SimStudent: Building an Intelligent Tutoring System by Tutoring a Synthetic Student

Noboru Matsuda, *Human-Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, 15213, USA*
Noboru.Matsuda@cs.cmu.edu

William W. Cohen, *Machine Learning Department, Carnegie Mellon University*
wcohen@cs.cmu.edu

Jonathan Sewall, *Human-Computer Interaction Institute, Carnegie Mellon University*
sewall@cs.cmu.edu

Gustavo Lacerda, *Machine Learning Department, Carnegie Mellon University*
gusl@cs.cmu.edu

Kenneth R. Koedinger, *Human-Computer Interaction Institute, Carnegie Mellon University*
koedinger@cs.cmu.edu

Abstract. SimStudent is a machine-learning agent that has been developed to help novice authors to build Intelligent Tutoring Systems (ITS) without heavy programming. Integrated into an existing suite of software tools called CTAT (Cognitive Tutor Authoring Tools), SimStudent helps authors to create an expert model for ITS by "teaching" SimStudent how to solve problems. There are two ways for the author to teach SimStudent: by *demonstrating* solutions or by *tutoring* SimStudent. In the former approach, the author demonstrates solution steps, and SimStudent attempts to induce underlying domain principles by generalizing those demonstrations. In the latter approach, the author gives SimStudent problems to solve, and provides feedback on the steps performed by SimStudent, and also gives "hints" for steps that SimStudent cannot perform (where the hints consist of demonstrating that particular step). To evaluate which authoring strategy better facilitates authoring, we conducted two studies. The first measured the quality of the expert model; here the authoring processes were simulated by using pre-recorded solutions for demonstrations as well as using an existing Cognitive Tutor for tutoring SimStudent. The second study measured the time for authoring; here a human author was actually using SimStudent to author a Cognitive Tutor. The results show the following: (1) Authoring by demonstration requires more time than authoring by tutoring, because when authoring by demonstration, the author frequently needs to test the quality of the expert model to determine if more demonstration is needed. In contrast, when authoring by tutoring, this decision making can be done naturally by observing SimStudent's performance during tutoring. (2) The expert model generated with authoring by tutoring tends to be more precise than authoring by demonstration. That is, when applied to solve problems, the expert model generated by tutoring makes more correct steps and/or fewer incorrect ones.

Keywords. Intelligent Authoring System, Machine Learning, Programming by Demonstration, Inductive Logic Programming, Cognitive Tutor Authoring Tools, SimStudent

INTRODUCTION

This paper describes a cutting-edge technology for authoring intelligent tutoring systems (ITS) by "teaching" a machine-learning system how to solve problems. The system is called *SimStudent*. SimStudent learns an *expert model* of the target task, which is one of the major components of a generic ITS (Shute & Psotka, 1994).

To create the expert model, one needs to do cognitive task analysis, to clarify pieces of knowledge that must be taught, and also to identify common misconceptions. Such cognitive task analysis is usually time consuming and requires substantial knowledge and skills in cognitive science. Also, to write the expert model, one must be familiar with AI-programming. Even for trained people, building an expert model for ITS is a notoriously time consuming task (Murray, 1999).

We assume here that the potential users of SimStudent are *domain experts* (e.g., curriculum developers). Domain experts are usually not familiar with cognitive task analysis and/or AI-programming; hence, it is challenging for those people to build an expert model for an ITS by hand. However, the domain experts have the expert knowledge needed to *solve* problems. Thus, our hypothesis is that SimStudent will enable these domain experts (who are novices at ITS authoring) to create an expert model easily and rapidly.

We are particularly interested in authoring a type of ITS called a *Cognitive Tutor* (Anderson *et al.*, 1995). Cognitive Tutors are known to be effective with their unique capability of immediate feedback, and context-sensitive hint messages. These intelligent assists are supported by a technique called *model-tracing*, which identifies the piece of cognitive skill from the expert model used in each problem-solving step. The technique of model-tracing is highly domain independent. This means that, in a Cognitive Tutor, a *student model* (another major component of a generic ITS) is automatically maintained by model-tracing when the expert model is provided appropriately. Thus, authoring a Cognitive Tutor is really authoring an interface for the student to communicate with the tutor (called a *student interface*) and an expert model.

There is a suit of software tools, called Cognitive Tutor Authoring Tools (CTAT), to author a Cognitive Tutor (Koedinger *et al.*, 2003). CTAT provides a graphical user interface (GUI) builder for the author to build a student interface with no programming effort. CTAT also allows authors to build a limited type of ITS, called an *Example-Tracing Tutor*, that only functions on the particular solutions demonstrated. However, the author still has to write an expert model to make a fully functional Cognitive Tutor. It is thus a reasonable extension to couple SimStudent into CTAT to generalize Example-Tracing Tutors and extract an expert model (Matsuda *et al.*, 2005).

The goal of this paper is to evaluate the effectiveness and challenges of using SimStudent for authoring. We are particularly interested in comparing alternative authoring strategies that are available for SimStudent. SimStudent learns an expert model by either (1) observing problem-solving steps demonstrated by the author, or (2) being tutored on how to solve problems. In the former approach, the author is only asked to demonstrate solution steps, and SimStudent attempts to induce the underlying cognitive principles by generalizing those demonstrations. This strategy was inspired by the technique of *programming by demonstration* (Cypher, 1993). In the latter approach, the author tutors SimStudent by giving SimStudent problems to solve, and providing feedback on each of the steps performed by SimStudent, as well as "hints" for steps that SimStudent cannot perform (basically demonstrations of those particular steps). Intuitively, tutored problem-solving might require more work from the author, but the interactivity might result in better learning outcomes. This observation brought us to some interesting research questions, including the following: How correct is a cognitive model authored with each strategy? How long does it take to author a Cognitive Tutor with each strategy? Are there strategy-specific challenges in authoring with SimStudent?

From an engineering point of view, the purpose of this paper is also to discuss details of an advanced authoring environment that allows the authors to practice *authoring by teaching* – i.e., to first teach the computer, then let the computer teach the students.

The rest of the paper is organized as follows. First, we discuss studies related to the current paper, including programming by demonstration, interactive machine learning, and authoring ITS by demonstration. We then introduce SimStudent followed by evaluation studies. There are two studies conducted for two different purposes. We then discuss pros and cons of each authoring strategy based on the results from the evaluation studies.

RELATED STUDIES

Building intelligent tutoring systems by demonstration is a very natural idea for authoring; however, few such systems have been developed to date. There was a preliminary attempt to integrate a machine-learning system into CTAT (Jarvis *et al.*, 2004). Although they have successfully demonstrated a potential of applying programming by demonstration to generate expert models, the expert models generated by their system tended to be overly generalized due to a weakness in learning the knowledge of when to apply a particular skill (see discussion of learning the feature tests in the section "Authoring Strategies and Learning Algorithms").

