Matsuda, N., Cohen, W. W., & Koedinger, K. R. (2005). Applying Programming by Demonstration in an Intelligent Authoring Tool for Cognitive Tutors. In AAAI Workshop on Human Comprehensible Machine Learning (Technical Report WS-05-04) (pp. 1-8). Menlo Park, CA: AAAI association.

Applying Programming by Demonstration in an Intelligent Authoring Tool for Cognitive Tutors

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

We are building an intelligent authoring tool for Cognitive Tutors, a highly successful form of computer-based tutoring. The primary target users (the authors) are educators who are not familiar with cognitive task analysis and AI programming, which are essential tasks in building Cognitive Tutors. Instead of asking authors to write a cognitive model by hand, a Simulated Student embedded in the authoring tool lets an author demonstrate how to perform the tasks in the subject domain, for instance, solving an algebra equation. The Simulated Student observes an author's demonstration and induces a set of production rules that replicate the demonstrated performances. Correct production rules, as well as production rules that are incorrect but similar to those a human student might produce, can be directly embedded in the Cognitive Tutor. We give a preliminary evaluation of an implemented Simulated Students based on inductive logic programming and path-finding.

Introduction

In the vast majority of practical settings, the "trustworthiness" of a learned hypothesis is evaluated by testing it on sample data. This sort of "test-based" evaluation is quantitative, can be easily automated, and can be rigorously justified statistically under certain assumptions, e.g., a stationary distribution of problems.

Unfortunately, once these statistical assumptions are relaxed, a test-based evaluation is not sufficient to ensure "trust." (As a simple example, a mail filter that had good performance over the last year might fail miserably after one switches jobs.) In settings in which learned hypotheses are to be exercised in novel ways, or used to make predictions outside the distribution of original problems, other schemes must be used to ensure that the output of a learning system can be "trusted." The most common of these other schemes is to manually inspect the *syntactic form* of hypothesis output by the learner for correctness. This is only possible, of course, if this syntactic form is easily "comprehensible" to a human reader. An alternative reason to "trust" a particular hypothesis is because of "trust" in the learning system that produced that hypothesis. For instance, a hypothesis produced by a learning system might be trusted, in this sense, if the learner is known to produce "good" hypotheses 90% of the time in similar circumstances.

In this paper we describe an application of machine learning where a single learner is used multiple times by the same user, where the learned hypothesis will be used in novel ways to make predictions, and where it is critical that the user trust the hypotheses of the learner. Interestingly, in our setting, "trusting" a hypothesis is *not* the same as verifying its correctness—there are some hypotheses which are not correct, but still potentially useful, if they can be properly understood by the end user.

Specifically, we are using machine learning methods as one component of an intelligent authoring system for Cognitive Tutors. In this setting, it is desirable that the machine learner produces generalizations similar to those that might be produced by the human students that will ultimately be instructed by the Cognitive Tutor. In short, we would like to produce hypotheses that have human-like performance, when used in a Cognitive Tutor; however, the learner that produces these hypotheses may itself not have human-like behavior.

In more detail, Cognitive Tutors are intelligent tutoring systems based on a rule-based cognitive model of the subject domain being taught to students [1]. For example, for a Cognitive Tutor that teaches algebraic equation (like the one shown in Figure 1), the cognitive model is a set of production rules that can solve equations in ways that students do. Cognitive Tutors are effective in increasing student learning rate over other alternative teaching methods, and they are in regular use in over 1,800 high schools [2]. However, they are expensive to construct, as building a cognitive model requires time-consuming cognitive task analysis, as well as AI programming skills. Our goal is to make it possible for educators (e.g., school teachers) to build their own Cognitive Tutors without these special skills to build a tutor, using only intuitive graphical interfaces, and their familiarity with the subject matter and how human students learn. The basic idea is that instead of writing a code for the cognitive model, the educators demonstrate how to perform the subject task on sample problems. A machine learning agent, called a *Simulated Student*, observes those demonstrations and induces a cognitive model, encoded as a set of production rules.

