Int J Artif Intell Educ (2015) 25:1–34
DOI 10.1007/s40593-014-0020-1

RESEARCH ARTICLE

Teaching the Teacher: Tutoring SimStudent Leads to More Effective Cognitive Tutor Authoring

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

Published online: 20 May 2014

Abstract SimStudent is a machine-learning agent initially developed to help novice authors to create cognitive tutors without heavy programming. Integrated into an existing suite of software tools called Cognitive Tutor Authoring Tools (CTAT), SimStudent helps authors to create an expert model for a cognitive tutor by tutoring SimStudent on how to solve problems. There are two different ways to author an expert model with SimStudent. In the context of Authoring by Tutoring, the author interactively tutors SimStudent by posing problems to SimStudent, providing feedback on the steps performed by SimStudent, and also demonstrating steps as a response to SimStudent's hint requests when SimStudent cannot perform steps correctly. In the context of Authoring by Demonstration, the author demonstrates solution steps, and SimStudent attempts to induce underlying domain principles by generalizing those worked-out examples. We conducted evaluation studies to investigate which authoring strategy better facilitates authoring and found two key results. First, the expert model generated with Authoring by Tutoring is better and has higher accuracy while maintaining the same level of completeness than the one generated with Authoring by Demonstration. The reason for this better accuracy is that the expert model generated by tutoring benefits from negative feedback provided for SimStudent's incorrect production applications. Second, authoring by Tutoring requires less time than Authoring by Demonstration. This enhanced authoring efficiency is partially because (a) when Authoring by Demonstration, the author needs to test the quality of the expert model, whereas the formative assessment of the expert model is done naturally by

N. Matsuda (✉) · K. R. Koedinger
Human-Computer Interaction Institute, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
e-mail: nmazda@gmail.com

K. R. Koedinger
e-mail: Koedinger@cs.cmu.edu

W. W. Cohen
Machine Learning Department, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA
e-mail: WCohen@cs.cmu.edu

observing SimStudent's performance when Authoring by Tutoring, and (b) the number of steps that need to be demonstrated during tutoring decreases as learning progresses.

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 a cognitive tutor by tutoring a machine-learning agent how to solve target problems. The cognitive tutor is a type of intelligent tutoring system (ITS) with a long-standing proven effectiveness (Ritter et al. 2007) We have developed a machine-learning agent, called SimStudent that learns an expert model of the target task through tutored-problem solving. In this context of intelligent authoring, the human author interactively tutors SimStudent using the exact same tutor interface crafted for the target cognitive tutor (Matsuda et al. 2005a, b; Matsuda et al. 2006).

The expert model is one of the major components of a generic ITS (Shute and Psotka 1994; Wenger 1987). To create an expert model, an author often first conducts a cognitive task analysis to identify pieces of knowledge to solve target problems both correctly and incorrectly (Chi 1997; Gott and Lesgold 2000; Jonassen et al. 1999; Kieras 1988). Cognitive task analysis is time consuming even when the author has substantial knowledge and skills in cognitive modeling. To write an expert model, the author needs to be familiar with AI-programming, for example, in a production-rule description language. Thus, building an expert model for ITS can be a notoriously challenging and time consuming task for all levels of users (Murray 1999, 2003). Therefore, developing tools for authoring an expert model is a crucial AIED research agenda.

We are particularly interested in authoring a type of ITS called a cognitive tutor (Anderson et al. 1995; Koedinger and Aleven 2007). The effectiveness of cognitive tutors is based on their capabilities for providing immediate feedback, context-sensitive hint messages, and individualized problem selection. Techniques called model tracing and knowledge tracing support these tutoring strategies. Model tracing is a heuristic version of plan recognition (Burton and Brown 1982; London and Clancey 1982) that attempts to identify cognitive skills in the expert model that sufficiently reproduce problem-solving steps performed by students (Anderson and Pelletier 1991). Knowledge tracing longitudinally models a student's mastery of individual cognitive skills (Corbett et al. 2000; Koedinger and Corbett 2006). These techniques are domain independent and their behaviors depend on the nature and quality of the expert model. Given an expert model, the model tracing and knowledge tracing techniques automatically maintain a student model, which in turn allows the cognitive tutor to automatically perform adaptive tutoring. Thus, authoring a cognitive tutor comes down to authoring two domain dependent components: (1) the graphical user interface (GUI) called a Tutoring Interface for students to show their work, and (2) the expert model representing skills to be learned with hint and error messages associated to each skill.

The proposed SimStudent authoring system is an extension of an existing suite of software tools called Cognitive Tutor Authoring Tools (CTAT) (Aleven

et al. 2006; Aleven et al. 2009; Koedinger et al. 2003). CTAT provides authors with tools to create a Tutoring Interface using a third party GUI builder with virtually no programming effort. CTAT also provides tools to demonstrate solutions on the Tutoring Interface and store them as model solutions. When integrated into CTAT, SimStudent learns to solve problems (hence generates an expert model) from the author who tutors SimStudent how to solve problems using the Tutoring Interface.

The expert model represents a how-type of knowledge about the domain. We assume that the prospective users of the SimStudent authoring system are domain experts who, by definition, know how to solve problems and can identify errors in solution attempts. Domain experts tend to find it difficult to articulate their knowledge (cf., Clark and Estes 1996 on Cognitive Task Analysis) and may even have an expert blind spot whereby they have incorrect beliefs about student needs (Koedinger and Nathan 2004). Therefore, they may not be able to write an expert model easily. However, domain experts would find it easy to demonstrate how to solve problems. The ambitious research question is then whether or not simply letting the domain expert's tutor SimStudent how to solve problems will result in SimStudent learning an expert model for a cognitive tutor.

One of the notable features of SimStudent is its ability to interactively learn an expert model from tutored-problem solving. In this paper, we shall call this type of interactive programming "programming by tutoring" in contrast to programming by demonstration (Cypher 1993; Lau and Weld 1998; McDaniel and Myers 1998).

Accordingly, we have developed two types of interactions for authoring with SimStudent—Authoring by Tutoring and Authoring by Demonstration. When Authoring by Tutoring, the author interactively tutors SimStudent by performing a tutor role in guided-problem solving that is, by providing feedback and hints. In this context, SimStudent is actively engaged in applying learned productions and receiving feedback on the correctness of production applications. When Authoring by Demonstration, the author simply demonstrates solutions while SimStudent passively generalizes demonstrated solutions without any feedback on the correctness of the generalization.

With the lack of theoretical and empirical implications of the advantages and disadvantages of the two authoring strategies, our primary research question is to ask which authoring strategy is better than the other and why. The goal of this paper is therefore to first introduce the SimStudent technology as an intelligent building block for authoring a cognitive tutor in the CTAT framework. This paper then evaluates the effectiveness and challenges of using SimStudent for intelligent authoring.