Demonstr8 is another example of an application of programming by demonstration for authoring; it was based on an earlier version of Cognitive Tutor, called ACT Tutor (Blessing, 1997). To overcome the computational complexity of learning, Demonstr8 provided a menu-driven language for the authors to manually compose the feature tests.

The same idea has been also applied to another type of intelligent tutoring system – a *Constraint-based ITS* (Mitrovic *et al.*, 2007). The Constraint Authoring System generates a domain model as a set of constraints, starting from a given domain ontology and a set of model solutions performed by domain experts (Suraweera *et al.*, 2005).

More broadly, programming by demonstration has been proven to be very effective as a way for novice programmers to improve their productivity. Lau et al (2003), for example, applied the technique of programming by demonstration for editing macros used in a text editor. It has been claimed that programming by demonstration often learns the target language with a small number of examples.

SIMSTUDENT: A SYNTHETIC STUDENT THAT LEARNS AN EXPERT MODEL

Although the SimStudent technology is domain-independent, in this paper, we use an Algebra Equation Tutor as a hypothetical intelligent tutoring system, which hypothetical authors will author with SimStudent. In this section, we first introduce the Algebra Equation Tutor, and then give a detailed explanation of how SimStudent works.

Example Cognitive Tutor: Algebra Equation Tutor

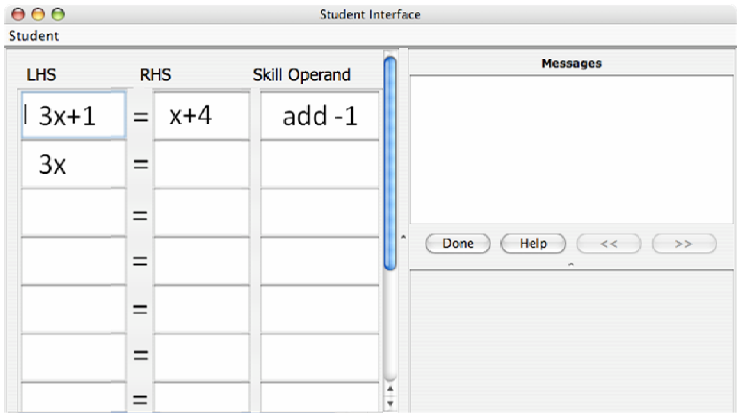
Suppose that the author is building the Intelligent Tutoring System shown in Fig. 1. This is the tutor that we use for illustrations in the rest of the paper.

In this tutor, a mathematical operation to transform a given equation to another form must be specified. To transform an equation, an operation must be entered first in a cell labelled "Skill Operand." Then, the left-hand and right-hand sides of the transformed equation must be entered in the adjacent row. Fig. 1 shows that the operation to "add -1" to both sides of "$3x+1=x+4$" was applied, and the left-hand side ("$3x$") has just been entered.

In other words, in this student interface, a single *equation-solving step* (e.g., transforming "3x+1=x+4" into "3x=x+3") is composed of three observable *tutor steps*: (1) selecting an operation for transformation, (2) entering an expression for the left-hand side, and (3) entering an expression for the right-hand side. In this paper, the word "step" means one of these three tutor steps. An operation for transformation must be specified prior to entering any expressions. The order of entering sides is arbitrary, but both sides must be entered before selecting the next operation. The first step is called *transformation step*, and the last two steps are called *type-in* *steps*. The skills to select an appropriate operation are called the *transformation skills*, and the skills to enter left- and right-hand sides are called the *type-in skills*.

The specific type of intelligent tutoring system used in this study is called a *Cognitive Tutor*. A Cognitive Tutor provides *immediate feedback* and *just-in-time hint messages*. On every step that a student performs, the Cognitive Tutor provides feedback that indicates the correctness of the step. The student is forced to perform a step until the step is performed correctly. If the student cannot perform the step, he/she can ask for a hint. Typically, the tutor provides a hint at a general level first, then gradually makes it more specific if the student asks for further assistance (by clicking the [>>] button in Fig. 1). The most specific level of hint is an explicit description of how to perform the step (e.g. "enter 3x"); hence the most specific hint is equivalent to a demonstration of a step.

The immediate feedback and the just-in-time hint messages can be provided automatically because the tutor can keep track of the student's problem-solving progress – i.e., the tutor knows where on the solution path the student is, and what are the correct actions at that point. This technique is called *model-tracing*. The Cognitive Tutor relies on a model of a domain expert's knowledge, called a *cognitive model*, to perform model-tracing. The next section describes the cognitive model in detail.



LHS		RHS	Skill Operand
3x+1	=	x+4	add -1
3x	=		
	=		
	=		
	=		
	=		
	=		

For this tutor, a student is asked to explicitly enter a transformation skill in the third column called "Skill Operand."

Fig. 1. The Student-Tutor Interface for the Algebra I Tutor.

Domain Expert Model

In Cognitive Tutors, a domain expert model is represented as a set of production rules. Each of the production rules represents an individual skill needed to perform a particular step on the student interface.[1]

Performing a step is recorded as a tuple that consists of the *action* taken (e.g., "entering some text"), the place that was *selected* to take the action (e.g., "the second cell in the first column"), and the value that was *input* (or "entered," if you will) as a result of taking the action (e.g., the string "*3x*"). These elements individually are called the *selection*, *action*, and *input*, and a tuple of <selection, action, input> is called an *SAI tuple*.

A production rule models a particular skill in terms of *what*, *when*, and *how* to generate a particular SAI tuple; it means, roughly "To perform this step, first look at X in the student interface and see if a condition Y holds. If so then do Z." The first part of the production rule, representing X (*what*), is called the *focus of attention*; it specifies particular elements of the student interface with certain constraints like "the second cell in the table" shown in the student interface. The second part of the production rule, representing Y (*when*), is called the *feature test*. The feature test is a set of conditions about the focus of attention that must be true – e.g.,

[1] Generally speaking, authors sometimes model a single observable step as a *chain of production-rule applications*. However, SimStudent generates a single production rule per observable step.

the two cells must be in the same row, the expression in the cell must be polynomial, etc. Lastly, the part of the production rule representing Z (*how*) is called the *operator sequence*. The operator sequence consists of a chain of primitive operations that generates the value of the input in the SAI tuple from the values of the focus of attention.

Together, the focus of attention and the feature tests compose the left-hand side (i.e., the condition part) of a production rule. The right-hand side (i.e., the action part) of a production rule contains the operator sequence.