The machine-learned production rules generalize the observed actions of the educator on the sample problems. If these generalizations are correct – i.e., are correct implementations of the actual task that is being taught – then they can be added to the cognitive model. If the generalizations are incorrect but "plausible," then they can be added to the cognitive model as *mal-rules*. Mal-rules are plausible in a sense that they represent generalizations that a human student might make from the same examples. If the generalizations are implausible – i.e., not human-like generalizations at all – then they must be discarded. Hence, the utility of the machine-learning system is related to the number of human-like generalizations it produces, rather than the number of correct generalizations.

For such a system to work, it is necessary that the output of the machine-learning system be *comprehensible* to the authors (i.e., users), in the usual sense, so that they can determine its correctness. To accomplish this goal, we integrate the Simulates Students to an existing authoring tool for Cognitive Tutors, called CTAT [3] so that the authors can easily demonstrate tasks and test hypothesis generated by letting Simulated Students to solve novel problems. To build the inference engine of the Simulated Students, we use a combination of techniques including inductive logic programming methods to generalize the educator's actions. The output is encoded in Jess, a well-known production rule description language [4]. This allows advanced users to manually modify the output if desired.

In this paper, we first explain basic framework of Cognitive Tutors. We then introduce Simulated Student and show how they observe authors demonstration and induces a set of production rules. We also discuss a structure of an intelligent authoring tool as a product of integrating Simulated Students to CTAT. Finally, we show results from preliminary experiments in an algebra equation solving domain. Given 8 feature predicates and 13 operators as background knowledge, and providing demonstrations of 10 problems with 44 problem-solving steps, the Simulated Students correctly induces 7 production rules. Two of other 3 production rules were overly general but plausible.

Cognitive Tutors

This section provides a brief overview on Cognitive Tutors, especially the model-tracing technique. It then describes the structure of a production rule, the target language for Simulated Students' learning.

Overview of the Cognitive Tutor Architecture

The effectiveness of Cognitive Tutors has been reported in a number of places (for example, [2]). The most unique feature of the Cognitive Tutors is *model tracing*, which requires a cognitive model that represents every single *cognitive skills* involved in the target subject task. A unit of such skill is represented as a *production rule*, which generates a determinate performance for a specific problem state (e.g., to enter a particular value into a particular place).

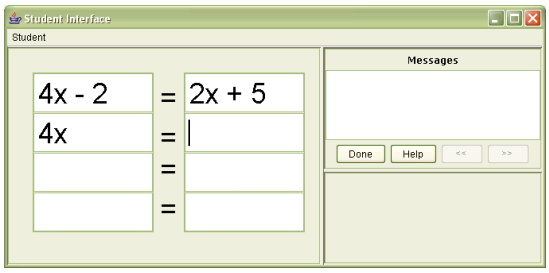

Figure 1: The Equation Tutor

Hence, the quality of instruction provided by Cognitive Tutors heavily relies on the quality of the production rules.

Students are supposed to learn each of those production rules. Namely, for a student to be considered that one has mastered a particular cognitive skill, he/she must show the same performance as the corresponding production rule does whenever that rule should be applied.

To assess students' skills, the Cognitive Tutor compares (i.e., *model traces*) student's performance against the model. When the student shows a performance that does not match any of the production rules, the tutor provides a negative feedback to the student. The cognitive model might involve not only correct production rules, but also incorrect or "buggy" production rules that represent students' typical errors. If student's performance matches with one of those buggy rules, then the tutor provides feedback specific to the error.

When the student gets stuck or does not know what to do, then the student can ask a hint. The tutor provides a hint message associated with a production rule that should be fired under the given state.

In sum, for model tracing to function properly, it is crucial that the cognitive model, which is represented as a set of production rules, is correct. The model might not only hold correct production rules, but also incorrect production rules.

