Applying Machine Learning to Cognitive Modeling for Cognitive Tutors

Noboru Matsuda[1], William W. Cohen[2], Jonathan Sewall[1], and Kenneth R. Koedinger[1]

July 2006
CMU-ML-06-105

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213

Abstract: The aim of this study is to build an intelligent authoring environment for Cognitive Tutors in which the author need not manually write a cognitive model. Writing a cognitive model usually requires days of programming and testing even for a well-trained cognitive scientist. To achieve our goal, we have built a machine learning agent – called a Simulated Student – that automatically generates a cognitive model from sample solutions demonstrated by the human domain expert (i.e., the author). This paper studies the effectiveness and generality of the Simulated Student. The major findings include (1) that the order of training problems does not affect a quality of the cognitive model at the end of the training session, (2) that ambiguities in the interpretation of demonstrations might hinder machine learning, and (3) that more detailed demonstration can both avoid difficulties with ambiguity and prevent search complexity from growing to impractical levels.

[1] Human Computer Interaction Institute, Carnegie Mellon University, PA, USA, {mazda, sewall, koedinger}@cs.cmu.edu}
[2] Machine-Learning Department, Carnegie Mellon University, PA, USA, wcohen@cs.cmu.edu

This material is based upon a work supported by National Science Foundation award number REC-0537198

Key words: Programming by Demonstration, Inductive Logic Programming, Cognitive Modeling, Cognitive Tutor, Authoring

1. Introduction

This paper describes how a machine learning technique, namely programming by demonstration, can help build a cognitive model for Cognitive Tutors.

Cognitive Tutors are known to be very effective, but they require the author to build a cognitive model that can generate the cognitive steps in the task to be taught. Building a cognitive model requires detailed analysis of the domain principles (cognitive task analysis, e.g.) as well as significant AI programming (familiarity with production systems, e.g.). Furthermore, it takes hundreds of hours even for a skilled expert to build and test a cognitive model (Murray, 1999).

The Cognitive Tutor Authoring Tools (CTAT) suite aims to enable non-programmers to create problem-specific Cognitive Tutors (Koedinger *et al.*, 2003) simply by demonstration. A problem-specific model records the steps demonstrated by the author to solve a particular instance of the task (see section 3). This "model" does not encode general domain principles and hence usually cannot solve problems other than the one demonstrated. This limitation becomes critical when the Cognitive Tutor must be applied to a significant number of exercises. This challenge can be resolved by a domain cognitive model, but providing an aid for non-programmers to build such a model is much more challenging.

We can assume that our target users (the potential authors) have no difficulty solving problems in the target domain. Thus, our proposed solution is to apply a machine learning technique that automatically learns a cognitive model by observing authors solve problems; this is *programming by demonstration* (Cypher, 1993). We call our machine-learning agent a *Simulated Student* in an analogy to human learning: the Simulated Student observes a teacher's (i.e., the author's) problem-solving demonstrations and learns a set of cognitive skills to reproduce such demonstrations.

This paper first discusses research questions on integrating the Simulated Student as a building block of an intelligent authoring tool. We then provide a brief overview on Cognitive Tutors and the authoring tools (CTAT), followed by a general description of Simulated Student. In section 5, we show several evaluation studies on the usefulness and generality of Simulated Students and discuss lessons learned.

2. Research Questions and Hypotheses

How effective is the learning algorithm used in the Simulated Student? Ideally, the author's task is to demonstration solutions to only on a few problems. In our previous study, we showed that solving 10 problems was enough to generate 9 production rules for algebra equation-solving (Matsuda *et al.*, 2005b). In this paper, we focus on the simplicity of demonstration in terms of the following two research questions.

Does the order of training problems in demonstration matter? For human learning, it is likely that students will learn better when first shown easy problems and gradually shifted to more complex ones, where the complexity of a problem is defined as a number of steps to solve a problem. This observation is reasonable because human students must operate within their cognitive resources (e.g., cognitive load, memory limitation, etc). But what about machine learners? Intuitively, providing more difficult problems over and over again might achieve a better learning outcome because they provide more opportunities for learning each skill. This question is important because we want the Simulated Student to learn domain principles with fewer demonstrated problems (discussed in section 5.1).

Does the organization of demonstrations matter? In the previous study, we observed that Simulated Student could learn wrong production rules (Matsuda *et al.*, 2005b). This could occur when an *ambiguity* exists in the interpretation of demonstration. For example by observing that "$3x=9$ simplifies $x=3$," Simulated Student might infer that "the right hand side of the simplified equation is the coefficient of the term in the left hand side of the original equation"; this rule leads to the erroneous behavior "$5x=10$ simplifies as $x=5$." This type of misconception could have been avoided if the author used an example problem whose answer was not coincidentally equal to the coefficient of the original term. In general, the ambiguity problem could be avoided by providing demonstrations on various types of problems at various levels of detail. We call the diversity in those aspects the *organization* of demonstration (discussed in section 5.2).

How general is the Simulated Student framework? So far, we have worked on an algebra equation as an example domain. The generality of the framework should be tested on other domains as well (discussed in section 5.3).