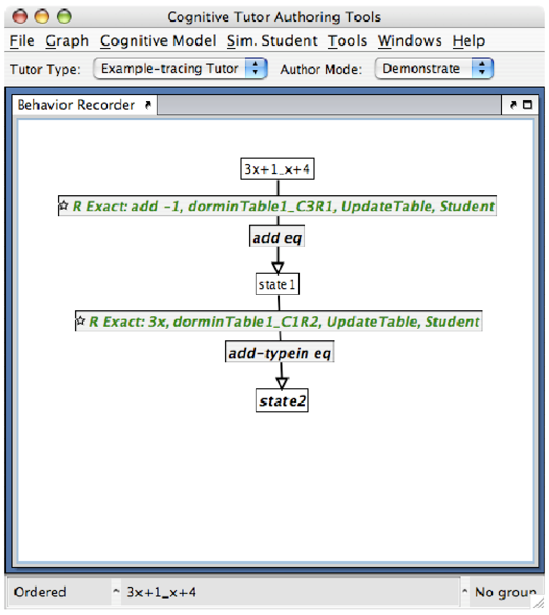
When being taught, SimStudent learns skills and generates production rules. The next section provides detailed explanation about the interaction between the author and SimStudent (i.e., how to author a Cognitive Tutor with SimStudent) followed by the learning algorithms.

Authoring a Cognitive Tutor with CTAT and SimStudent

It must be clarified that SimStudent, embedded in Cognitive Tutor Authoring Tools (CTAT), only helps authors create a cognitive model. All of other aspects of authoring supports are provided by CTAT. This includes building a graphical user interface (GUI), visualizing solution steps, annotating solutions steps with hint messages, and testing and debugging the tutor (mostly, this means debugging the production rules).

With the bare CTAT (i.e., no assistance from SimStudent in creating a cognitive model), the authors can build a limited version of a Cognitive Tutor, called an *Example-Tracing Tutor*. An Example-Tracing Tutor is a collection of model solutions composed by the author. Example-Tracing Tutors have the same instructional capability as Cognitive Tutors, i.e., they provide immediate feedback and just-in-time hint messages. This can be done by annotating each of the steps in the Example-Tracing Tutor with associated hint messages and skill names. In other words, Example-Tracing Tutors are fully functional Cognitive Tutors, but only work for the specific problems that have been explicitly solved by the author.

Suppose that the author has just finished building the student interface shown in Fig. 1, using the interactive GUI components provided by CTAT. Now the author can demonstrate solutions on this interface and store those solutions into Example-Tracing Tutors. The solution steps are recorded and visualized in a directed graph, called a *behavior graph*, as shown in Fig. 2.


Cognitive Tutor Authoring Tools
File Graph Cognitive Model Sim. Student Tools Windows Help
Tutor Type: Example-tracing Tutor
Author Mode: Demonstrate
Behavior Recorder
3x+1_x+4
R Exact: add -1, dorminTable1_C3R1, UpdateTable, Student
add eq
state1
R Exact: 3x, dorminTable1_C1R2, UpdateTable, Student
add-typein eq
state2
Ordered
3x+1_x+4
No group

A behavior graph showing the first two steps to solve an equation "3x+1 = x+4." The first step is to enter "add -1"; its skill name is called "add." This step transitions from the initial state (labelled as "3x+1_x+4") into a state called "state1." The second step is to enter "3x"; its skill name is called "add-typein." This step transitions from the state called "state1" into another state called "state2."

Fig. 2. An example of a behavior graph.

A behavior graph consists of *states* and *edges*. An edge connects two states (e.g., “state1” and “state2”) and has two property boxes visually associated – one property box showing an SAI tuple (with some extra information) and another property box showing a skill name. The author can associate hint message to the step by clicking on the property box.

To turn an Example-Tracing Tutor into a fully capable *Model-Tracing Tutor* that performs model-tracing on a wide variety of problems, one must write production rules that model state transitions (represented as the edges in the behavior graph) for arbitrary problems. This is what SimStudent does for the author. Instead of writing production rules, the author can simply “teach” SimStudent how to perform each step. There are two ways to teach: (1) by demonstrating steps, or (2) by tutoring SimStudent. Both can be done by using the student interface. The next section describes these two styles of authoring with SimStudent.

Authoring Strategies and Learning Algorithms

Learning by Demonstration is the most basic authoring strategy implemented for SimStudent. There is also a slightly modified version of Learning by Demonstration, called Example Study, where learning happens only when SimStudent does not have a production rule that generates a step demonstrated. The third learning strategy we considered is learning by being tutored, where SimStudent is engaged in tutored problem-solving.

Background knowledge

Prior to learning, SimStudent is given the hierarchical structure of the elements in the student interface, which are used to express the constraints for the focus of attention part of a production rule; a set of *feature predicates,* which are used to define the feature tests; and a set of *operators,* which are composed to form the operator sequence. SimStudent has a library of feature predicates and operators that are useful for tutoring general arithmetic and algebra, but authors might need to write domain-specific feature predicates to use SimStudent in other domains.

For the current example domain, i.e., algebra equation solving, there are 28 operators and 16 feature predicates shown in Fig. 3.

Feature predicates	**Operators**	
`HasCoefficient`	`AddTerm`	`MulTermBy`
`HasConstTerm`	`AddTermBy`	`Numerator`
`HasVarTerm`	`Coefficient`	`ReverseSign`
`Homogeneous`	`CopyTerm`	`RipCoefficient`
`IsFractionTerm`	`Denominator`	`SkillAdd`
`IsConstant`	`DivTerm`	`SkillClt`
`IsDenominatorOf`	`DivTermBy`	`SkillDivide`
`IsNumeratorOf`	`EvalArithmetic`	`SkillMultiply`
`IsPolynomial`	`FirstTerm`	`SkillRf`
`Monomial`	`FirstVarTerm`	`SkillMt`
`NotNull`	`GetOperand`	`SkillSubtract`
`VarTerm`	`InverseTerm`	`VarName`
`IsSkillAdd`	`LastConstTerm`	
`IsSkillSubtract`	`LastTerm`	
`IsSkillDivide`	`LastVarTerm`	
`IsSkillMultiply`	`MulTerm`	

Fig. 3. Feature predicates and operators

Learning by Demonstration

When demonstrating a step, the author first needs to specify the focus of attention by double-clicking the appropriate elements on the student interface. Then he/she performs a step—that is, takes an “action” upon a “selection” with an appropriate “input” value. Finally, the author needs to label the demonstrated step (i.e., annotate it with a skill name).

When a step is demonstrated for a particular skill K with a focus of attention F and an SAI tuple T, the pair <F, T> becomes a *positive example* of the skill K. The pair <F, T> also becomes a *negative example* for all other skills. This indicates the following to SimStudent: "to apply skill K to carry out the SAI tuple T when you see the focus of attention F, but do not apply any skills K′ (where K′ is different from K) when you see F." We call this kind of negative example an *implicit negative example* for K′*,* as contrasted with the explicit negative examples used for tutored problem solving (which will be described in the next section). Once a positive example is acquired, it stays a positive example throughout a learning session. On the other hand, an implicit negative example for a skill K′ may later become a positive example for K′, if the same focus of attention is eventually used to demonstrate K′.