Example of Cognitive Tutor: The Equation Tutor

As an example of the task domain, we use algebra equation solving. As shown in Figure 1, an interface of the Equation Tutor is as simple as just having one table with two columns each corresponds to the left- and right-hand side of equations. Here, an equation is supposed to be solved by filling a cell on the left- and right-hand side alternatively, one at a time, from top to bottom without skipping any cells. At the moment shown in Figure 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 transposition of the term "-2" from one side of the equation to the other. In the following sections, the first step is refereed to as "`trans-lr-lhs`," (an abbreviation for "transposition left to right on the left-hand side," see Table 2 for details) and the second step "`trans-lr-`

```
1  (defrule trans-lr-lhs

2    ?problem <- (MAIN::problem (interface-elements ?table ? ?))
3    ?table <- (MAIN::table (columns ?column-1 ?))
4    ?column-1 <- (MAIN::column (cells $?m208 ?cell-1 $?))
5    ?cell-1 <- (MAIN::cell (value ?lhs))

6    ?table <- (MAIN::table (columns ? ?column-2))
7    ?column-2 <- (MAIN::column (cells $?m220 ?cell-2 $?))
8    ?cell-2 <- (MAIN::cell (value ?val1))

9    ?column-1 <- (MAIN::column (cells $?m232 ?cell-3 $?))
10   ?cell-3 <- (MAIN::cell (name ?name198) (value ?new-lhs))

11   (test (consecutive-row ?cell-1 ?cell-3))
12   (test (same-column ?cell-1 ?cell-3))
13   (test (consecutive-row ?cell-2 ?cell-3))

14   (test (not (polynomial ?lhs))
15   (test (not (has-var-term ?rhs))

16   ?special-tutor-fact-correct <- (special-tutor-fact-correct)

17   =>

18   (bind ?new-lhs (remove-last-const-term ?lhs))

19   (modify ?cell-3 (value ?new-lhs))

20   (modify ?special-tutor-fact-correct
21      (selection ?name198)
22      (action UpdateTable)
23      (input ?new-lhs)
24      (hint-message
25       (construct-message
26        [ Try to translate a constant trem from ?lhs to ?rhs ]
27        [ What is the last constant term in ?lhs ]
28        [ Write ?new-lhs into the blank cell under ?lhs ] ) ) )
29 )
```

Figure 2: Example of production rule

`rhs`." A cell in the table where a new value is entered is called "*Selection*" and the value entered is called "*Input*." In the example shown in Figure 1, "4x" is the "Input," and the cell in the 2nd row on the left-hand side is the "Selection."

Production Rules

Figure 2 shows an example of a production rule. The lines from 2 to 16 compose a condition part (the left hand side, or LHS) and from 18 to 28 compose an action part (the right hand side, RHS).

There are two unique features in LHS in our production-rule description language that must be explained: *WME-path* and *feature tests*. The explanations follow.

A feature test specifies a condition in LHS as a relation that must be held among one or more working-memory elements (WME). There are two different types of feature tests: (1) a test for topological relation among WMEs (e.g., two table cells in the same column), and (2) a test for a semantic context specified by the WME values (e.g., a term shown in one cell is the sum of the terms in another cell). The former is called a *topological constraint* and the latter a *contents constraint*. In Figure 2, lines 11 through 13 show topological constraints whereas the lines 14 and 15 show contents constraints.

A WME-path is a chain of WMEs from the problem WME to a certain WME. For example, in Figure 2, the problem WME is shown on the line 2, and bound to the variable `?problem`. The first element in the `interface-elements` slot of the problem WME is a table that is bound to a variable `?table`. The first column of the table is a "column" WME. Finally, a cell in the column WME is bound to the variable `?cell-1`, which corresponds to any cell in the left hand side in the tutor interface shown in Figure 1. More precisely speaking, variables with a '`$`' mark at the beginning is a *multivariable*, which can be bound to a list of variables. Thus, the variable `?cell-2` on the line 7 can be any element in the `cells` slot.

There are two assumptions in our implementation of Cognitive Tutors: (1) all of the WMEs that are subject for feature tests must be represented in the graphical user interface of the tutor, and (2) all of the WMEs can be reached from the "problem" WME by following a slot value. The former ensures that all WMEs in the feature tests can be directly mentioned in the demonstration by the authors. The latter assumption guarantees that all WMEs in the feature tests can be uniquely identified as the terminal node of a WME-path.