3. Authoring Cognitive Tutors: No general solution yet

3.1. Cognitive Tutors and the Authoring Tools: CTAT

Building a Cognitive Tutor requires 2 basic tasks: (1) building a graphical user interface (GUI), and (2) building a cognitive model.

CTAT integrates with off-the-shelf tools for building GUIs. Those tools enable authors to simply drag and drop various GUI components (e.g., text boxes, drop-down menus, buttons, etc) into a custom dialogue without actually writing any code. Figure 1 is a GUI to learn to solve algebraic equations. This Equation Tutor simply has two text boxes, one for the left hand side and the other for the right hand side of equations. Other GUI elements such as the "Message" window, the "Done" and "Help" buttons are common in all tutors hence embedded in the GUI automatically.

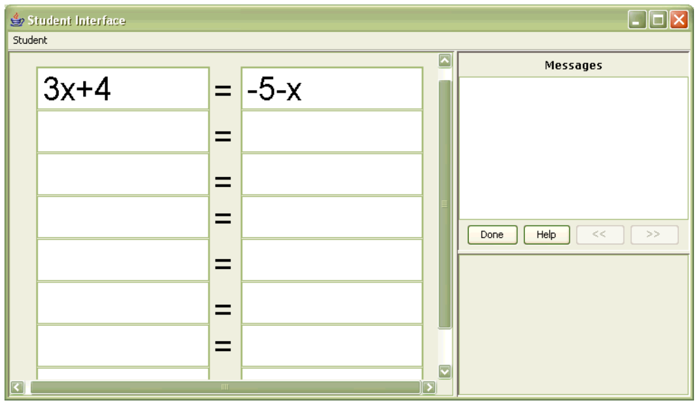


Figure 1: Example GUI for Equation Tutor

The distinguishing feature of Cognitive Tutors is *model tracing,* the process that identifies whether or not a student is performing the target task correctly. While the student is solving a problem, the Tutor monitors each step and provides appropriate hints and error messages. The tutor can do this because its *cognitive model* generates the steps in one or more solutions, while its model-tracing algorithm compares student input with the generated steps to determine where the student is in the solution space.

CTAT also provides tools to build a version of cognitive model that enables the tutor to perform a limited type of model tracing. The following section explains this.

3.2. Building a Pseudo Tutor: Problem-Specific Cognitive Modeling

The simplest version of cognitive model that can be authored with CTAT is a *record of solution* demonstrated by the author. The author uses the same GUI that the student will use and solves the same problems in the same way that the students are expected to perform. (Koedinger *et al.*, 2004).

These so-called pseudo-intelligent tutors (Pseudo Tutors for short) can perform model tracing on problems demonstrated a priori. However, if more problems are required than an author can practically demonstrate then the Pseudo Tutor technology is inadequate. To overcome this restriction, one needs to build a *generalized cognitive model*: with this, the Cognitive Tutor can perform model tracing on any instance of the problem. The following section describes these generalized models.

3.3. Building Fully Functional Model Tracing Tutor: Domain-General Cognitive Modeling

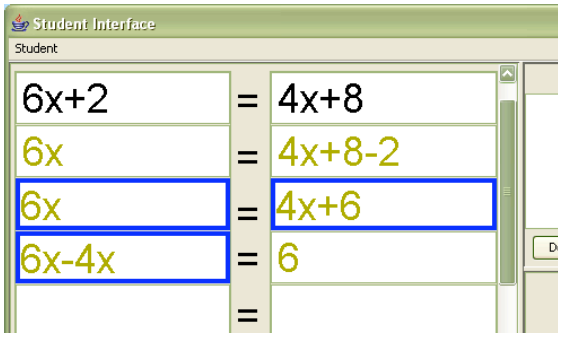
A generalized cognitive model is represented as a set of production rules. CTAT has tools to aid manually writing and debugging production rules in Jess (Friedman-Hill, 2003). But building a successful cognitive model in this way is problematic for authors who are neither cognitive scientists nor AI programmers. The next section describes our solution: using a Simulated Student to automatically generate a cognitive model by demonstration.

4. Authoring Cognitive Tutors with Programming by Demonstration

This section briefly describes how to author with programming by demonstration and then presents a basic architecture of Simulated Student. More detailed explanations can be found elsewhere (Matsuda *et al.*, 2005a).

4.1. Cognitive Modeling by Demonstration

When building a Cognitive Tutor by demonstration, the author must specify (by double-clicking) all the GUI elements that should appear in the production rule. Those GUI elements are called the *focus of attention*, because they are the elements that control decision making for the step performed. For example, in Figure 2, the author specifies "6x," "4x+6," and "6x-4x" (highlighted) as the focus of attention to enter "6." The author's demonstration is visualized as a directed graph where a node represents a solution state and an edge a cognitive skill (i.e., a production rule) to be applied to change a state. It is the author's task to annotate each edge with a *skill name*, which corresponds to a name of the production rule to be learned by the Simulated Student.