When a new positive or negative example is added for a particular skill, SimStudent *learns* the skill by generalizing and/or specializing the production rule for the skill. In particular, SimStudent modifies the rule so that it applies to all positive examples and does not apply to any negative examples. The focus of attention, the X (*what*) in the production rule, is generalized so that it is consistent with all instances of the focus of attention appearing in the positive examples. Generalization is done by using a predefined version space for the hierarchically structured graphical interface elements. For example, "the first column" is generalized to "any column." Feature tests, Y (*when*), are also generalized and/or specialized so that they cover all positive examples and no negative examples; this is done by Inductive Logic Programming (Muggleton & de Raedt, 1994) in the form of Foil (Quinlan, 1990). The operator sequence, Z (*how*), is generalized so that it generates "input" values from the focus of attention for all SAI tuples in the positive examples. This generalization is done by searching for a shortest chain of operators that covers all positive examples.

Fig. 4 shows pseudo-code representing the learning procedure for Learning by Demonstration. This code is used to explain other authoring strategies.

```
Learning-by-demonstration(foa, sai, skill) {
    add-positive-example-for(skill, foa, sai);
    foreach existing-skill in previously-learned-skills do
        add-negative-example-for(existing-skill, foa, sai);
    all-positive-examples ← get-all-positive-example-for(skill);
    all-negative-examples ← get-all-negative-example-for(skill);
    searchRhsOperatorSequence (foa, sai, all-positive-examples);
    searchLhsConditions(skill, all-positive-examples, all-negative-examples);
}
```

Fig. 4. Pseudo code for learning from demonstration

Learning by Example Study

In the Example Study authoring strategy, the author does the same thing as in Learning by Demonstration, so from the author's point of view, the strategy is identical to Learning by Demonstration. However, the way SimStudent learns skills is slightly different. In the Example Study strategy, SimStudent tries to "*explain*" the step demonstrated, by identifying previously-learned production rules that generate the step demonstrated. Only when there is no such production rule does SimStudent attempt to learn skills. When learning is applied, SimStudent learns in the same way as in the Learning by Demonstration strategy described above. In other words, SimStudent learns skills only when it fails to explain a step. This is analogous to the way in which human students behave when learning from worked-out examples: the students try to self-explain the demonstrated solutions.

Fig. 5 shows pseudo-code for Example-Study. The function `model-trace`(*sai*) encapsulates the model-tracing process described earlier; it returns a skill name if there is a skill that explains the step performed (represented as a SAI tuple).

```
step-demonstrated-for-example-study(foa, sai, skill) {
    model-skill ← model-trace(sai);
    if (model-skill = null) {
        step-demonstrated(foa, sai, skill);
    }
}
```

Fig. 5. Pseudo code for Example Study

Learning by Tutored Problem-Solving

To tutor SimStudent with this strategy, the author enters a problem (in the student interface), and lets SimStudent to solve it. For every step, SimStudent does one (or both) of two things: (1) SimStudent applies existing production rules. Each rule application is visualized as a step in the behavior graph, just like the one shown in Fig. 2. The author then provides feedback for each rule application. In the current implementation, SimStudent shows the author *all* of the possible rules that could be applied (one after another), and thus obtains feedback on each applicable rule. (2) If there is no correct rule application found, then SimStudent asks for a hint from the author. The author then demonstrates the step, in the same way as in the learning-by-demonstration strategy,

Fig. 6 shows pseudo-code for Tutored Problem Solving. The function `apply-rule`(*skill*) applies a specified *skill* to a current problem situation, and receives feedback from the author. It returns a tuple including the feedback and the result of the skill application (in terms of the focus of attention and the SAI).

```
tutored-problem-solving(problem) {
    foreach step in problem do {
        step-performed ← false;
        foreach skill in all-applicable-skills do {
            <feedback, foa, sai> ← apply-rule(skill);
            if (feedback = "correct") {
                add-positive-example-for(skill, foa, sai);
                step-performed ← true;
            } else {
                add-negative-example-for(skill, foa, sai);
            }
        }
        if (step-performed = false) {
            <modelSai, modelFoa, modelSkill> ← askHint(step);
            step-demonstrated(modelFoa, modelSai, modelSkill);
        }
    }
}
```

Fig. 6. Pseudo code for Learning by Tutored Problem-Solving

OVERVIEW OF THE EVALUATION STUDIES

We conducted studies to evaluate and compare two authoring strategies – authoring by demonstration (Example Study) and authoring by tutoring (Tutored Problem-Solving). We measured the *quality of production rules learned* and the *time spent on authoring* as the two major dependent variables.

We conducted two different studies, one for each of the two dependent variables. The first study (the *learning strategy study*) evaluated the quality of production rules. For this study, instead of asking human participants to author Cognitive Tutors, we simulated the authoring process by using pre-recorded solutions and by having SimStudent interact with an existing hand-coded Cognitive Tutor. The second study (the *authoring study*) evaluated the time spent on authoring. A human author was asked to author a Cognitive Tutor with each of the two authoring strategies.

In both studies, we used the Algebra Tutor shown in Fig. 1 as an example tutor to be authored. The following sections describe these studies and results in detail.

LEARNING STRATEGY STUDY

The goal of the learning strategy study is to see which learning strategy, Tutored Problem-Solving and Example Study, is better in terms of the quality of the cognitive model generated.

Method

For each authoring strategy condition (Example Study and Tutored Problem-Solving), SimStudent was trained on 20 training problems and tested on 10 test problems. There were seven different sets of training problems. In other words, for each authoring strategy condition, we built Cognitive Tutors seven times, each using SimStudent on a different set of 20 training problems. Therefore, there were a total of 14 Cognitive Tutors built in this study.

To test the quality of the production rules learned by SimStudent during training, we made a (single) set of 10 test problems. Each time SimStudent completed training on a single problem, we tested the quality of the production rules, by having SimStudent solve the test problems.

Since this study is quite time-consuming for the human participants, due to the nature of the training-testing iteration, for the current study, rather than using actual human participants to author Cognitive Tutors with SimStudent, we simulated the authoring processes. For the Example-Study condition, all demonstrations were pre-recorded. These pre-recorded demonstrations were collected from a previous classroom study conducted in a Pittsburgh Science of Learning Center (PSLC) LearnLab.[2] In the LearnLab study, the Carnegie Learning Algebra I Tutor was used in an urban high-school algebra class. The high-school students were asked to use the Algebra I Tutor individually. The students' activities were logged and stored into the PSLC database, called DataShop.[3] We then extracted problems and human students' *correct* steps from DataShop to use in the current study. An entire (correct) solution for a particular problem made by a particular student became a single training problem. The seven sets of 20 training problems and a set of ten test problems were randomly selected from the DataShop data.