Another unique feature in our production rule is the *selection-action-input* tuple (SAI tuple) appeared in RHS (lines 21, 22, and 23 in Figure 2). The SAI tuple represents an observable change in GUI made when the production rule is applied. An example of such change involves entering a text in a table cell, pressing a button, selecting a list, etc. When the tutor attempts to *model trace* students' performances, a problem-solving step made by a student is considered to be correct if the SAI tuple matches the student's performance.

Simulated Students

This section describes the overall architecture of Simulated Students and its integration to an existing authoring tool for Cognitive Tutors.

Next Generation Authoring: CTAT & Simulated Students

The Cognitive Tutor Authoring Tool (CTAT) [3] is an integrated development environment to facilitate authoring with Simulated Students. CTAT offers, among other features, a GUI builder that helps authors to build a GUI by basically dragging and dropping GUI elements such as buttons, tables, text fields, and so forth. As the first step of authoring, the authors build a graphical user interface (GUI) for their desired tutor using CTAT. An example interface shown in Figure 1 is indeed built with CTAT.

Once authors finish building a tutor's GUI, they then specify *predicate symbols* and *operator symbols* appearing in LHS and RHS of the production rules. Those symbols might be common in several closely related task domains (e.g., `coefficient(X,Y)` in algebra) hence defined a priori by the developers of Simulated Students. The authors can also define new symbols that are novel for a particular subject domain.

Given a GUI and a set of predicate and operator symbols, the authors now demonstrate the task. That is, they solve a number of problems through the GUI in a way that the human students are supposed to perform. A *problem-solving step* is segmented when the selection-action-input (SAI) tuple is observed.

For each problem-solving step, the authors must specify the following: (1) *focus of attention* that is a set of elements in GUI that has contributed to the step performed, and (2) a unique name of the step, which corresponds to the name of a production rule to be induced for the step. Individual problem-solving steps are recorded along with the SAI tuple, focus of attention, and the name of the step.

Each time a problem-solving step is demonstrated, Simulated Student attempts to model trace the step. The result of this attempt is visualized in CTAT so that the author can immediately know the correctness of production rules induced by Simulated Student by the current point. If a demonstrated step can not be model traced, then the Simulated Student attempts to "refine" the production rules by reading all demonstrations performed so far and re-compiling a whole set of production rules. The author is then asked to solve another problem.

That all of the problem-solving steps are successfully model traced and the author agrees with the result of the model trace implies that the current set of production rules are correctly generalized *with respect to demonstrations* performed so far, hence it is reasonable to stop demonstrating on new problems. At this point, the authors can either *test* the production rules, or *modify* them manually. To test the production rules, authors provide a new problem, and let the Simulated Students solve it. If any steps turned to be wrong (even if the model says it is correct), then the author provides a negative feedback (i.e., signals that it is wrong and provides a correct SAI tuple). The Simulated Students then re-induce production rules by merging the feedback to the previous demonstrations.

To modify production rules, which are represented in Jess language, the authors can use the Production Rule Editor embedded in CTAT.

Learning Techniques

As mentioned in a previous section, we have decomposed the task of learning production rules into three components: WME-paths, feature tests, and operators. The first two components are in LHS and the last one is in RHS. The Simulated Students induce each of these parts separately from authors' demonstrations.

Search for WME-paths. An algorithm for a search for working-memory elements (WMEs) thoroughly depends on the structure of the working memory. The current implementation of Cognitive Tutor assumes that all WMEs can be reached by following a path from the problem WME, and there must be no multiple paths for any of the WMEs. Given these restrictions, a search for WMEs appearing in a production rule is indeed a search for WME paths for each of those WMEs. Since there is no multiple paths in the tree, and there are at most finitely many WMEs, this search should be straightforward given that binding a list to a multivariable is taken care of. An example might best explain this issue.