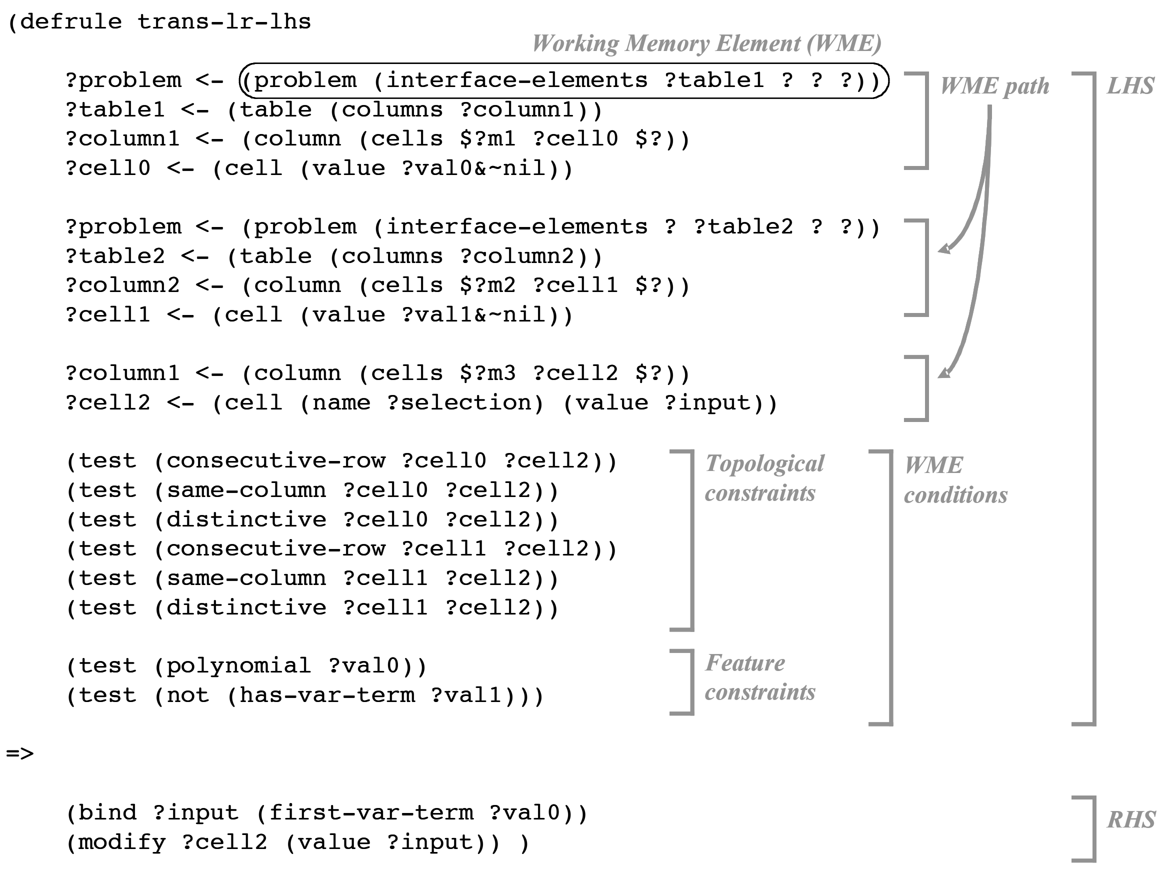
6x+2	=	4x+8
6x	=	4x+8-2
6x	=	4x+6
6x-4x	=	6
	=	

Figure 2: Providing focus of attention during demonstration

Simulated Student's learning is incremental and interactive. When a step is demonstrated, even before its skill name is labeled, Simulated Student attempts to model trace the step. A result of model-tracing is then reported to the author that allows him/her to assess the quality of the production rules. When the author annotates the name of the step, the Simulated Student generates a new production rule or modifies an existing one.

4.2. Structure of Learned Production Rules

A sample Jess production rule is shown in Figure 3. A production rule consists of two major parts: the left hand side (LHS) specifies the conditions of working memory elements (WMEs) required for the production rule to be applied, and the right hand side (RHS) specifies actions to be taken upon application.



```
(defrule trans-lr-lhs

    ?problem <- (problem (interface-elements ?table1 ? ? ?))
    ?table1 <- (table (columns ?column1))
    ?column1 <- (column (cells $?m1 ?cell0 $?))
    ?cell0 <- (cell (value ?val0&~nil))

    ?problem <- (problem (interface-elements ? ?table2 ? ?))
    ?table2 <- (table (columns ?column2))
    ?column2 <- (column (cells $?m2 ?cell1 $?))
    ?cell1 <- (cell (value ?val1&~nil))

    ?column1 <- (column (cells $?m3 ?cell2 $?))
    ?cell2 <- (cell (name ?selection) (value ?input))

    (test (consecutive-row ?cell0 ?cell2))
    (test (same-column ?cell0 ?cell2))
    (test (distinctive ?cell0 ?cell2))
    (test (consecutive-row ?cell1 ?cell2))
    (test (same-column ?cell1 ?cell2))
    (test (distinctive ?cell1 ?cell2))

    (test (polynomial ?val0))
    (test (not (has-var-term ?val1)))

=>

    (bind ?input (first-var-term ?val0))
    (modify ?cell2 (value ?input)) )
```

Figure 3: An example of production rule for algebra equation

The LHS of a learned production rule has two types of conditions: *WME paths* and *WME conditions*. A WME path identifies a working memory element representing a particular GUI element. The WME conditions represent constraints that must hold among GUI elements. There are, in turn, two types of WME conditions: *topological constraints* and *feature constraints*. Topological constraints are requirements on the locations of GUI elements (e.g., two cells next to each other). Feature constraints make requirements on the *value* of the GUI elements (e.g., a cell contains a polynomial expression). Since feature constraints can be directly translated into first order logic, we employ FOIL (Quinlan, 1990) to identify them.