For the Tutored Problem-Solving condition, we used the Algebra I Cognitive Tutor – a commercial software package produced by Carnegie Learning Inc. That is, SimStudent was tutored by a computer tutor, where the computer tutor simulates the human tutor. To perform this simulation, we implemented an interface for SimStudent to communicate with the Algebra I Cognitive Tutor, so that when SimStudent asked for a "what-to-do-next" hint from the Carnegie Learning Algebra I tutor, the Carnegie Learning Tutor provided a demonstration—i.e., a precise instruction for what to do next in the form of an SAI tuple. Also, whenever SimStudent

[2] A LearnLab is a classroom or other real-world educational setting instrumented for experiments in learning science. The PSLC is funded by the National Science Foundation award number SBE-0354420. See www.learnlab.org.

[3] www.learnlab.org/technologies/datashop

performed a step, each of the possible rule applications was sent to the Carnegie Learning Algebra I Tutor to get feedback. The same seven sets of 20 training problems were used for the Tutored Problem-Solving conditions, but the ways they were solved were generally different from the students' pre-recorded demonstrations.

In summary, we conducted a study where SimStudent was used to author a Cognitive Tutor for algebra equation solving. There were two authoring strategy conditions – Example Study and Tutored Problem-Solving. There were seven "author" conditions – each "author" was given a set of 20 training problems and a set of 10 test problems. The training problems were all different across the seven "author" conditions, but all "author" conditions used the same test problems. Each "author" created two Cognitive Tutors, one by Example Study and the other one by Tutored Problem Solving. Each time an "author" finished training SimStudent on a single training problem, the quality of the production rules was tested. All these authoring actions were simulated, i.e. done automatically without human interaction, by making use of either pre-recorded solutions (made by human students) or an existing Algebra I Cognitive Tutor.

The next section describes the testing procedure, and more importantly, how the dependent variable was calculated.

Evaluation Metrics

The quality of production rules was measured in terms of the accuracy of rule applications during problem-solving—the more that the rules are correctly applied, the better the production rules.

For each of the steps in a test problem, we asked SimStudent which production rules could be fired. Since we also wanted to see how *badly* SimStudent might solve problems in addition to how well it *could* solve problems, we evaluated all *possible* rule applications for each step. More precisely, for each step, we enumerated all production rules whose left-hand side conditions held. The correctness of each rule application was evaluated using the Carnegie Learning Algebra I Tutor. The steps performed by SimStudent were coded as *correct* if there was at least one correct rule application attempted. Otherwise, the steps were coded as *missed*.

We define a dependent variable (called the *Step score*) that represents how well the learned production rules performed on individual steps in the test problems. A step score is zero if the step was missed. Otherwise, a step is scored as a ratio of the number of correct rule applications to the total number of rule applications applicable to that particular step. For example, if there were 2 correct and 6 incorrect rule applications for the step, then the Step score for that step is 0.25. The step score ranges from 0 (no correct rules applicable at all) to 1 (no incorrect rules applicable, and at least one correct rule applies). We define the *Problem score* as the average *Step score* for all steps in a test problem.

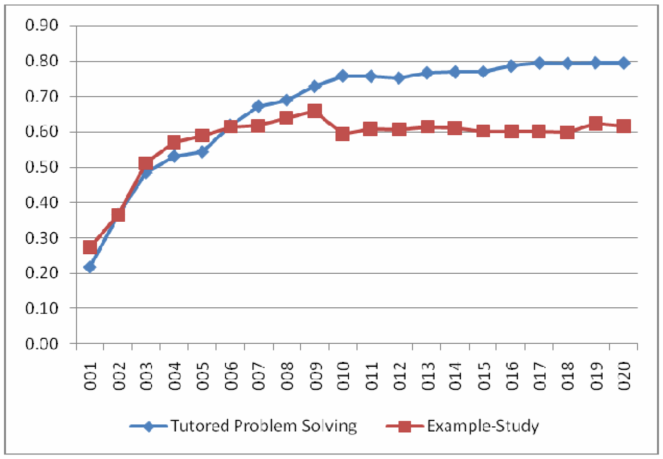
In general, there are several correct and incorrect rule applications available for each step. Since SimStudent does not have any strategy to select a single rule among these conflicting alternative rule applications, the Step score can be seen as a probability that the step is performed correctly at the first attempt.

Results

Overall learning performance

Fig. 7 shows average Problem score of the ten test problems aggregated across the seven "author" conditions (i.e., the average of 70 Problem scores) measured after each time SimStudent was trained on a training problem. In both conditions, the performance improved almost equally on the first eight training problems, but then the Example Study condition started to perform more poorly. Tutored Problem-Solving, on the other hand, kept improving until the 17th training problem. There was a point, for both conditions, where the improvement of the performance on the test problems diminished to almost nothing. After training on 20 problems, the average Problem score for the Tutored Problem-Solving condition was 0.80, and 0.62 for the

Example Study condition. The difference was statistically significant (t = 7.94, N = 70, $p <$ 0.001). Hence, *overall, Tutored Problem-Solving outperformed Example Study on the Step score metric.*



X-axis shows the number of training problems. Y-axis shows the average of the Problem score on the ten test problems aggregated across the Training Problem conditions.

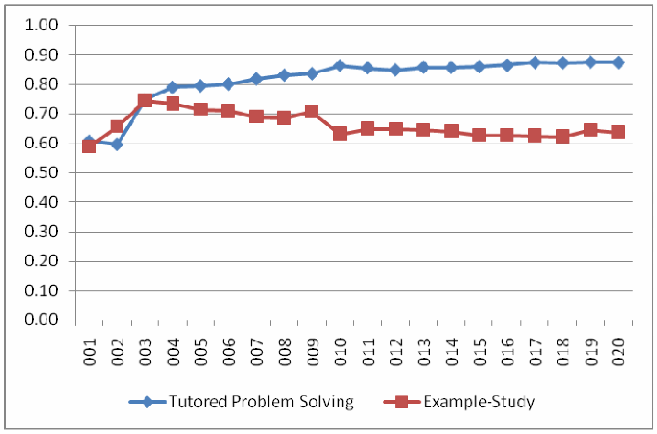
Fig. 7. Average Problem scores.

To understand why Tutored Problem Solving led to a better learning outcome, we measured two additional dependent scores: (1) The *Precision Score*, showing the ratio of the number of correct to incorrect rule applications for a step (if there are any rule applications at all), and (2) the *Recall Score* showing the ratio of the number of steps performed correctly to the total number of steps in a test problem. Note that the Precision score is undefined for steps where there is no rule applications made, while the Recall score does not take the number of rule applications into account.

A high Precision score does not necessarily predict a high Recall score. The Precision score does not account for the missed steps, which by definition, are the steps where no correct rule application was made – i.e., the Precision score gets credit for the correct rule applications, no matter how many other steps were missed. Similarly, a high Recall score does not guarantee a high Precision score. The Recall score does not take into account incorrect rule applications – a step is coded as “correct” if there is at least one correct rule application, no matter how many incorrect rule applications are made.