The step `trans-lr-lhs` shown in Figure 2 can be used not only for an equation at the first row, but for any equations at any row. Hence, in general terms, the rule `trans-lr-lhs` should be read something like "given any non-empty cell in the first column (i.e., the column representing the left-hand side) and a non-empty cell on the same row in the second column (i.e., the right-hand side), drop the last term of the polynomial expression in the first cell, and enter the result in a cell immediately below the first cell." Notice that the first cell can be any cell on the left-hand side; this is where the multivariable takes place. In Figure 2, the "any non-empty cell in the first column" corresponds to the variable `?cell-1` bound at the line 4 as a value of the cells slot. Since the variable `?cell-1` can be any member in the list in the cells slot, it is preceded by a multivariable `$?m208`, which would be bound to a list of arbitrary number of elements.

In sum, there are two patterns to bind elements in a list: (1) Binding an element at its absolute location with a (non-multi) variable following absolute number of non-multivariables. An example of this pattern can be found at the first line in Figure 2 where the variable `?table` is bound as the first element in a list with three elements. (2) Binding an element as an arbitrary element in the list with a (non-multi) variable following a multivariable. An example of this pattern can be found at the third line in Figure 2 where the variable `?cell-1` can be any element (a cell) in the cells slot. In terms of the "generality" of the representation that defines a version space for the search for working memory elements, the latter is more general than the former.

Since Authors specifies all WMEs appearing in each production rule, the task of searching WMEs for a particular production rule is really a search for the least general WME paths for each of those specified WMEs. This can be done by a brute-force depth-first search given that search space is structured as a version space where the least general state contains no multivariables to bind a list element whereas the most general state uses multivariables for all list bindings.

Search for feature tests. Simulated Students utilize FOIL [5] to induce feature tests. They generate an input data to FOIL for each of the production rule, which consists of the specification of *types*, a *target relation*, and *supporting relations*.[1] Figure 3 shows an example of a FOIL input data where these three categories of information are shown as "Types," "Training data," and "Background knowledge."

The figure shows a FOIL input for the production rule `trans-lr-lhs`. The target relation is specified as a literal

[1] In the original literature [5], both the target relation and supporting relations are simply called "relations." However, for the sake of explanation, we call relations that are not a target relation the supporting relations.

Types:

V2: x-5, x+2, 3x+2.
V1: x, x-5, x+2, 3x, 3x+2.
V0: x, 3, 6, 3x, 11.
T1: 3, 6, 11, -3x+11, 4, 12, 9, 3+5, 6-2, 11-2.
V4: 1, 3.
T0: x-5, x+2, 3x+2, 2, 3x, 2x, 4x, x.
V3: x, 3x.

Training data:

+trans-lr-lhs(x-5, 3)
+trans-lr-lhs(x+2, 6)
+trans-lr-lhs(3x+2, 11)
-trans-lr-lhs(2, -3x+11)
-trans-lr-lhs(3x, 6)
-trans-lr-lhs(2x, 4)
-trans-lr-lhs(4x, 12)
-trans-lr-lhs(3x, 9)
-trans-lr-lhs(x, 3+5)
-trans-lr-lhs(x, 6-2)
-trans-lr-lhs(3x, 11-2)

Background knowledge:

+Monomial(x)
+Monomial(3)
+Monomial(6)
+Monomial(3x)
+Monomial(11)
… … …

+VarTerm(x)
+VarTerm(x-5)
+VarTerm(x+2)
+VarTerm(3x)
+VarTerm(3x+2)
… … …

Figure 3: An example of the input to FOIL

`trans-lr-lhs` with two arguments. The number of focus of attention for this production rule is three, but that includes the WME where the new value is entered. Since the third WME (i.e., the "Selection" WME) is empty at the time when LHS is evaluated, it has no information for a search for distinctive features. Thus, it is eliminated from the target relation.

