules have the same name. Also,

authors are required to specify all GUI elements involved in a production rule. Since every single GUI element is associated with a unique WME, this step is essentially identifying the WMEs that appear in a production rule. These GUI elements (or WMEs) are called the *focus of attention*. In the case of `trans-lr-lhs` shown in **Fig. 1**, the focus of attention consists of the top and the 2nd cells on the left-hand side and the top cell of right-hand side.

Each time the author demonstrates a step, the Simulated Student induces production rules with the learning technique described in Section 3.2. The resulted production rules are then loaded to the Cognitive Tutor with the GUI component.

After authors solve a number of problems, the Cognitive Tutor may be ready to run, that is, the induced production rules are capable of solving problems correctly. To test the production rules, authors enter a new problem to the Cognitive Tutor, and let the tutor solve it. When the tutor shows an incorrect or undesired performance, authors provide a feedback by first clicking a [Wrong] button and then entering a correct value into a correct place. This feedback then triggers a refinement of the incorrect production rule.

Lastly and optionally, authors may directly modify production rules to obtain a desired set of production rules without providing more problems to trigger further refinement of irrelevant production rules.

3.2 Architecture of Simulated Students

The Simulated Student applies three different techniques for three major components of a production rule mentioned in Section 2.2: WME paths, feature tests, and operators. Given focus of attention specified by the author, searching WME paths and an operator sequence can be done by a brute-force search. To search a shortest operator sequence, the Simulated Student utilizes iterative-deepening depth first search.

Brute-force searching for feature tests is computationally very expensive as they involve relationships between WME elements. We use FOIL [4] for this task. Each time a problem-solving step is demonstrated, the Simulated Student generates input data for FOIL that is a collection of *positive* and *negative* examples for applicability of each production rule; a step demonstrated becomes a positive example for a production rule corresponding to the step. At the same time, the step also becomes a negative example for all other production rules. Those data are accumulated over the different problems, thus the number of positive and negative examples is continuously increasing as more steps are demonstrated.

4 Evaluation

To evaluate a performance of the Simulated Student, we have conducted an evaluation with algebra equation as an example subject domain.

For the sake of efficiency, the evaluation was done in such a way that instructions were provided with a text file. The output from the Simulated Student (i.e., production rules) was manually examined. The evaluation was run on a PC with Pentium IV 3.4GHz processor with 1GB RAM. The Simulated Student was written in Java.

We used 8 feature predicates and 13 operators as background knowledge. In total 44 steps were demonstrated to solve the 10 problems shown in **Fig. 2**. Those problems were solved by 10 different rules (shown in the top row in **Fig. 2**). In other

words, the Simulated Student induced 10 production rules from 44 demonstrated steps.

A cell with the letter “C” shows that the corresponding production rule was a correct generalization of the demonstrations. The cells with “P” or “W” show that the Simulated Student induced a production rule that was overly general. The difference between “P” and “W” is that the former (*plausible*) production rule yields a correct performance, but such application is strategically not optimal (i.e., yielding redundant steps), whereas the application of latter (*wrong*) production rule might lead to a wrong result.

An example of incorrectly generalized rule (a “W” in **Fig. 2**) is `trans-lr-lhs` learned from Problem 4 through 9 where the RHS operator says “write the *first variable term* into the target cell.” While this rule correctly produces “3x=4–2” from “3x+2=4,” given that `trans-lr-rhs` is correctly induced, it incorrectly produces “3x=4–2” from “3x+2x+2=4.”

An example of plausible over-generalization (“P”) is a rule `div-rhs` learned from Problem 4 and 5 where the feature tests in LHS say “apply this rule when LHS has a coefficient.” The RHS operators of this rule was correct hence the application of this rule always generates correct result, but since the conditions in LHS is still weak (i.e., overly generalized) this rule could apply to “3x+4=2x+5,” which is not a recommended strategy.