Fig. 8 shows the average Precision scores for the ten test problems aggregated across the seven “author” conditions. The Tutored Problem-Solving condition showed a rapid improvement until the 9th training problem. After that, the Precision score stayed at the same level. The Example-Study, on the other hand, did not improve the Precision score even after being trained on 20 problems – it started at 0.59 and ended up with 0.64. On the 20th training problem, the difference between Tutored Problem-Solving and Example-Study was statistically significant (t = 10.61 N = 70; $p <$ 0.001).

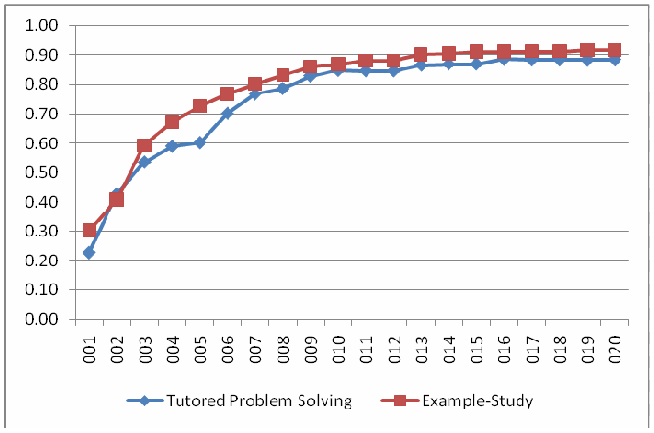
These data show that *Tutored Problem-Solving learned more correct and/or less incorrect production rules than Example-Study.*



X-axis shows the number of training problems already executed at the time the Precision score (Y-axis) was measured.

Fig. 8. Average Precision scores.

Fig. 9 shows average Recall scores for the ten test problems aggregated across the seven author conditions. Both conditions showed improvement over the training problems. At the beginning, the Tutored Problem Solving showed slightly inferior performance on the Recall score, but at the end, the difference was not statistically significant (t = 1.50, N = 70; p = 0.13). This means that *the two authoring strategies learned cognitive models equally well in terms of the number of steps eventually performed correctly*.



X-axis shows the number of training problems already executed at the time that the Recall score (Y-axis) was calculated.

Fig. 9. Average Recall scores.

Error of Commission

To see what kind of errors were made on the test problems, and more interestingly, to see if there was any difference in the errors made by different conditions, we conducted a qualitative analysis of incorrect rule applications by categorizing the errors made on the test problems.

Regardless of the learning strategy, there were only two types of errors after training on 20 problems: (1) *No-Progress errors* – i.e., applying a transformation skill that does not make the equation any closer to a solution, and (2) *Step-Skipping errors* – i.e., attempting to apply a transformation skill without completing the previous type-in step(s).

An example of a No-Progress error is to "subtract 2x" from 2x+3=5. This is a mathematically valid step, but the resultant equation 3=-2x+5 requires two transformation skills and four type-in steps, which is same as for 2x+3=5. Thus, "subtract 2x" is considered as a wrong step in the current study (and by the Carnegie Learning Algebra I Tutor as well). No-Progress errors appeared in both learning conditions.

An example of a Step-Skipping error is to apply another transformation skill to the situation shown in Fig. 1. Suppose that the student entered "divide 3" into the rightmost cell on the second row when the middle cell (right-hand side of the equation) is still blank. A type-in step entering "x+3" has been incorrectly skipped.

Quite interestingly, there were no Step-Skipping errors observed for the Tutored Problem Solving condition. Why? We hypothesized that only the Tutored Problem-Solving strategy had a chance to revise incorrect production rules during training, by making a Step-Skipping error and receiving negative feedback, and that this negative feedback allowed SimStudent to accumulate the negative examples needed to correctly learn LHS conditions for the incorrect production rules. In short, we hypothesize that *making an explicit error and getting a feedback on the error (recall that this feedback simply indicates if the step is correct) should positively contribute to learning*. To test this hypothesis, we controlled the creation of negative examples by modifying SimStudent for the Tutored Problem Solving condition so that it would not record negative examples for incorrect rule applications. More precisely, in Fig. 6, the invocation of the function `add-negative-example-for`() was disabled, but SimStudent still received negative feedback for incorrect rule applications. Other functionalities of the inner `foreach` loop over all applicable skills were intact, thus SimStudent still received the same amount of positive examples during training as the original version.

With this modification, the Tutored Problem Solving condition made the same Step-Skipping errors that were otherwise made only by the Example-Study condition. Thus, *it was the* explicit *negative examples obtained by incorrect rule applications that caused the high Precision score for the Tutored Problem-Solving condition*.

This modification did not affect the appearance of the No-Progress errors – having more negative examples did not prevent skills from being incorrectly generalized and making No-Progress errors. It turned out that *learning appropriate conditions for when to apply transformation skills is indeed quite challenging in this domain*. This is also true for human learning – human learners often make the No-Progress errors. To avoid No-Progress error, one must have a macro level view of rule applications to assess if a rule application makes a progress towards a solution. SimStudent apparently needs other type of feature predicates to learning feature tests for such a macro level optimality.

AUTHORING STUDY

This section describes the second study, which evaluated the amount of time required for authoring using each authoring strategy.

Method

There was one (human) participant involved in this study. The participant was asked to author an Algebra Equation Tutor twice, once for each of the two authoring strategies. The participant (called the author hereafter) was given 20 training problems and 10 test problems.

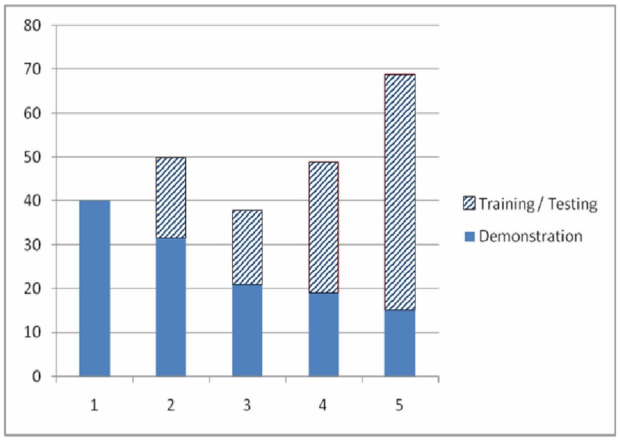
In the *Example-Study condition*, the author first demonstrated solutions for the ten test problems and saved them for future use. The author then demonstrated solutions for the first five training problems, stored them, and trained SimStudent using those five problems. Once SimStudent learned a cognitive model, the author used the test problems to measure the quality of production rules (measuring quality with the same measure used in the previous section). Tools were provided so that the author could carry out this test easily, and examine the results. The author then decided whether he should provide more demonstrations by examining SimStudent's performance on the test problems. If the author decided that further training was necessary, then the author demonstrated another five problems, and added them to the existing pool of training problems (i.e., the number of training problems increased by five for each iteration). This iteration of training and test was repeated until either the author decided to stop, or a total of 20 training problems was demonstrated (i.e., for a maximum of four iterations).