To compile an input data for a particular production rule, Simulated Student performs the following procedure each time a problem-solving step is demonstrated:

1. Let *I* be the demonstration. Let *R* be a name of the problem-solving step annotated for *I*. Let *V* be a set of values specified as a focus of attention for *I* except the "Input," that is, the value to be entered in the selection WME. Let *D* be a partial input data that has been compiled for *R* so far (if any). If such *D* does not exist, then create an empty *D* for *R*.
2. For each feature predicate *p*, do following steps.
   2.1. Let *n* be the number of arguments of *p*. Let *C* be all possible *n*-permutations of values specified as focus of attention; $C = \{(v_1, \ldots, v_n) \mid v_i \in V\}$.
   2.2. For each *c* in *C*, if $p(c)$ holds, then add *c* to a positive tuple of *p* for *D*. Otherwise add *c* to a negative tuple.
3. Add all elements in *V* as a positive tuple of the target relation for *D*.
4. For each production rule *R'* that is not *R*, add all elements in *V* as a negative tuple of the target relation for *D*', an input data for *R*', iff the number of arguments for the target relation agrees with the number of elements in *V*.

Search for operators. Recall that Authors specify the value of "Input" and other values that were involved in the problem-solving step (i.e., the focus of attention) as a mandatory part of demonstration.

Given these, the operators in the right-hand side can be searched as a sequence of operator applications that yields the "Input" value from the other values. The current version of Simulated Student simply applies the iterative-deepening depth-first search to find a shortest operator sequence. Starting from no operator, the search space is explored by adding one operator at the time until either it reaches a depth limit or the combination of the operator generates the desired "Input" value. A similar technique is applied in Richards *et al* [6], but their search is bidirectional whereas the proposed search is more straightforward.

Related Studies

FOIL is one of the most efficient and robust learning tools that has been widely used in very many studies. Lau *et al*, for example, used FOIL as their earlier work [7]. There are several variations of FOIL and improvement (see, for example, [8]).

Programming by Demonstration (PbyD) has been proven to be effective for novice programmers to improve their productivities. Lau *et al*, for example, applied PbyD technique for editing macros used in a text editor [9]. It has been claimed that PbyD learns target language with a small number of examples.

There has been a number of studies reported so far to integrate PbyD into authoring tool to build intelligent tutors. Jarvis *et al* built a machine learning agent for Cognitive Tutor [10]. They have successfully identified WME paths and a sequence of operators, but they have not addressed the issue of feature extraction on the condition part hence their production rules tended to be overly generalized. Blessing also applied a PbyD technique to authoring tool for Cognitive Tutors, called Demonstr8 [11]. Demonstr8 can induce WME paths and the action part from authors' demonstrations. It also has a tool for authors to *manually* specify feature tests that must be embedded into the condition part. Such kinds of feature tests were apparently predefined and hard coded into Demonstr8 hence the flexibility for authors to add new features were unclear.

In the current study all necessary feature tests are automatically identified and embedded into production rules. Simulated Students are modular in both feature tests and RHS operators hence new predicate and operator symbols can be added and deleted easily.

Evaluation

To evaluate efficiency and usefulness of the Simulated Students, we have conducted an evaluation with algebra equation as an example subject domain.

Since we have yet to integrate the Simulated Student into CTAT, the evaluation was done within a quasi-authoring environment where we first built an interface for the Equation Tutor and then provide demonstrations with a text file. The output (i.e., a set of production rules) from the Simulated Student was manually verified, instead of load-

ing it to Equation Tutor and getting model traced on new problems.

The evaluation was run on a PC with Pentium IV 3.4GHz processor with 1GB RAM. The Simulated Student was written in Java. So far, we have finished implementing the main inference engine for the Simulated Students. Currently, they do not output induced production rules in the target language (i.e., Jess), but they do indeed induce all information to compile production rules, that is, WME paths, feature tests, and RHS operators. Thus, in the following sections, the evaluation of the quality of "production rule" was done manually. FOIL was invoked as an external process with a file based communication.

Methods and Materials

We used 8 feature predicates and 13 operators as background knowledge. Total of 44 steps were demonstrated to solve 10 problems shown in Table 1. Those problems were solved by 10 different rules (shown in Table 2). In other words, the Simulated Students were supposed to induce 10 production rules from 44 demonstrated steps.

Results and Discussions

7 out of 10 production rules were correctly induced after 10 problems were demonstrated. Two of 3 overly generalized rules were plausible.

To have better understanding on how the learning occurred, we conducted an item analysis. Each time a problem is solved by an author, we let the Simulated Student induce production rules with *all demonstrations performed by that point*. We then examine, for each production rule, how each part of the production rule (i.e., WME-path, feature test, and operators) was induced.