The rest of the paper is organized as follows. First, we summarize prior research on authoring both for intelligent tutoring systems in general and more specifically for cognitive tutors. Next, we provide a literature survey on applications of the machine-learning technique for authoring, including programming by demonstration and interactive machine learning. We then provide technical details of SimStudent followed by the evaluation study. We conclude the paper with pros and cons of using SimStudent for intelligent authoring.

Related Research

An Intelligent Tutoring System (ITS) is complex machinery hence authoring an ITS is a challenging and time-consuming task. Different components of ITSs require different techniques to assist authoring. As a consequence, there have been many efforts made to develop authoring tools for specific aspects of ITSs (see, for example, Ainsworth and Grimshaw 2004; Angros et al. 2002; Arruarte et al. 1997; Heffernan et al. 2006; Kiyama et al. 1997; Munro et al. 1997; Murray 1998; Sparks et al. 1999).

Most of these authoring tools are aimed to help authors build a Tutoring Interface and create contents of the tutoring materials. As a consequence, authoring an ITS remains quite challenging for novice authors, because authoring an expert model and instructional strategies still requires rather intensive task analysis and knowledge engineering.

Our approach to using programming by tutoring for intelligent authoring addresses these issues by using the generic framework of cognitive tutors that has a natural separation of the expert model and the instructional strategies. Furthermore, the instructional strategy is natively and domain independently equipped in cognitive tutors. Thus, we anticipate that integrating SimStudent into CTAT would provide a domain generic authoring framework.

Prior Research on Authoring Cognitive Tutors: CTAT

One of the major advantages of CTAT is to help authors build a particular type of cognitive tutors called example-tracing tutors (Koedinger et al. 2004). Example-tracing tutors can provide immediate feedback and just-in-time hint messages. An expert model of an example-tracing tutor is essentially a collection of solutions demonstrated by the author. Therefore, the pedagogical capability of example-tracing tutors is limited to those particular solutions that have been demonstrated.

To improve the extent to which an example-tracing tutor can recognize the diversity of student solutions, a recent version of CTAT allows authors to specify alternative solutions by writing sophisticated pattern matching functions and also by relaxing constraints among the order of the steps to be performed (Aleven et al. 2009). Nevertheless, the generality of the pedagogical capability of example-tracing tutors is still limited compared to cognitive tutors with a well-written expert model.

CTAT provides tools to help authors manually write an expert model. For example, there is an application programming interface for the third party Java editor to directly communicate with the production system interpreter embedded in CTAT—authors can therefore use a sophisticated Java editor to compose production rules. There is also a debugging tool called WhyNot that visualizes the production-rule applications. Nonetheless, manually creating an expert model is still a very challenging task for novice authors. It is thus a reasonable extension to couple a machine learning technology into CTAT to automatically generate an expert model by generalizing the demonstrated solutions.

Applying Machine Learning to Authoring Expert Models

There have been studies on the machine-learning application for authoring ITSs. The Constraint Authoring System (CAS) (Mitrovic et al. 2007; Suraweera et al. 2005) is an

authoring system for constraint-based ITS (Mitrovic and Ohlsson 1999). CAS allows authors to build a set of constraints used for a constraint-based ITS by demonstration. Given domain ontology, CAS generates domain constraints from model solutions demonstrated by the author. Since CAS is designed specifically for constraint-based tutors, most of the CAS technologies are not applicable to authoring cognitive tutors.

Demonstr8 (Blessing 1997) is an example of authoring an expert model by demonstration for an ACT Tutor (Corbett et al. 1995), a predecessor of a cognitive tutor. To overcome the computational complexity of machine learning, Demonstr8 provided a menu-driven vocabulary for authors to describe primitive features to be used to compose the conditionals for productions. Since the learning algorithm used in Demonstr8 implicitly relies on the semantics of the menu-driven vocabulary, the generality of Demonstr8 is limited. In contrast, SimStudent addresses this issue by separating a domain-general learning model from a domain-specific learning model. The domain dependency is implemented as the background knowledge to make sense of the solutions demonstrated by the author. The domain-general learning model is implemented using techniques for inductive logic programing (see the section "Authoring Strategies and Learning Algorithms" for details).

SimStudent builds on a preliminary attempt by Jarvis et al. (2004) to integrate a machine-learning component into CTAT for authoring an expert model by demonstration. Using the Tutoring Interface authored by CTAT, the system developed by Jarvis et al. allows the author to demonstrate solutions. Jarvis et al. have successfully demonstrated a potential contribution of programming by demonstration for intelligent authoring. However, the ability to learn knowledge of when to apply individual production rules was limited, and preconditions of learned productions were relatively ad-hoc. As a consequence, expert models generated by their system tended to be overly general. SimStudent addresses this issue by combining three different learning algorithms each specialized to learn different components of a production rule (see the section "Authoring Expert Model by Tutoring SimStudent" for details).

Interactive Machine Learning for Cognitive Modeling

When authoring an expert model by interactively tutoring SimStudent, SimStudent is engaged in interactive machine learning (Fails and Olsen 2003). One of the key features of interactive machine learning is that the learning system can execute the program learned (i.e., the expert model in the current context) and receive feedback on the results of its performance (Kosbie and Myers 1993). Interactive machine-learning systems proactively apply learned knowledge to solve novel problems (Olsen and Pace 2005) or compose innovative hypotheses (Sammut and Banerji 1986), and then ask users to provide feedback on the correctness.

One of the strengths of interactive machine learning lies in the ability to accumulate feedback from users; in particular, the benefit of negative feedback is well-known (Kosbie and Myers 1993; Olsen and Pace 2005). A previous study using SimStudent to model human learning also showed that interactive learning (i.e., programming by tutoring) outperformed non-interactive learning (programming by demonstration) in terms of the "accuracy" of the expert model learned due to the explicit negative feedback for incorrect production applications (Matsuda et al. 2008).

Research Questions and Hypotheses

We will test two major hypotheses. First, the theory of interactive learning predicts that Authoring by Tutoring is a better authoring strategy than Authoring by Demonstration in terms of the quality of the expert model—the proficiency hypothesis.

Second, as for the speed of authoring, Authoring by Demonstration might be a better strategy than Authoring by Tutoring at the micro level (i.e., tutoring individual problems), because the latter requires additional work, e.g., providing feedback to the steps performed by SimStudent, which is not necessary for Authoring by Demonstration. On the other hand, at the macro level (i.e., the entire process of authoring), Authoring by Tutoring might be quicker, because it would provide the author with an opportunity for formative assessment of the quality of the expert model. The formative assessment would convey to the author the rich information that potentially facilitates authoring, e.g., to select problems that effectively reveal flaws in the expert model. Authoring by Demonstration, on the other hand, requires additional time to test the quality of the expert model—the efficiency hypothesis.