The RHS of a learned production rule specifies actions to take on GUI elements. In the Equation Tutor, these actions read values from one or more cells, generate a new value from them, and write the new value to another cell.

5. Evaluation of Simulated Student

To evaluate the efficiency and generality of Simulated Student, we conducted three studies, corresponding respectively to the research questions in section 2: the test on the sequence of problems, the test on the organization of demonstration, and the cross domain generalization test.

5.1. Sequence of Problems Demonstrated

The main purpose of this study is to see if a difference in the sequence of problems demonstrated affects Simulates Student's learning.

5.1.1. Methods

Eight training problems for algebra equation-solving were demonstrated with 10 different production rules in a total of 54 steps (i.e., production rule applications). Table 1 shows the use of the rules (columns) in each of the problems (rows): an asterisk '*' indicates that the corresponding production rule was applied once on this problem; two asterisks '**' means that the rule was applied twice.

Table 1: Training problems used for the curriculum evaluation

Problem	do-arith-lhs	do-arith-rhs	done	add-lhs	add-rhs	div-lhs	div-rhs	multi-lhs	multi-rhs	copy-rhs	
x/7 = 6	*	*	*					*	*		5
8x = 16	*	*	*			*	*				5
-x = 5	*	*	*			*	*				5
x + 4 = 9	*	*	*	*	*						5
3x + 4x = 21	**	*	*			*	*			*	7
4x + 5 = 13	**	**	*	*	*	*	*				9
(x + 5)/6 = 7	**	**	*	*	*			*	*		9
x/4 + 5 = 8	**	**	*	*	*			*	*		9
	12	11	8	4	4	4	4	3	3	1	54

The training problems were selected to cover the most basic skills in this domain. The numbers in the margins show the total numbers of rule applications. To compare learning outcomes from different training-problem sequences, 12 different *problem sets* were created by randomly ordering these eight training problems.

Ten feature predicates and 24 operators were provided as the background knowledge (Table 2).

Table 2: Feature predicates and operators used for the curriculum evaluation

Feature Predicates for LHS conditions	Operators for RHS actions	
`HasCoefficient`	`CopyTerm`	`Coefficient`
`VarTerm`	`InverseTerm`	`ReverseSign`
`Monomial`	`EvalArithmetic`	`RemoveCoefficient`
`Polynomial`	`FirstVarTerm`	`LastTerm`
`HasVarTerm`	`LastConstTerm`	`RemoveFirstVarTerm`
`HasConstTerm`	`RemoveLastTerm`	`RemoveLastConstTerm`
`AllSameTypeTerms`	`Denominator`	`Numerator`
`NotNull`	`AddTerm`	`DivTerm`
`CanBeSimplified`	`MulTerm`	`DivTen`
`IsFractionTerm`	`ModTen`	`AddTermBy`
	`DivTermBy`	`MulTermBy`
	`GCD`	`LCM`

For validation, seven test problems were solved in a total of 67 steps (production rule applications) with the ten production rules generated from the training problems. Each time a training problem had been completely demonstrated, a validation test was run over the seven test problems, and solution steps were model-traced. The *accuracy* of a production rule was measured as the ratio m/N where N is the total number of times the rule should be applied in the seven test problems, and m is the number of steps that were correctly model-traced.

5.1.2. Results

Figure 4 shows the learning curves for each training condition in terms of the accuracy of production rules defined above. The x-axis shows the number of times that a production rule was applied in the demonstration (i.e., the *opportunity of learning*). The y-axis shows the average accuracy of the production rule. The graph is aggregated across all the production rules for each condition. The bold curve shows an average across the conditions.

As shown in the figure, the accuracy of production rules converged to the maximum at the end of the learning sessions regardless of the order of training problems. The current learning algorithm employed in Simulated Student is not problem-order sensitive when enough problems were demonstrated.