In the *Tutored Problem-Solving condition*, the author tutored SimStudent on the given training problems one after another until the author determined that SimStudent learned the cognitive model well enough. The decision was made again based on the performance of SimStudent during tutoring – if SimStudent had learned skills well enough, then SimStudent should eventually perform every step correctly without making any incorrect attempts. There was also a limit of 20 training problems placed on this condition.

Results

Fig. 10 shows time spent to author with the Example-Study strategy. The X-axis shows the number of the demonstration-training-test iterations for authoring. The solid bars show time (in minutes) spent on demonstrating training problems whereas the hashed bars show time spent on training and testing. The first iteration was to prepare for the test – demonstrating solutions for the 10 given test problems.

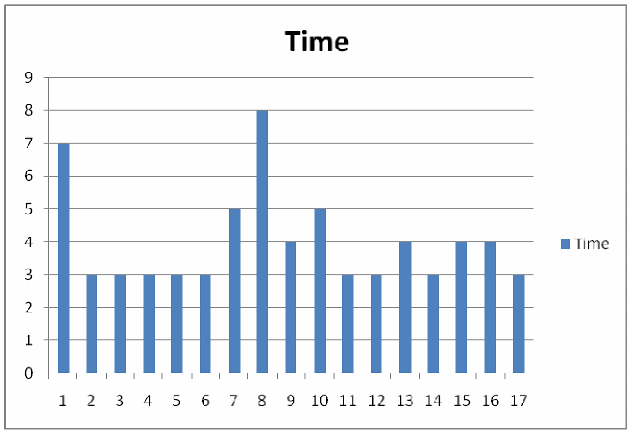
The results show that demonstrating a solution for a single problem took about 4.35 minutes on average. (This is about the same as for tutoring a single problem, as is discussed below). However, in the Example-Study condition, the author needs to train SimStudent on the training problems as well as to test the production rules, which takes about 20 minutes on average over the four iterations. The time to train SimStudent gets longer, because every time new training problems are constructed, they must be added to existing problems, and hence the number of demonstrated training examples increases over time.



The X-axis shows the number of iterations for authoring, where the first iteration is only to demonstrate on the 10 problems for testing. The second through fifth iterations consist of demonstrating on five problems (“Demonstration”) as well as training and testing SimStudent (Training / testing). The Y-axis shows time (min.).

Fig. 10. Time spent to author with the Example-Study strategy.

Fig. 11 shows time spent to author with the Tutored Problem-Solving strategy. For algebra equation solving, the average time to tutor on a single problem was about four minutes. The time spent on each tutoring problem was nearly constant.



The x-axis shows the number of training problems tutored. The y-axis shows time (min.) spent on tutoring.

Fig. 11. Time spent authoring by tutoring SimStudent.

For the Tutored Problem-Solving condition, there is no extra time cost other than tutoring. Even though the author has to provide feedback on the steps performed by SimStudent (an extra task compared with the Example-Study), the time spent tutoring a single problem is, on average, about the same as just demonstrating a solution for it. The time for tutoring is comparable because the number of steps demonstrated during tutoring ("hints") decreases as learning progresses, and because providing feedback is much quicker than demonstrating steps. Regarding the balance between demonstration and feedback, the author needed to demonstrate 8.37 steps per problem in the Example-Study condition, while for Tutored Problem-Solving, the author needed to demonstrate 1.67 steps per problem in addition to providing 2.08 instances of positive and 8.48 instances of negative feedback. *Given that providing feedback is faster and easier than demonstrating a step, Tutored Problem Solving is indeed relatively easier for authoring*.

DISCUSSION

Impact of the Learning Strategy

It turned out that *Tutored Problem-Solving is a better authoring strategy, when using the SimStudent technology, both in terms of the quality of the cognitive model generated by SimStudent and the time needed for authoring*. Even though both authoring strategies require the same number of training problems, Tutored Problem-Solving takes less time than Example-Study, where the author needs to spend more time on the demonstration-training-test cycle.

One might be curious, however, about the effects of modifying the authoring strategies. The fact that the number of positive and negative examples affects the quality of production rules motivated us to slightly change the Example-Study so that SimStudent always commits to generalizing demonstrations, regardless of whether the step can be explained with the existing production rules. We defined a new learning algorithm, called *Diligent Learning*, as follows: the author provides a demonstration on every step and SimStudent attempts to learn skills each time a step is demonstrated.

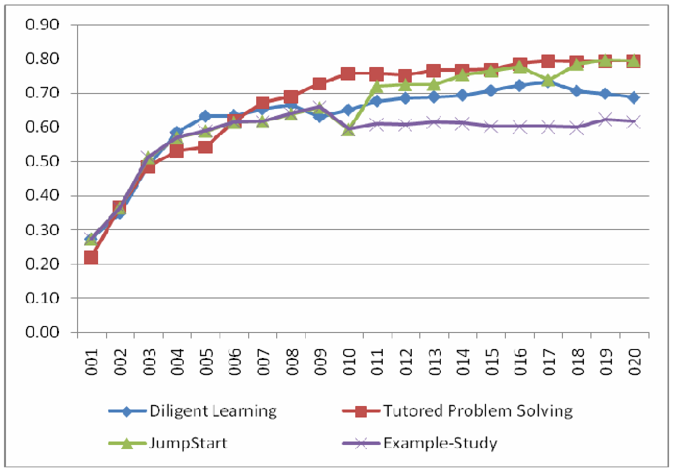
Another alternative strategy was motivated by the observation that while Example Study is resilient to hidden false negatives and has more flexibility to provide diverse demonstrations, Tutored Problem-Solving has the advantage of getting explicit negative feedback, which leads to better Precision scores. We wondered if it would be possible to obtain both advantages by simply combining these two strategies, for instance by starting the authoring process using Example-Study and then switching to Tutored Problem-Solving after a certain number of problems are demonstrated. We call this fourth learning strategy the *Jump Start* strategy. For the

Jump Start condition, we had SimStudent perform Example-Study on the first ten training problems, and then switch to Tutored Problem-Solving.

Using the same training and test problems, we ran an additional study with the Diligent Learning and the Jump Start strategies. Fig. 12 shows the average Step scores, and Fig. 13 shows the average Precision scores. As expected, Diligent Learning was better than Example-Study. However, it was not better than Tutored Problem Solving – it suffered from the same weakness as the Example-Study, i.e., the lack of explicit negative feedback. As can be seen in the graph, Jump Start showed the exact same improvement on the first ten training problems, and then improved its performance once the learning strategy was switched. However, Jump Start ended up with the same performance level as Tutored Problem-Solving.