These observations bring us the central research questions for the current paper: Which of the authoring strategies better facilitate authoring? This research question can be further broken down into two specific questions: How accurate is the expert model authored with each strategy? And, how long does it take to author an expert model with each strategy? By investigating these questions, we might also be able to address another important research question: Are there strategy-specific advantages and challenges for authoring a cognitive tutor with SimStudent?

We use accuracy of the expert model and speed of authoring as two dependent variables to measure proficiency and efficiency. Since users' skills and experience are human factors, an experiment measuring accuracy and speed could be very complicated. In this paper, we control human factors by simulating the authoring process and focus on the technological factors—i.e., the quality of the expert model generated and the speed of the machine-learning algorithms invoked by the two authoring strategies (see section "Overview of the Evaluation Study" for details). We also describe a case study to address the efficiency hypothesis.

SimStudent: A Synthetic Student That Learns an Expert Model

Although the SimStudent technology is domain-independent, for the sake of explanation, we use an Algebra Equation Tutor shown in Fig. 1 as an example tutor. In this section, we introduce the Algebra Equation Tutor, and then give a detailed explanation of how SimStudent helps authors build an expert model for the Algebra Equation Tutor.

Example of Cognitive Tutor: Algebra Equation Tutor

In this hypothetical tutor, an equation is represented in two cells, one representing the left-hand side (LHS in Fig. 1) and the other the right-hand side (RHS). A problem to solve is shown on the first row (e.g., $3X+1=X+4$). In this hypothetical cognitive tutor, a single equation-solving step (e.g., transforming $3x+1=x+4$ into $3x=x+3$) is broken down into three observable tutoring steps: (1) entering a basic arithmetic operation

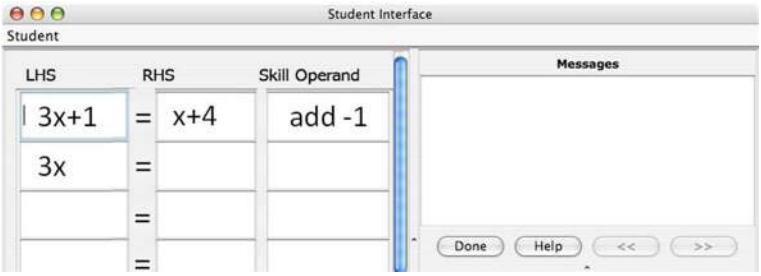

Fig.1 The tutoring interface for the algebra equation tutor. For this tutor, a student is asked to explicitly enter a transformation skill in the third column called Skill Operand. In the figure, the student applied "add −1" to *both sides*, and the *left-hand side* ("3x") has just been entered

(e.g., "add −1") in a column labeled "Skill Operand" to be applied on the both sides of the equation, (2) entering a left-hand side of the new equation in a column labeled "LHS" as a result of applying the basic operation, and (3) entering a right-hand side in "RHS." In this paper, the word "step" is used to mean one of these three tutoring steps unless otherwise specified. In this example, we assume that the hypothetical author decided that the Skill Operand must be specified prior to entering any sides, but the order of entering the sides of the equation is arbitrary. As an example, Fig. 1 shows that a basic operation to add −1 to both sides of $3x+1=x+4$ was entered as the first step, and "3x" has just been entered to the left-hand side as the second step.

In our discussion below, a step to enter a basic arithmetic operation in "Skill Operand" is called a transformation step, and two steps to enter left-and right-hand sides are called type-in steps. In the example shown in Fig. 1, the step to enter "add −1" is an example of the transformation step, whereas the step to enter "3x" is an example of the type-in step.

This hypothetical Equation Tutor, therefore, tutors two types of skills. The skills to perform transformation steps are called transformation skills, and the skills to perform type-in steps are called type-in skills. For example, in Fig. 1, a skill to enter "add −1" is a transformation skill, whereas a skill to enter "3x" is a type-in skill.

Expert Model

In cognitive tutors, an expert model is represented as a set of production rules. Each production rule represents an individual cognitive skill required to perform a particular tutoring step in the Tutoring Interface.

SAI Tuple and the Production Model

A tutoring step is represented as a tuple that contains information about the place selected to perform the step (e.g., the second cell in the first column), the action taken (e.g., entering an expression), and a value that was input for the action (e.g., the string "3x" entered in the cell). These elements individually are called the selection, action, and input. A tuple of [selection, action, input] is called an SAI tuple (Ritter and Koedinger 1996).

A production rule models a particular skill in terms of what, when, and how to generate a particular SAI tuple to perform a step; it roughly means "to perform the step, first look at <what> in the Tutoring Interface and see if a condition <when> holds. If so then do <how>."

The first part of the production rule, representing <what>, specifies a general information retrieval path that selects particular elements of the Tutoring Interface with certain locational constraints like "the second and third cells in the first column of the table above the input." To aid learning the information retrieval path, the author can specify a set of interface elements, as the instances of focus of attention (FoA). Such FoA instances are generalized across multiple examples to form the information retrieval path of the production. The second part of the production rule, representing <when>, is called the precondition. The precondition is a set of conditions held among the instances of the focus of attention, e.g., "the expression in the cell must be polynomial." Lastly, the part of the production rule representing <how> is called the operator function sequence. The operator function sequence consists of a chain of primitive operations that generates the input value in the SAI tuple from the values of the focus of attention.

Together, the information retrieval path and the preconditions compose the if-part (or the condition) of a production rule, whereas the operator function sequence becomes the then-part (or the response) of the production rule.

Background Knowledge

Prior to learning, SimStudent is typically provided with the following background knowledge: (1) a hierarchical representation of the elements on the Tutoring Interface, called the Working Memory Element structure, or WME structure, for short, (2) a set of Boolean functions, called feature predicates, and (3) a set of generic functions, called operator functions.[1]

The WME structure is used to express the topological relations among the elements on the Tutoring Interface. Examples of the interface elements include buttons, text boxes, tables, and labels. The WME structure is hierarchical because, for example, a table contains columns that contain cells.

A feature predicate takes the value within the interface element appearing in the focus of attention as an argument and returns a Boolean value depending on the value of a given focus of attention instance. For example, HasCoefficient ("3x") returns true whereas HasConstTerm ("2x–3x") returns false. Together, the WME structure and feature predicates are used to compose preconditions in the if-part of a production.