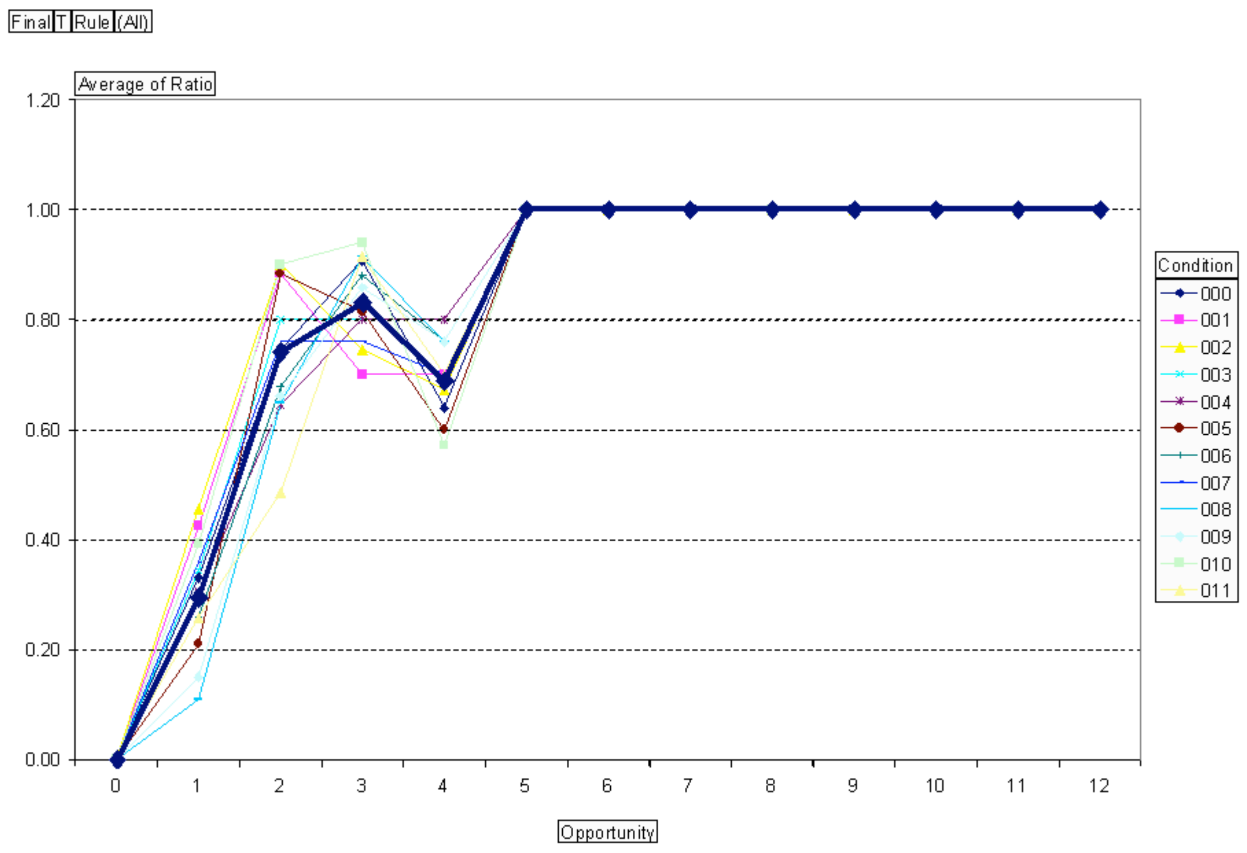


Figure 4: Learning curves aggregated across production rules and test problems

To see how many training problems were needed to reach the correct production rules, the changes in the LHS conditionals and the RHS operator sequences across the learning opportunities were compared. Table 3 shows the number of learning opportunities needed for each of the production rules to have correct LHS conditionals. For all but one (the "done" rule) production rule, three learning opportunities were sufficient to learn "correct" LHS conditionals.

Table 3: Number of training problems needed to learn correct LHS conditionals

Rule	#Conditionals	# Learning opportunities		
		Average	Min	Max
multi-lhs(A,B)	2	1.9	1	2
do-arith-lhs(A,B)	1	2.3	2	3
add-lhs(A,B)	2	2.3	2	4
mult-rhs(A,B,C)	2	2.3	2	3
div-lhs(A,B)	3	2.3	2	3
div-rhs(A,B,C)	2	2.4	2	3
do-arith-rhs(A,B,C)	1	2.6	2	3
add-rhs(A,B,C)	2	2.7	2	4
done(A,B)	4	5.3	4	7

To our surprise, the RHS operator sequences for nine of the 10 production rules were captured correctly on the first rule application. The remaining rule, "`add-lhs`" was overly specific even when all eight training problems were demonstrated. This rule cancels a constant term in the LHS as a part of an upper level operation to "move" the cancelled term to RHS. The learned incorrect rule said "take the *last term* in LHS, reverse its sign, and add it to RHS"; the rule fails on a test problem such as "*2-3x=17*," because in this case it is the first term that must be cancelled. This is an example of what we call *ambiguity* in demonstration. The next section addresses this issue in detail.

5.2. Organization of Demonstration

The *organization of demonstration* in this study is twofold: (1) the ambiguity in the problem representation, and (2) the level of detail provided in demonstration.

The ambiguity of the problem representation refers to the presence of a nondeterministic interpretation of the demonstration especially in the feature extraction. For example, the problem "*3x=9*" is ambiguous when it is simplified as "*x=3*," because "*3*" in the RHS could be "the coefficient of *3x*" or "the quotient of 9 divided by the coefficient of *3x*." This ambiguity can be clarified by another instance of an isomorphic problem, say, *5x=10*, where the isomorphic problems can be solved by applying same production rules in the same order. We call this type of ambiguity *parameter ambiguity*.