Surprisingly, Simulated Students *always* induced correct WME-path. That is, in the current experiment, only a single demonstration is enough to correctly identify WME-path.

What makes learning so complicated was acquisition of feature tests and operators. Figure 4 shows a chronological improvement in learning each production rule. The figure shows problems in the order that they were demonstrated. The top row in the remaining columns shows names of the production rules.

Table 1: Problems used for the evaluation

3x = 6, 2x = 4, 4x = 12,
x – 5 = 3, x + 2 = 6,
2 = –3x + 11,
3x – 4 = 2,
2x + 3x = 3 + 7,
3x = 2x + 4,
3x – 3 = 2x + 5

A shaded cell 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 it also served as a *negative* example for the other production rules. Thus, a quality of production rule might improve on 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).

A cell with "C" shows that the corresponding production rule was a correct generalization of the demonstrations. The cells with "P" and "W" both shows that the Simulated Students induced a production rule, but it was overly generalized. The difference between "P" and "W" is that the application of 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.

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

Figure 4: Results over sequential learning from demonstration. W: The application of the rule might generate a wrong result. P: The rule is overly generalized hence the application of the rule might not be strategically optimal but a result of the rule application is correct. C: The rule is correct. A shaded cell shows that the corresponding production rule appears in the demonstration.

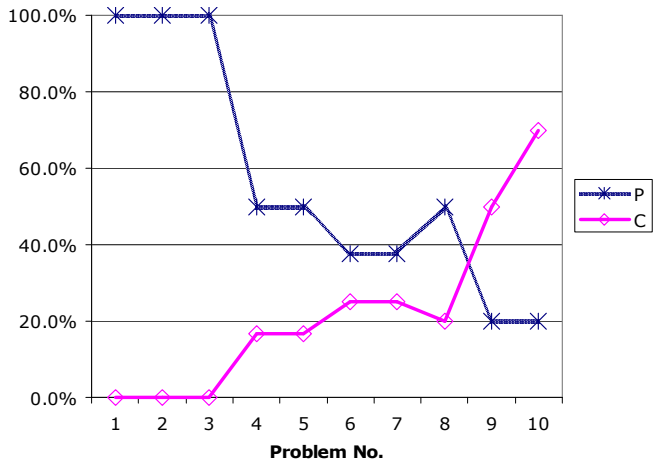

Figure 5: % correct and plausible generalizations

An example of incorrectly generalized rule (a "W" in Figure 4) is `trans-lr-lhs` learned from problem #4 through #9 where the RHS operator says "write the *first variable term* into the target ("Selection") 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 through #5 where the feature tests in LHS say "apply this rule when LHS is a monomial and 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 = 2x + 5," which is not a recommended strategy.

There are several interesting observations read from Figure 4:

1. All but one (`do-arith[metic]-rhs`) production rules required a few (3 to 8) examples to make a correct generalization.
2. All but one (`do-arith[metic]-rhs`) production rules required both positive and negative examples for appropriate generalization.
3. Learning was monotonic, that is, once a correct generalization is made, it is never incorrectly modified. Figure 5 shows ratio of correct and plausible generalizations to the number of production rules to be learned for each demonstration.
4. There were a few incorrect generalizations observed that could have been learned by human students as well (plausibly incorrect generalizations).

These results show that the Simulated Students learn cognitive skills by observing teachers demonstrations in a similar way human students do, namely, they make overly generalized rules that are sometimes wrong and sometimes plausible.

A potentially interesting issue is to explore the impact of *curriculum* on learning. Namely, what if we change the order of problems to demonstrate? In the current experiment, we started to demonstrate very simple problems that can be solved in a single line (i.e., the solution that immediately follows the initial equation). We then gradually increased a level of complexity of the problem by introducing a few new production rules. The learning results may well be affected by the choice and sequencing of problems [12] and we plan future experiments to test alternative curriculum sequences and whether they lead to or prevent errors observed by students.