Operator functions are used to form an operator function sequence for the then-part of a production. The input values for an operator function can be either the instance of the focus of attention or output values from other operator functions.

Both feature predicates and operator functions are implemented in Java. Currently, SimStudent has a library of 16 feature predicates and 28 operator functions for algebra equation solving as shown in Fig. 2.

[1] Advances of SimStudent are reducing these background knowledge requirements (Li et al. 2011)

Feature predicates	Operator Functions	
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. 2 The list of feature predicates and operator functions for algebra equation domain. Feature predicates are Boolean functions to test if a certain condition holds among given focus of attention. Operator functions transform given values into another value

Authoring an Example-Tracing Tutor with CTAT

Cognitive Tutor Authoring Tools (CTAT) help authors build the Tutoring Interface and demonstrate solutions both correctly and incorrectly using the Tutoring Interface. The demonstrated steps are visualized in the Behavior Recorder as shown in Fig. 3.

The steps in the Behavior Recorder are represented as a directed graph, called a behavior graph. A behavior graph consists of states and edges. An edge has two properties visually associated—one shows an SAI tuple and another shows a skill name. In the example shown in Fig. 3, the edge connecting state1 and state2 has an SAI tuple [dorminTable1_C1R2, UpdateTable, 3x] and the skill name "add-typein." The author can also annotate edges with their correctness as well as the hint and feedback messages.

The behavior graph then becomes an "expert model" for the example-tracing tutor. The example-tracing tutor recognizes a student's action as correct when it matches with any step in the behavior graph.

Authoring an Expert Model by Tutoring SimStudent

This section explains details of authoring an expert model by tutoring SimStudent. We first describe the tutoring interaction between the author and SimStudent, followed by a description of how SimStudent accumulates examples for generalization. We then explain details about SimStudent learning algorithms.

Tutoring Interactions Between the Author and Simstudent

To tutor SimStudent, the author first enters a problem in the Tutoring Interface. SimStudent then attempts to solve the problem by applying productions learned so far. If an applicable production is found, the production application is visualized as a step in the behavior recorder represented as a new state-edge pair like the one shown in Fig. 3. The author then provides flagged feedback on the step performed by SimStudent. The flagged feedback merely tells SimStudent the correctness of the suggested production application. When there are multiple productions that are

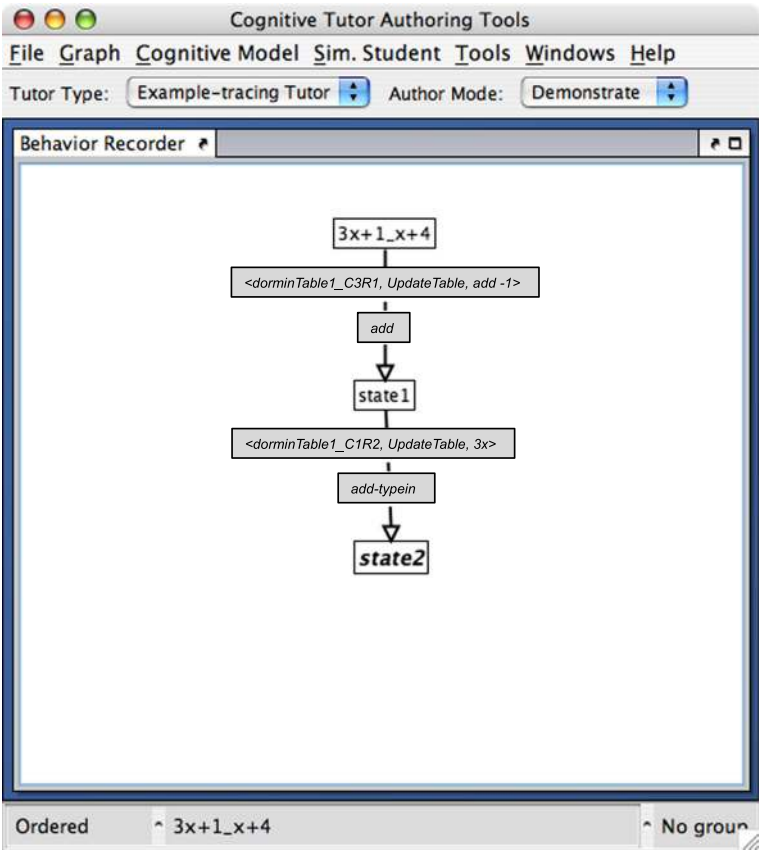

Fig. 3 An example of a behavior *graph*. A behavior graph showing two possible first steps for solving the equation "$3x+1=x+4$." The first step is to enter "add −1"; its skill name is called "add." This step transitions from the initial state (labeled as "$3x+1\_x+4$") into a state called "state1." State 1 corresponds with the interface state when "add −1" has just been entered in Fig. 1. The second step is to enter "3x", and its skill name is called "add-typein

applicable, SimStudent shows the author all corresponding production applications one after another, and obtains yes/no feedback on each.

If no correct production application is found, then SimStudent asks the author what to do next. The author then demonstrates the next step in the Tutoring Interface. To demonstrate a step as a hint, the author first specifies the focus of attention by double-clicking the corresponding interface elements (e.g., a LHS cell and a RHS cell). The specified elements become the FoA instances and are highlighted as shown in Fig. 4. The author then takes an action upon a selection with an appropriate input value. In Fig. 4, the author entered the string "add −1" in the Skill Operand cell with "3x+1" and "x+4" as the FoA instances. In this case, the SAI tuple is [Skill Operand, UpdateTable, "add −1"]. The step demonstrated by the author is visualized immediately in the behavior graph. Finally, the author specifies the skill name by clicking on the newly added edge of the behavior graph. A small dialogue box appears to enter a skill name.

Generally speaking, authors could model a single observable step (i.e., an edge in the behavior graph) with a chain of production-rule applications. However, when using

SimStudent to author an expert model, SimStudent generates a single production rule per each observable step.

Accumulating Positive and Negative Examples

When a step is demonstrated by the author as a hint for a particular skill S with the FoA instance foa and an SAI tuple sai, the pair <foa, sai> becomes a positive example of the skill *S*. At the same time, the instance of the skill application <foa, sai> also becomes an implicit negative example for all other already demonstrated skills.

For example, at the situation illustrated in Fig. 4, "add −1" has been just demonstrated with the focus of attention being "3x+1" and "x+4." Assume that the author labeled this step as 'add.' Then the pair E_1=<[3x+1, x+4], [Skill Operand, UpdateTable, add −1]> becomes a positive example of the skill 'add.' Assume that the author had already demonstrated a skill for multiplication prior to demonstrating the step shown in Fig. 4, and labeled it as 'multiply'. The pair E_1 also becomes an implicit negative example for the skill 'multiply'.