Another type of ambiguity can be even more subtle. The term *5* in *2x+5=4*, which must be transposed to the right hand side (resulting in the equation *2x=4-5*), could be a term in the left hand side that is either the last term, a constant term, the first constant term, etc. This type of ambiguity cannot be clarified with isomorphic problems but instead requires another problem with different *structure* to work as a negative example against irrelevant interpretation. For example, the term 5 cannot be the last term in LHS in *2x+5-3x+6*. We call this type of ambiguity *structure ambiguity*.

In this paper, we consider only parameter ambiguity and test a specific hypothesis: providing more detailed demonstration on problems with parameter ambiguities suppresses learning wrong production rules. To test this hypothesis, we have compared two demonstrations at the different level of detail on the same set of problems.

We have also hypothesized that decreasing a level of detail also affects learning on the problems that have no parameter ambiguity.

5.2.1. Methods

Thirteen problems shown in Table 4 were used for the study. Two different demonstrations were provided on those problems. The *less detailed demonstration* showed only a simplified equation after applying all algebraic operations. For example, when dividing both sides of an equation (say, *3x=9*) with the same number (*3*), the demonstration shows the result of the division (*x=3*) without any intermediate steps (e.g., *3x/3=9/3*). Table 4 shows how 13 problems were solved, with an asterisk showing a single rule application. This demonstration suffers from parameter ambiguity on the first four problems in Table 4. Those problems have multiple interpretations on the right hand side of the solution state, because the exact same number appears in the left hand side of the original equation. The 5th through 8th problems are isomorphic to the first four problems but have no parameter ambiguity; hence the erroneous production rules should be corrected by the time that the first eight problems are demonstrated.

Table 4: Training problems used in the study for organization of demonstration

	done	trans-lr-lhs	trans-lr-rhs	trans-rl-lhs	trans-rl-rhs	div-lr-lhs	div-lr-rhs	multi-lr-lhs	multi-lr-rhs	
x+3=6	*	*	*							3
x-5=0	*	*	*							3
3x=9	*					*	*			3
x/8=1	*							*	*	3
x+5=8	*	*	*							3
x-4=10	*	*	*							3
4x=12	*					*	*			3
x/4=3	*							*	*	3
3x-4=2	*	*	*			*	*			5
3x=2x+4	*			*	*					3
3x-3=2x+5	*	*	*	*	*					5
2=-3x+11	*	*	*	*	*	*	*			7
13=x+8	*	*	*	*	*	*	*			7
	13	8	8	4	4	5	5	2	2	51

Next, a *detailed demonstration* was made for the first four training problems shown in Table 4. In the detailed demonstration, those problems were solved by applying five production rules (instead of three as in the less detailed demonstration). The intermediate steps for the algebraic operations are explicitly demonstrated, and new production rules called "`do-arith-lhs`" and "`do-arith-rhs`" were introduced. For instance, the problem "*x+3=6*" is solved as "*x+3-3=6-3*" by applying rules "`d-trans-lr-lhs`" and "`d-trans-lr-rhs`." It then becomes "*x=3*" by applying "`do-arith-lhs`" and "`do-arith-rhs`." The detailed demonstration is summarized in Table 5.

Table 5: The detailed demonstrations

	done	do-arith-lhs	do-arith-rhs	d-trans-lr-lhs	d-trans-lr-rhs	d-div-lr-lhs	d-div-lr-rhs	d-multi-lr-lhs	d-multi-lr-rhs	
x+3=6	*	*	*	*	*					5
x-5=0	*	*	*	*	*					5
3x=9	*	*	*			*	*			5
x/8=1	*	*	*					*	*	5
	4	4	4	2	2	1	1	1	1	51

5.2.2. Results

We first examined the RHS operator sequences to see how they are generated during the learning progress. With the less detailed demonstration, several wrong production rules were generated and then fixed by the end of the learning session. On the other hand, with the detailed demonstration, RHS operators were learned correctly on the first demonstration for 7 out of 9 production rules. In the remaining two rules, "`d-trans-lr-lhs`" and "`d-trans-lr-rhs`," there were still wrong operator sequences learned (overly specific). Together these two rules transform, say, "*x+3=6*" into "*x+3-3=6-3*." Changing a level of detail does not improve learning on problems with structural ambiguity.

The difference in the degree of detail also affects the search complexity. When details of the demonstration decrease, search complexity increases because Simulated Student must search all implicit operations. As a result, when the number of RHS operators reaches four, the search becomes impractical as shown in Table 6.

Table 6: Search complexity with demonstrations at different levels of detail

# RHS operators	Detailed demonstration		Less detailed demonstration	
	Time [sec]	Space	Time [sec]	Space
1	0.00	1	0.00	1
2	0.12	19	0.04	6
3	10.21	1452	40.31	3563
4	-	-	2396.52	212780

A dash mark '-' means that there was no rule generated at the specified operator length. "Space" shows the number of nodes expanded in the search space.

In summary, less detailed demonstrations are risky for two reasons: (1) they increase the chance of parameter ambiguity, so that it becomes more likely to learn incorrect rules; and (2) the resulting production rules tend to have more RHS operators, so that the search complexity becomes impractical. The same issue could interact with human students' learning. We have yet to investigate how the level of demonstration affects students' learning.