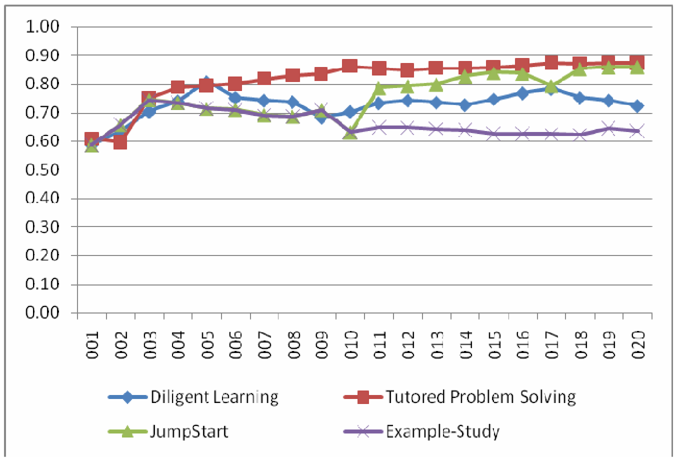
As for the Recall score, both Diligent Learning and JumpStart showed almost identical progress as Example-Study.

This observation also supports the claim that Tutored Problem-Solving is the best authoring strategy in terms of the quality of the cognitive model learned.



For JumpStart, SimStudent performed Example-Study on the first ten training problems, and then switched to Tutored Problem-Solving.

Fig. 12. Average Step scores including Diligent Learning and Jump Start



Precision scores calculated based on the same data as shown in Fig. 12.

Fig. 13. Average Precision scores including Diligent Learning and Jump Start

Impact of Negative Feedback

Another interesting finding is the impact of the negative feedback in guiding the learner to appropriate generalizations. *Making errors and getting feedback on them is crucial for successful authoring with SimStudent.* This is why Tutored Problem-Solving was superior to other learning on the Precision score.

This is indeed a very interesting finding in the current study. Despite the importance of the negative examples, programming by demonstration in most cases only produces positive examples, or at best demonstrates a positive examples on a particular concept that serve as negative examples for other concepts (i.e., the implicit negative examples in the current study).

Kosbie and Myers (1993) emphasized the issue of *program invocation* in the shared common structure of programming by demonstration. That is, programming by demonstration must not only be to create a program (i.e., to learn "what to do"), but also to teach the learning agent when to invoke the program – "otherwise, of what use is the program? (p.424)."

We further emphasize the importance of feedback about incorrect program execution in providing *explicit* negative examples. Interactive Machine Learning (Fails & Olsen, 2003) is a good example of successful application of programming by demonstration where the learning agent can acquire negative examples explicitly through program invocation. Shapiro (1982) also implemented the Model Inference System in a natural way to let the system run a program to find "conjectures that are too strong," meaning let the system makes errors, so that the user can provide appropriate demonstration to motivate the system to debug the error.

It is also interesting to note that the number of negative examples is not necessarily a predictor of a better learning outcome. For Example-Study, when a step is demonstrated on a particular skill, that instance of demonstration not only becomes a positive example of the target skill, but also becomes an implicit negative example of *all other skills*. The study showed that this type of negative example is not as useful as the explicit negative examples obtained by feedback on an incorrect rule application.

Interleaving Tutored Problem-Solving and Example-Study

Even though Tutored Problem-Solving has advantage of detecting false rule applications, there is still the issue of detecting *errors of omission* – i.e., knowing when SimStudent fails to apply an appropriate rule at the right time. The author should not be satisfied simply by observing SimStudent performing all steps correctly; rather, the author also must pay attention to whether all *expected* rules are applied.

Here is another reason why Tutored Problem-Solving is the preferred strategy for authoring—it provides a natural way to combine a formative assessment and necessary remediation. When the author noticed that SimStudent does not perform an expected skill at an expected situation at all, then the author can *demonstrate* that particular skill for SimStudent. Thus, we conjecture that interleaving Tutored Problem-Solving and Example-Study would be continued until further improve the quality of an expert model authored with SimStudent. Further studies are necessary to test the efficiency of interleaving strategies.

CONCLUSION

SimStudent is a learning system that has been embedded into CTAT (Cognitive Tutor Authoring Tools). While CTAT allows domain experts to construct a limited type of cognitive tutor (called an *Example-Tracing Tutor*), CTAT in combination with SimStudent allows authors to build complete intelligent tutoring system with only the cost of writing background knowledge for the target task domain. Authors create an ITS system by teaching the SimStudent how to solve problems. The entire authoring process is very natural, because SimStudent is taught by using the student interface, which is built with CTAT's built-in graphic user interface builder.

Writing feature predicates and operators, the background knowledge to compose expert models, can be expensive, but this knowledge can often be re-used for different tasks within a domain; however, further support for authoring background knowledge would be necessary to make the entire intelligent authoring environment both domain-independent and completely free of programming effort.

Here we explored alternative strategies for teaching SimStudent in this setting. We have showed that when authoring an expert model for an ITS with SimStudent, the Tutored Problem-Solving strategy better facilitates authoring both in the quality of the expert model generated and the time spent for authoring. The most interesting finding is that to learn a better expert model, SimStudent needed to make errors by proactively applying skills from the learned model on problems, and getting negative feedback on the errors made – simply "studying" demonstrations of problem-solving activity produced by an expert led to less accurate models. The technical explanation for this is that accumulating appropriate negative examples is a necessary for successful learning.

REFERENCES:

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *Journal of the Learning Sciences, 4*(2), 167-207.

Blessing, S. B. (1997). A programming by demonstration authoring tool for model-tracing tutors. *International Journal of Artificial Intelligence in Education, 8*, 233-261.

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

Fails, J. A., & Olsen, D. R., Jr. (2003). Interactive machine learning. In *Proceedings of iui2003* (pp. 39-45). Miami Beach, FL: ACM.

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

Kosbie, D. S., & Myers, B. A. (1993). Pbd invocation techniques: A review and proposal. In A. Cypher (Ed.), *Watch what i do: Programming by demonstration* (pp. 423-431). Cambridge, MA: MIT Press.

Lau, T., Wolfman, S. A., Domingos, P., & Weld, D. S. (2003). Programming by demonstration using version space algebra. *Machine Learning, 53*(1-2), 111-156.

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.

Mitrovic, A., Martin, B., & Suraweera, P. (2007). Intelligent tutors for all: The constraint-based approach. *IEEE Intelligent Systems, 22*(4), 38-45.

Muggleton, S., & de Raedt, L. (1994). Inductive logic programming: Theory and methods. *Journal of Logic Programming, 19-20*(Supplement 1), 629-679.

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.

Shapiro, E. Y. (1982). *Algorithmic program debugging*. Cambridge, MA: MIT Press.

Shute, V. J., & Psotka, J. (1994). *Intelligent tutoring systems: Past, present, and future* (No. AL/HR-TP-1994-0005). Brooks Air Force Base, TX: Armstrong Laboratory.

Suraweera, P., Mitrovic, A., & Martin, B. (2005). A knowledge acquisition system for constraint-based intelligent tutoring systems. In C.-K. Looi (Ed.), *Artificial intelligence in education* (pp. 638-645): IOS Press.