An implicit negative example for a skill *S* may later become a positive example, if the same FoA instance is eventually used to demonstrate the skill *S*. This indicates the following to SimStudent: "Given a set *U* of skills already demonstrated, apply skill *S* in a given situation, but do not apply any previously demonstrated skills in *U* other than *S* unless otherwise instructed." For example, assume that SimStudent had already been trained on the same equation "3x+1=x+4" prior to the situation illustrated in Fig. 4 and was instructed to "subtract 1" for it. Also assume that SimStudent had already been exposed to the skill 'add' when the step to "subtract 1" was demonstrated (i.e., the hypothetical situation illustrated in Fig. 4 is the second time for the skill 'add' to be demonstrated). Under this scenario, when the step for "subtract 1" was demonstrated, the pair E_2=<[3x+1, x+4], [Skill Operand, UpdateTable, subtract 1]> was an implicit negative example for skill 'add.' Then, when "add −1" in Fig. 4 is demonstrated, the pair E_2 is withdrawn from the negative example for skill 'add,' and the pair E_3=<[3x+1, x+4], [Skill Operand, UpdateTable, add −1]> becomes a positive example for skill 'add.'

When a new positive or negative example is added for a particular skill for the first time, SimStudent creates a new production rule. If a production rule for the demonstrated skill already exists, then SimStudent modifies the existing production rule by taking all positive examples into account, while assuring that the resulted production

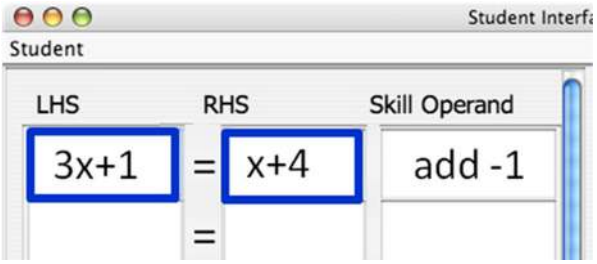

Fig. 4 Focus of attention highlighted on the Tutoring Interface. Particular elements on the Tutoring Interface are highlighted as the FoA instance. In this figure, the author double clicked on the cells with the values "3x+1" and "x+4" as FoA, and then entered "add −1

rule does not yield a step that matches any of the negative examples. The focus of attention in both positive and negative examples are used for learning preconditions, but only positive examples are used for generalizing information retrieval path (based on FoA) and operator function sequences (based on FoA and SAI). See the next section for how production rules are learned from FoA and SAI tuples.

One major difference between the two authoring strategies is that only Authoring by Tutoring provides explicit negative examples, which are generated when the author provides negative feedback on SimStudent's incorrect suggestion. The negative feedback on the incorrect production application tells SimStudent "do not apply this particular skill in this particular situation." Unlike implicit negative examples, explicit negative examples permanently remain as negative examples.

Notice that the author can demonstrate incorrect steps to have SimStudent learn incorrect (or "buggy") production rules. CTAT also allows the author to demonstrate steps incorrectly. The essential difference is that CTAT does not generalize the demonstrated errors whereas SimStudent learns underlying principles to make errors. Learning incorrect productions from author's purposeful incorrect demonstrations is different from learning incorrect productions due to induction errors from correct demonstrations. When the author purposefully demonstrates incorrect steps, they become positive examples for targeted buggy skills. SimStudent, of course, does not recognize incorrect steps as incorrect. Therefore, the author has to tag incorrect production rules accordingly (by using CTAT, for example).

SimStudent Learning Algorithm

Three machine-learning techniques are used to learn production rules. First, to learn the information retrieval path of a production rule, the algorithm must find a generalization that works across all instances of FoA appearing in the positive examples. This generalization is guided by a version space (Mitchell 1997) predefined by the hierarchical representation of the interface elements. For example, suppose the Tutoring Interface has one table with two columns and five rows. An interface element that is "the third cell in the first column" may be generalized to "any cell in the first column," which may be further generalized to "any cell in any column."

Second, the precondition is learned by finding features that are common in positive examples but do not appear in negative examples. This is done by inductive logic programming (Muggleton 1999; Muggleton and de Raedt 1994) in the form of open source software FOIL, [2] First Order Inductive Learner (Quinlan 1990). Given a language to represent the conditional features to compose hypotheses, FOIL inductively generates hypotheses that are consistent with demonstrated examples. In our application, an individual hypothesis represents the applicability of a particular skill. Using an example shown in Fig. 4, suppose that the author labeled the skill application as 'add-constant.' In this case, the target hypothesis to be learned is add-constant (X,Y) that describes when to apply the skill "add-constant" with the instances of focus of attention X and Y. In this example, add-constant (3x+1,x+4) becomes a positive example for add-constant (X,Y).

[2] http://www.rulequest.com/Personal/

The hypothesis on the applicability of the skill add-constant (X,Y) is described with the feature predicates given as the background knowledge in a form of the Prolog-like clause such as: $\text{add-constant}(X, Y) \text{ :- } \text{isPolynomial}(X), \text{isPolynomial}(Y)$.

SimStudent then converts the body of the hypothesis into the if-part conditionals of the production for the skill add-construct. In this case, the production suggests applying the skill add-constant when both left and right sides of the equation are polynomial.

To achieve this goal, the instances of focus of attention (i.e., "3x+1" and "x+4") are fed into all feature predicates given to SimStudent to make positive and negative examples for those feature predicates, which are called literals in FOIL terminology. For example, since the Boolean function isPolynomial (3x+1) returns true, isPolynomial (3x+1) becomes a positive example of the feature predicate isPolynomial (X), whereas isNumeratorOf (3x+1,x+4) becomes a negative example of the feature predicate isNumeratorOf (X,Y). Given the positive and negative examples for the target hypothesis and the literals, FOIL generates a hypothesis.

The preconditions acquired from just a few examples are typically overly general and thus the resulting production rule will sometimes fire inappropriately. For example, the precondition above inappropriately applies to the equation 4−7=2x+5 and the acquired production rule incorrectly suggests to "add 7" in this context. This type of over generalization (and over specialization as well) is the nature of inductive learning. We discuss in the evaluation section how different learning strategies deal with induction errors differently.

Third, the operator function sequence in the then-part of a production rule yields a demonstrated action (an SAI tuple) from a given input (FoA instances) for all the stored positive examples (i.e., all <foa, sai> pairs). A straightforward iterative-deepening depth-first search is used to do this generalization. This approach is a generalization of the Bacon model of scientific discovery of laws, in the form of mathematical functions, from data of input–output pairs (Langley et al. 1987). Because our operator function sequence learner uses iterative-deepening, it prefers the simplest explanation of the observed examples (shortest function composition) and currently does not incorporate any bias based on past experience (e.g., "it is more likely to add a term to a polynomial expression than to divide").