5.3. Cross Domain Generalization

In addition to the algebra equation, we have tested Simulated Student in three other domains: multi-column multiplication, fraction addition, and Tic-Tac-Toe. The main purpose for these studies was to explore whether or not adding features and FOIL extends the generality and accuracy of the Simulated

Student beyond that shown in a previous study, which also employed programming by demonstration for authoring of Cognitive Tutors (Jarvis *et al.*, 2004). Unlike the machine-learning agent in the previous study, (1) only our Simulated Student employs FOIL to learn the feature constraints in LHS, and (2) only our Simulated Student identifies the topological constraints. Thus, the question here is how the production rules learned with these features are different from the production rules that do not have such LHS conditionals.

In the two arithmetic domains, multi-column multiplication and fraction addition, the same 10 feature predicates and 24 operators provided in the Equation Tutor (Table 2) are used. Hence it was also of interest to see whether having extra features and operators would bias learning.

5.3.1. Multi-column multiplication

This tutor has rows and columns of cells in a single table as shown in Figure 5. Some of the empty cells must be filled, but each only with a single digit. Filling in a cell corresponds to a single rule application.

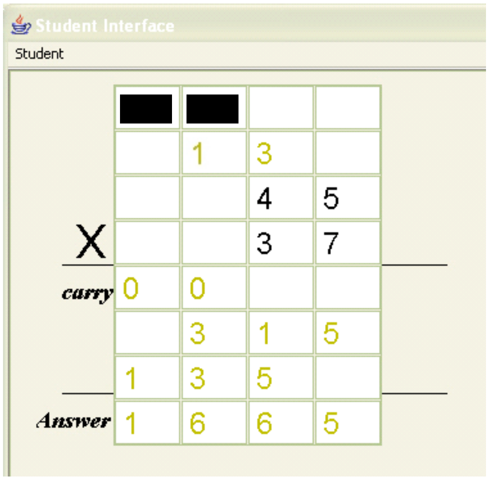


Figure 5: Multi-Column Multiplication Tutor

A 2-digit × 2-digit problem was solved in 14 steps with 10 unique rules. Simulated Student was able to learn all 10 productions in less than a few seconds each.

Ten operators (out of 24) appeared in the production rules. As for the LHS conditions, the topological constraints were captured correctly as well. FOIL did not find any feature constraints at all. This is because that the objects of manipulation in this tutor are all single digit numbers with no specific features that need to be extracted. However, this observation also depends on the strategy taken in the demonstration. For example, the production rule for writing a carry could be applied only when the product of two digits is greater than 10. Such constraint did not appear in the LHS conditionals in our study, because a carry was always filled in even when it was zero.

5.3.2. Fraction addition

The student's interface for the Fraction Tutor consists of 12 cells as shown in Figure 6. A problem is to add two fractions, shown vertically in the left-most column, by first finding a common denominator (the top two fractions in the middle column), then adding those two fractions (the bottom fraction in the middle column), and then reducing the result. There are eight unique steps to fill in eight cells in the second and third columns.

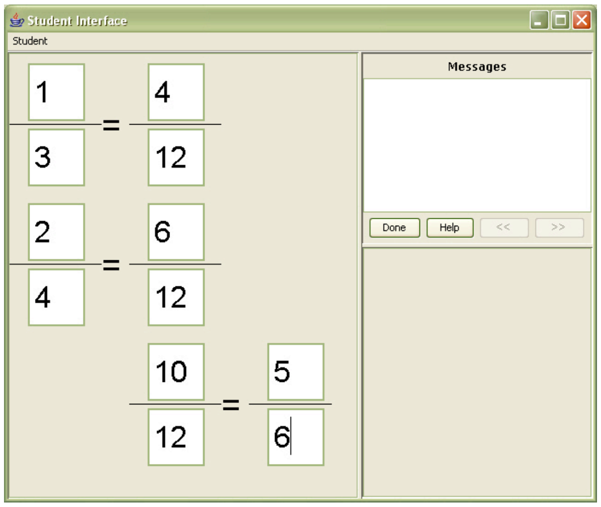


Figure 6: Student Interface for Fraction Tutor

Simulated Student was able to learn all eight rules correctly. There was parameter ambiguity that affected learning a rule to calculate the denominator. The example shown in Figure 6 illustrates this problem. The author's intention was that the denominator "12" must be the least common multiple of the denominators, but in this particular case, it could be simply the product of 3 and 4.

5.3.3. Tic-Tac-Toe

The third example is a Tic-Tac-Toe Tutor. The student's interface consists of a single 3 by 3 table. A problem represents a particular situation of the game, and students are supposed to pick the next best move, which is either to prevent the opponent from winning, to bring an immediate win, or to place a token in the central cell.

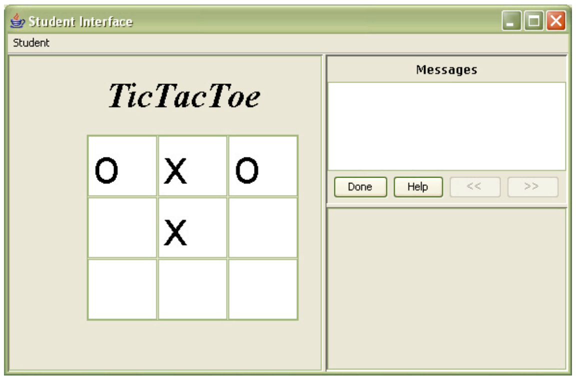


Figure 7: Tic-Tac-Toe Tutor

The most interesting issue observed through this experiment is that a fatal move, which by definition is a move that loses the game, could be identified as fatal only at the end of the demonstration. To learn rules to avoid fatal moves, the examples must be classified as either positive or negative retroactively after observing later moves.