Conclusion

Although the size of experiment was rather small (10 production rules on 10 problems), the experiment showed a potential usefulness of the Simulated Students. That most production rules were correctly generalized in at most 8 demonstrations was rather surprising result. The simulated Students also showed a wrong but plausible generalization that human students might have learned as well. The major goal of the current study – learning human *comprehensible* cognitive models in a *human-like* way – has been tested affirmatively.

The current study has potential benefits not only of developing an intelligent authoring tool to educators, but also a test bed to explore principle in human learning from worked-out examples and problem solving (see, for example, [13]). Observing how incorrect but plausible generalizations would be made might be one of the most interesting issues.

We have yet to complete the integration of Simulated Students into CTAT so that an evaluation of the overall authoring environment can be done in an authentic situation with real human educators.

Acknowledgement

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

References

1. Anderson, J.R., et al., *Cognitive modeling and intelligent tutoring.* Artificial Intelligence, 1990. **42**(1): p. 7-49.
2. Anderson, J.R., et al., *Cognitive tutors: Lessons learned.* Journal of the Learning Sciences, 1995. **4**(2): p. 167-207.
3. Koedinger, K.R., 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, Editors. 2003, IOS Press: Amsterdam. p. 455-457.
4. Friedman-Hill, E., *Jess in Action: Java Rule-based Systems*. 2003, Greenwich, CT: Manning.
5. Quinlan, J.R., *Learning Logical Definitions from Relations.* Machine Learning, 1990. **5**(3): p. 239-266.
6. Richards, B.L. and R.J. Mooney, *Learning Relations by Pathfinding*, in *Proceedings of the Tenth National Conference on Artificial Intelligence*. 1992, AAAI Press: Menlo Park, CA. p. 50-55.
7. Lau, T.A. and D.S. Weld, *Programming by demonstration: an inductive learning formulation*, in *Proceedings of the 4th international conference on Intelligent user interfaces*. 1998, ACM Press: New York, NY. p. 145-152.
8. Cohen, W.W., *Grammatically biased learning: Learning logic programs using an explicit antecedent description language.* Artificial Intelligence, 1994. **68**(2): p. 303-366.
9. Lau, T., et al., *Programming by Demonstration Using Version Space Algebra.* Machine Learning, 2003. **53**(1-2): p. 111-156.
10. Jarvis, M.P., 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, Editor. 2004, Springer: Heidelberg, Berlin. p. 541-553.
11. Blessing, S.B., *A Programming by Demonstration Authoring Tool for Model-Tracing Tutors.* International Journal of Artificial Intelligence in Education, 1997. **8**: p. 233-261.
12. Zhu, X. and H.A. Simon, *Learning mathematics from examples and by doing.* Cognition and Instruction, 1987. **4**(3): p. 137-166.
13. Renkl, A., *Learning from worked-out examples: A study on individual differences.* Cognitive Science, 1997. **21**(1): p. 1-29.

Table 2: Problem-solving steps demonstrated

Name	Description
`copy-lhs`	Copy an expression in an immediately above line on the left-hand side (LHS)
`copy-rhs`	Copy an expression in an immediately above line on the right-hand side (RHS)
`trans-lr-lhs`	Given a polynomial expression on the LHS with a constant term(s), remove the constant term and enter the resulting expression into the LHS on the following line
`trans-lr-rhs`	Given that `trans-lr-lhs` has been taken place in the previous step, add the term removed to the expression on the RHS
`trans-rl-lhs`	Similar to `trans-lr-lhs`, but transfer a variable term from the RHS to LHS
`trans-rl-rhs`	Similar to `trans-lr-rhs`, but transfer a variable term from the RHS to LHS
`div-lhs`	Given a variable term on the LHS, and a constant term on the RHS, divide the term on the LHS with its coefficient
`div-rhs`	Given that `div-lhs` has been taken place in the previous step, divide the constant term on the RHS with the same coefficient
`do-arithmetic-lhs`	Given a polynomial expression on the LHS that does not contain any variable terms, simplify it by adding all terms
`do-arithmetic-rhs`	Given a polynomial expression on the RHS that does not contain any variable terms, simplify it by adding all terms