Pseudo-Code Representation of Interactive and non-Interactive Learning Strategies

Figure 5 shows a pseudo-code representing the learning algorithm to generalize a step demonstrated with a focus of attention foa and an SAI tuple sai for a skill skill. When a step is demonstrated, corresponding sets of positive and negative examples are updated. The function remove-negative-example (skill, foa, sai) uses this positive example to override a prior implicit negative example, if there is one, for the same skill and <foa, sai> pair. The if-part of a production rule (i.e., the information retrieval path and preconditions) is learned with the functions generalize-information-retrieval-path () and induce-preconditions (). The then-part of the production rule (i.e., the operator function sequence) is learned with search-for-operator-function-sequence ().

Both Authoring by Tutoring and Authoring by Demonstration call the function generalize-action-performed (). Figure 6 shows pseudo code for Authoring by Tutoring. It first gathers all applicable productions by calling gather-production-activations (). The function apply-production (skill) applies the production skill, and returns the

```
generalize-action-performed(foa, sai, skill) {
        add-positive-example(skill, foa, sai);
        /* Implicit negative example */
        foreach existing-skill in previously-learned-skills do
             add-negative-example(existing-skill, foa, sai);
        /* Eliminate incorrect implicit negative example, if any */
        remove-negative-example (skill, foa, sai);
        positive-examples ← get-all-positive-examples-for(skill);
        negative-examples ← get-all-negative-examples-for(skill);
        generalize-information-retrieval-path(positive-examples);
        induce-precondition(positive-examples, negative-examples);
        search-for-operator-function-sequence (foa, sai, positive-examples);
}
```

Fig. 5 Pseudo code to generalize a step demonstrated. When a step is demonstrated for a particular skill, a positive example for the skill application is generated. The demonstration also becomes an implicit negative example for all other existing skills. A production rule is then generated by combining an information retrieval path, preconditions, and the operator function sequence

feedback from the author. In addition to the feedback, the function also returns the focus of attention and the SAI tuple of the production skill.

Figure 7 shows pseudo-code for Authoring by Demonstration. SimStudent first checks whether it can already perform the step demonstrated by searching for a previously learned production that yields the same SAI tuple that has been demonstrated. If no such production is found, then SimStudent attempts to learn the skill demonstrated by invoking the function generalize-action-performed ().

Overview of the Evaluation Study

We conducted two evaluation studies. As discussed in the section "Research Questions and Hypotheses," we focused on the theoretical aspect of the effectiveness of SimStudent for intelligent authoring. To control human factors, we simulated the entire authoring process with Carnegie Learning's Cognitive Tutor™ technology (Ritter et al. 2007). That is, we used the Cognitive Tutor™ technology as a simulated author (see the section "Method" in "The Simulation Study" for details). [3]

In our first study, the Simulation Study, we tested the proficiency and efficiency hypotheses by comparing Authoring by Demonstration and Authoring by Tutoring both simulated with the Cognitive Tutor™ technology. Proficiency was operationalized as the quality of an expert model learned by SimStudent. Efficiency was measured as the computational time required by the machine-learning algorithm.

Although the Simulation Study provides us with an insight into the efficiency of the proposed technology, the simulation also involves extraneous factors such as the overhead for the inter-process communication that is not a part of the actual authoring process. For example, for Authoring by Tutoring, a SimStudent component intensively communicates with a Cognitive Tutor Algebra I™ component that requires the non-

[3] This is a nice example of intelligent systems interacting with each other!

```
authoring-by-tutoring(problem) {
       until is-problem-solved(problem) do {
              step-performed-correctly ← false;
              all-applicable-skills ← gather-production-activations();
              foreach applicable-skill in all-applicable-skills do {
                     <feedback, foa, sai> ← apply-production(applicable-skill);
                     if (feedback = "correct") {
                            add-positive-example(applicable-skill, foa, sai);
                            remove-negative-example (applicable-skill, foa, sai);
                            step-performed-correctly ← true;
                     } else {
                          /* explicit negative example */
                            add-negative-example(applicable-skill, foa, sai);
                     }
              }
              if (step-performed-correctly = false) {
                     < modelFoa, modelSai, modelSkill> ← askHint(step);
                     /* positive as well as implicit negative examples would be accumulated */
                     generalize-action-performed (modelFoa, modelSai, modelSkill);
              } } }
```

Fig. 6 Pseudo code for authoring by tutoring. SimStudent is given a problem to solve. For each step, SimStudent suggests a possible production application one at a time, and the author provides yes/no feedback. A correct production application becomes a positive example and an incorrect rule application becomes an explicit negative example. When no correct production application is found, SimStudent asks the author for help and the author demonstrates the step

trivial overhead time, which does not happen for Authoring by Demonstration. To address this issue while controlling as much of human factors as possible, we conducted the second study as a case study mentioned below to evaluate the time spent on authoring with a single human subject who is a co-author of this paper, and thus knows the task and how to operate the system well.

In the second study, the Authoring Efficiency Case Study, the participant, who is an author of this paper, authored an expert model twice using SimStudent once for each authoring strategy. The participant followed specific instructions on how to interact with SimStudent to minimize user specific variance.

For both studies, we used the Algebra Equation Tutor shown in Fig. 1 as an example cognitive tutor to be authored. In the Case Study, we provided the participant with the

```
authoring-by-demonstration(focus-of-attention, action-performed, skill-name) {
       if (not model-trace(action-performed)) {
              generalize-action-performed (    focus-of-attention,
                                               action-performed,
                                               skill-name );
       }
}
```

Fig. 7 Pseudo code for authoring by demonstration. When the author demonstrates a step, SimStudent first attempts to match (i.e., model-trace) the step with existing productions. If the model-tracing fails, then SimStudent invokes the generalize-action-performed procedure to learn the skill that has been just demonstrated

Tutoring Interface so that authoring time was measured only for creating an expert model. Since the cognitive tutor does not generate incorrect solutions, the Simulation Study only included correct demonstrations, and so did the Case Study. The following sections describe the studies and their results.

The Simulation Study

Method

We used two study conditions to compare two authoring strategies—Authoring by Demonstration and Authoring by Tutoring. For each authoring strategy, SimStudent was trained on the same training problems in the same order. There were five sets of training problems where each set contained 20 training problems (i.e., equations). In other words, SimStudent was trained five times with different problems. All five training sets were designed to teach ten skills—five transformation skills (add, subtract, multiplication, division, and combine-like-terms) and five corresponding type-in skills (add-typein, subtract-typein, etc.). For the sake of discussion, we define an authoring session as an attempt to author an expert model with a single training set (i.e., 20 training problems).