6. Discussion

6.1. Practicality of Authoring by Demonstration

For algebra equation solving where problems are solvable in at most 10 steps and the problems share production rules (i.e., the same production rule appears across several problems), building cognitive model with programming by demonstration works quite well with relatively a few training problems, given that the demonstration is well organized in terms of the level of detail (discussed more in section 6.3).

6.2. Impact of the Sequence in Training Problems

From an authoring point of view, it is convenient that the order of training examples does not affect the quality of the production rules at the end of the learning session. The author need not carefully design a curriculum sequence.

A probable reason that the problem sequence does not matter to Simulated Student's learning is that the learning algorithm employed in Simulated Student is not cumulative in the way production rules are generated and/or refinement. Whenever an instance of production-rule application is demonstrated, Simulated Student attempts to generate a whole set of production rules that are consistent with all rule applications demonstrated so far. Thus, at the end of a learning session, the only constraint that the set of production rules hold is the consistency with the demonstrated rule applications regardless of the order.

6.3. Impact of Organization of Demonstration

The level of detail of demonstration is a particularly important question, because it affects both Simulated Student's learning and, probably, human students' learning also.

As shown in the study for organization of demonstration, as more algebraic operators are implicitly involved in a step demonstrated, it becomes more expensive (in time and space) to generate a production rule, and, more importantly, the Simulated Student is more likely to learn a wrong production rule. The latter pitfall becomes critical when the demonstration steps have parameter ambiguity. To prevent Simulated Student from learning incorrect rules, the author needs to provide a detailed demonstration or carefully design problems so that they do not have parameter ambiguity.

From the cognitive studies in the sciences of learning, it is known that to start from fully-demonstrated worked-out examples and then to gradually fade scaffolding facilitates learning (Renkl & Atkinson, 2003). Yet the current model of Simulated Student does not explain why such fading strategy works. This is a future research issue.

6.4. Limitations

The success of the learning depends in part on the available features and operators. The goal is to reduce programming effort for the author, so ideally the features and operators should be simple to code. However, some of the features and operators used in the current studies are not so simple. For instance, "`CanBeSimplified`" is one of the more complex features. If this feature is left out, Simulated Student composed rules that were overly general. For example, without this feature some productions had `(not (polynomial` *x*`))` instead of `(not (canBeSimplified` *x*`))`. As a consequence, the performance of model tracing decreased. In particular, the performance of `do-arith-lhs`, which had 100% accuracy with `CanBeSimplified`, decreased to 84%.

7. Conclusion

The evaluation studies support the thesis that Simulated Student can be a building block of intelligent authoring tools for Cognitive Tutors, while some issues for future improvement are suggested.

So far we have only tested modeling functionality of Simulated Student in the laboratory studies. To test whether this cutting-edge technology actually facilitates authoring Cognitive Tutors, evaluation studies for authoring with real human authors on a practical domain must be conducted.

As a broader benefit, Simulated Student might also inform cognitive studies for human learning. That is, Simulated Student has potential interests in simulating human learning and testing cognitive principles in teaching and learning as well. We will conduct more studies along this line of exploration.

Reference

Cypher, A. (Ed.). (1993). *Watch what i do: Programming by demonstration*. Cambridge, MA: MIT Press.

Friedman-Hill, E. (2003). *Jess in action: Java rule-based systems*. Greenwich, CT: Manning.

Jarvis, M. P., Nuzzo-Jones, G., & Heffernan, N. T. (2004). Applying machine learning techniques to rule generation in intelligent tutoring systems. In J. C. Lester (Ed.), *Proceedings of the international conference on intelligent tutoring systems* (pp. 541-553). Heidelberg, Berlin: Springer.

Koedinger, K. R., Aleven, V., Heffernan, N., McLaren, B., & Hockenberry, M. (2004). Opening the door to non-programmers: Authoring intelligent tutor behavior by demonstration. In J. C. Lester, R. M. Vicari & F. Paraguaçu (Eds.), *Proceedings of the seventh international conference on intelligent tutoring systems*.

Koedinger, K. R., Aleven, V. A. W. M. M., & Heffernan, N. (2003). Toward a rapid development environment for cognitive tutors. In U. Hoppe, F. Verdejo & J. Kay (Eds.), *Proceedigns of the international conference on artificial intelligence in education* (pp. 455-457). Amsterdam: IOS Press.

Matsuda, N., Cohen, W. W., & Koedinger, K. R. (2005a). 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.

Matsuda, N., Cohen, W. W., & Koedinger, K. R. (2005b). 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.

Murray, T. (1999). Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education, 10*, 98-129.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning, 5*(3), 239-266.

Renkl, A., & Atkinson, R. K. (2003). Structuring the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspective. *Educational Psychologist, 38*(1), 15-22.