A shaded cell in **Fig. 2** shows that a corresponding production rule appeared in the demonstration. It must be emphasized that, as described in a previous section, a demonstration on a particular problem-solving step is not only used as a *positive* example for the specified production rule, but also serves as a *negative* example for the other production rules. Thus, a quality of production rule might also improve with a demonstration that does not involve that rule. See, for example, the rule `div-lhs`. Its first feature tests were captured (though they were overly generalized) on the 4th problem that does not involve `div-lhs`. It must be also emphasized that blank cells do contribute for learning by serving as negative examples. Thus, `do-arith[metic]-lhs`, for example, had 3 positive examples (shaded ones) and 7 negative examples (the first 7 blank cells in the same column).

No	Problem	div-lhs	div-rhs	trans-lr-lhs	trans-lr-rhs	copy-lhs	do-arith-rhs	trans-rl-lhs	trans-rl-rhs	do-arith-lhs	copy-rhs
1	3x = 6	**P**	**P**								
2	2x = 4	**P**	**P**								
3	4x = 12	**P**	**P**								
4	x - 5 = 3	P	P	**W**	**W**	**P**	**C**				
5	x + 2 = 6	P	P	**W**	**W**	**P**	**C**				
6	2 = -3x + 11	**C**	**P**	**W**	**W**	**P**	**C**	**P**	**W**		
7	3x - 4 = 2	**C**	**P**	**W**	**W**	**P**	**C**	P	W		
8	2x + 3x = 3 + 7	**C**	**P**	W	W	**P**	**C**	P	W	**P**	**P**
9	3x = 2x + 4	C	C	W	W	C	C	**C**	**W**	**P**	**P**
10	3x - 3 = 2x + 5	C	C	**C**	**C**	C	**C**	**C**	**W**	**P**	P

Fig. 2: Results over sequential learning from demonstration.

5 Conclusion

We have shown an application of programming-by-demonstration to a novel task domain of building Cognitive Tutors. With the Simulated Student, educators who are not necessarily skillful cognitive scientists can build their own Cognitive Tutors without getting involved in heavy programming. Due to a mandatory demand of the task, the output of Simulated Student must be comprehensible for the authors. The preliminary evaluation showed that Simulated Student has effectively induced sufficient information to compose accurate production rules that are written in widely accepted programming language.

The Simulated Student might learn incorrect rules that a human student might also learn, although we did not observed such erroneous learning in the evaluation. Further study is needed to investigate the amount of demonstrations necessary to learn stable production rules as well as the effect of different curriculum on the quality of learning. The current evaluation was conducted with only a small number of demonstrations and a small set of background knowledge (i.e., feature predicates and operators). It must be scaled up so that cross validation can be taken place to have better understanding of both the quantitative and qualitative aspects of programming by demonstration.

Acknowledgement

This research was supported by the Pittsburgh Science of Learning Center funded by National Science Foundation award No. SBE-0354420.

Reference

[1] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier, "Cognitive tutors: Lessons learned," *Journal of the Learning Sciences*, vol. 4, pp. 167-207, 1995.

[2] M. P. Jarvis, G. Nuzzo-Jones, and N. T. Heffernan, "Applying Machine Learning Techniques to Rule Generation in Intelligent Tutoring Systems," in *Proceedings of the International Conference on Intelligent Tutoring Systems*, J. C. Lester, Ed. Heidelberg, Berlin: Springer, 2004, pp. 541-553.

[3] S. B. Blessing, "A Programming by Demonstration Authoring Tool for Model-Tracing Tutors," *International Journal of Artificial Intelligence in Education*, vol. 8, pp. 233-261, 1997.

[4] J. R. Quinlan, "Learning Logical Definitions from Relations," *Machine Learning*, vol. 5, pp. 239-266, 1990.

[5] K. R. Koedinger, V. A. W. M. M. Aleven, and N. Heffernan, "Toward a Rapid Development Environment for Cognitive Tutors," in *Proceedigns of the International Conference on Artificial Intelligence in Education*, U. Hoppe, F. Verdejo, and J. Kay, Eds. Amsterdam: IOS Press, 2003, pp. 455-457.