To test the accuracy of the expert model generated by SimStudent, a single set of ten test problems was used for all authoring sessions—each time SimStudent completed training on a single training problem, SimStudent solved the 10 test problems and the results were recorded. This means that an expert model was tested 20 times (on the same test problems) during a single authoring session.

For Authoring by Demonstration, all demonstrations were pre-recorded. These demonstrations were extracted from empirical data collected from an algebra classroom study conducted by Booth and Koedinger (2008) The study was done at a LearnLab[4] participating high school as part of a normal class activity. In the algebra study, high-school students were learning how to solve equations using Carnegie Learning's Cognitive Tutor Algebra I™ (or, the Tutor, hereafter). The students' interactions with the Tutor were logged and stored into an open data repository, DataShop,[5] maintained by Pittsburgh Science of Learning Center (Koedinger et al. 2010; Koedinger et al. 2008). We only extracted students' correct solutions and used them as pre-recorded solutions as if a hypothetical author demonstrated them. The five training sets and a set of test problems were randomly selected from the pool of students' correct solutions. When SimStudent was trained with Authoring by Demonstration, SimStudent did not actually interact with the Tutor, but merely read the pre-recorded solutions.

For Authoring by Tutoring, we had SimStudent directly tutored by the Tutor with the same training problems used for Authoring by Demonstration, but solutions were determined by actual interactions between SimStudent and the Tutor. That is,

[4] LearnLab courses involve pre-arranged agreements with schools to run learning experiments in real-world educational settings instrumented for process and outcome data collection. The PSLC is funded by the National Science Foundation award number SBE-0836012. See www.learnlab.org.
[5] www.learnlab.org/technologies/datashop

knowledge tracing was not used and the Tutor did not adaptively select training problems. We implemented an application programing interface (API) for SimStudent to communicate with the Tutor. When SimStudent asked for a hint, the API returned the most specific hint from the Tutor. Since the most specific hint is a precise instruction for what to do next in the form of an SAI tuple, it is practically equivalent to demonstrating a step. When SimStudent performed a step, an SAI tuple as the result of a production rule application was sent to the Tutor to be checked, and the API responded with the correctness of the suggestion.

In both conditions, SimStudent was trained with the solutions created (for Authoring by Tutoring) or guided to create (for Authoring by Demonstration) by a pre-engineered expert model embedded in Cognitive Tutor Algebra I™. Because the Tutor's expert model has been tested and proven to be valid through numerous field trials (Ritter et al. 2007), we argue that using those solutions is a fair simulation for human authoring when errors made by authors are controlled.

Evaluation Metrics

The quality of a learned expert model was measured in terms of the accuracy of solutions made by the expert model when solving the test problems. Since SimStudent does not learn strategic knowledge to select a production among equally available productions (i.e., a conflict resolution strategy), the "accuracy" in this context should reflect both correct and incorrect production rule applications. We therefore evaluated the correctness of all applicable production rules for each step of each test problem.

To be precise, we computed a conflict set for each step, which is a set of production rules whose if-part conditions hold. The correctness of each production rule application in the conflict set was then evaluated using the expert model embedded in Cognitive Tutor Algebra I™. SimStudent's performance on a step was coded as "correct" if there was at least one correct production rule application in the conflict set. Otherwise, the performance on the steps was coded as "missed."

A dependent variable, called the problem score was defined as follows. First, for each step, the step score is defined to be zero if the step was missed. Otherwise, the step score is a ratio of the number of correct production rules to the total number of production rules in a conflict set. For example, if there were 1 correct and 3 incorrect production rules, then the step score is 0.25. The step score ranges from 0 (no correct production applicable at all) to 1 (at least one production applicable and all applicable productions are correct). The step score is the probability of SimStudent performing a step correctly on the first attempt, assuming it performs a step by a uniform random selection of the production instances in the conflict set. More importantly (for the practical goal of using the resulting expert model in a cognitive tutor), the step score indicates the power of recognizing student actions as correct when used for model tracing. For example, an expert model with a low step score tends to grade incorrect student actions as correct. The problem score was then computed as the average of the step score across all steps in a test problem. The problem score also varies from 0 to 1.

We also computed the recall score. The recall score is defined for each test problem as a ratio of the number of steps performed correctly to the total number of steps. The recall score does not take incorrect production applications into account—a step is coded as "correct" if there is at least one correct production rule application regardless

of the number of incorrect production rule applications. For example, if a problem had 5 steps for which SimStudent correctly performed two steps, then the recall score is 0.4. The recall score indicates the likelihood of recognizing students' correct steps as correct (when used for model-tracing)—i.e., an expert model with a low recall score would more likely produce false-negative feedback.

Results

Overall Learning Performance

Figure 8 shows the average problem score for the ten test problems aggregated across five different training sequences. The X-axis shows the order of training problems in a training sequence.

In both conditions, the performance improved almost equally on the first six training problems—a chance to perform a step correctly at the first attempt increased in both authoring strategies as SimStudent was trained on more problems. However, starting at the sixth training problem, Authoring by Demonstration stopped improving. Authoring by Tutoring, on the other hand, kept improving until the 16th training problem. After training on 20 problems, the average problem score for Authoring by Tutoring was 0.80, and 0.62 for Authoring by Demonstration. The difference was statistically significant, $t(69)=7.94$, $p<0.001$. These results suggest that Authoring by Tutoring generated more correct and/or less incorrect production rules than Authoring by Demonstration.

Figure 9 shows average recall scores for the ten test problems aggregated across five training sequences. Both authoring strategies showed improvement over the training problems. At the beginning, Authoring by Tutoring showed slightly inferior performance on the recall score, but at the end, the difference was not statistically significant; $t(49)=1.50$, $p=0.13$.

Putting the above two findings on step and recall scores together, it is evident that the Authoring by Tutoring is a better strategy for authoring an expert model, because SimStudent learns fewer incorrect productions while learning correct productions equally well as Authoring by Demonstration.

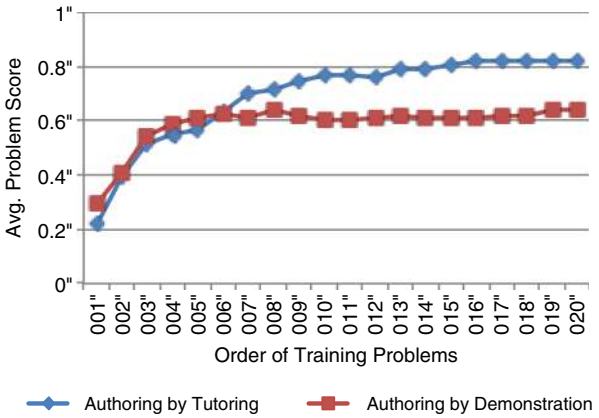

Fig. 8 Average problem scores. *X*-axis shows the order of training problems. *Y*-axis shows the average Problem score on the ten test problems aggregated across the five "authors

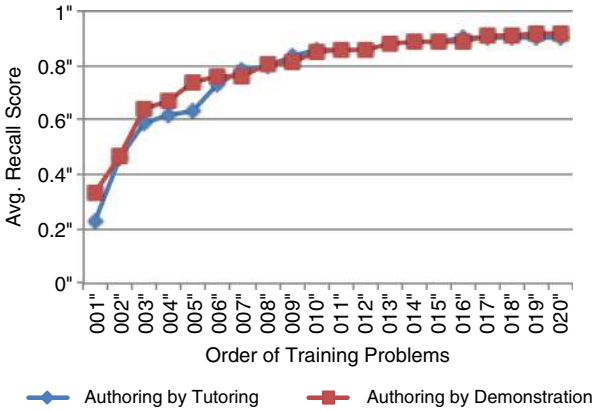

Fig. 9 Average recall scores. *X*-axis shows the number of training problems already executed at the time that the Recall Score (*Y*-axis) was calculated

Error of Commission

To understand the types of errors made on the test problems and to see if there was any difference in errors made by the different authoring strategies, we conducted a qualitative analysis of the errors that SimStudent made on the test problems at the end of the training sequence (i.e., after the 20th training problem).

There were four types of errors observed: (1) No-progress error—a mathematically correct application of a transformation skill, but it does not make the equation any closer to a solution. (2) Inconsistent Transformation error—an application of a transformation skill without completing the previous type-in steps. (3) Inconsistent type-in error—typing in an incorrect expression as a result of a correctly applied transformation. (4) Wrong type-in error—an incorrect execution of a type-in step.

An example of the no-progress error is to "subtract 2x" from $2x+3=5$. This step is mathematically valid, but the resultant equation $3=-2x+5$ requires at least two transformation skills and four type-in steps, which is the same as solving the original equation $2x+3=5$.

An example of the inconsistent transformation error is shown in Fig. 10. In this example, the author entered "divide 3" into the rightmost cell on the second row when the middle cell (right-hand side of the equation) is still empty. A type-in step to enter "3" for the right-hand side has been incorrectly skipped.

An example of the inconsistent type-in error is shown in Fig. 11. Suppose that SimStudent correctly applied "add −1" for "$3x+1=4$," and then typed in "3x" correctly on the left-hand side. SimStudent then typed in "5" to the right-hand side by incorrectly applying subtract-typein—subtracting (instead of adding) -1 from 4, which yields 5.

An example of the wrong type-in error is shown in Fig. 12. In this example, SimStudent correctly applied a production for combine like terms (CLT) as a transformation. However, it incorrectly applied a wrong version of CLT-typein that simply copies an expression from the cell above. This production could have been learned with an equation, say, "$2x=2+4$" to combine like terms 2 and 4 on the right-hand side. Since the skill CLT-typein applies for both sides, SimStudent needs to learn to combine like terms on the right-hand side, but simply copy a term (i.e., 2x) on the left-hand side.

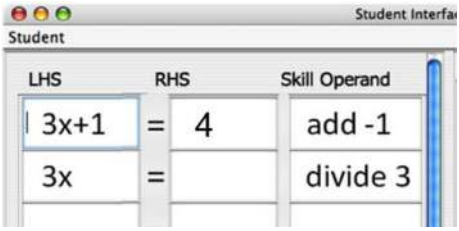

Fig. 10 An example of an inconsistent transformation error. Entering a skill-operand "divide 3" without completing previous type-in steps (i.e., entering "3") is an example of an inconsistent transformation error

An inconsistent transformation could be mathematically reasonable. For example, seeing a monomial variable term on the left-hand side (e.g., 3x) may be sufficient to divide both sides of the equation with the coefficient of the variable term. Similarly, a no-progress error, by definition, is to apply a mathematically valid operation. We codify those steps as incorrect steps, because Carnegie Learning Cognitive Tutor Algebra I™, which we used to automatically classify incorrect production applications, does not allow students to make those steps.

Table 1 shows the frequency of each type of error averaged across five training sequences, counting all conflicting production applications for each step on each test problem. Overall, there were notably fewer errors of all types observed for Authoring by Tutoring than Authoring by Demonstration.

Inconsistent transformation errors (ITR in Table 1) and wrong type-in errors (WT) were observed only for Authoring by Demonstration. Inconsistent type-in errors (ITY) almost exclusively happened only for a particular type-in skill, the skill CLT-typein—a skill to perform a type-in step that follows a step to combine like terms.

When we investigated the cause of errors by reading incorrect production rules learned by SimStudent, we noticed that all errors were due to incorrect preconditions with an inappropriate if-part. For example, an overly generalized production for subtraction applies to any equations with a polynomial expression on the left-hand side. Such an overly general production incorrectly subtracts a variable term from, say "$ax+bx=c$," instead of an equation with a left-hand side that has a polynomial with both constant and variable terms, e.g., "$ax+bx=c$."

It appeared that CLT-typein is particularly hard to learn correctly. The skill CLT-typein was demonstrated for two different situations—one for actually combining like terms and the other one for a straight copy from the cell above. For example, when CLT is applied to an equation "$3x+2x=10$," the value to be typed in to the left-hand side is

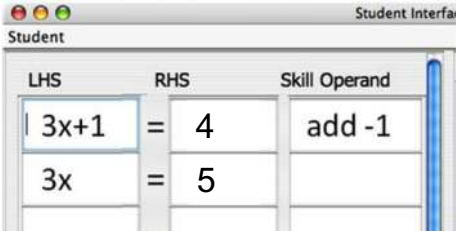

Fig. 11 An example of an inconsistent type-in error. Entering "5" as a type-in for "add −1" to the *right-hand side* (RHS) by incorrectly applying a production for subtract-typein (i.e., subtracting −1 from 4) is an example of an inconsistent type-in error

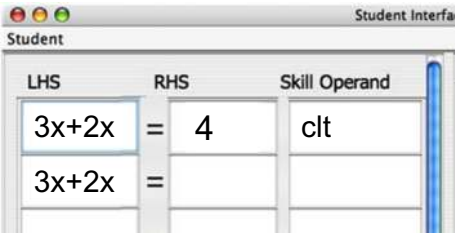

Fig. 12 An example of a wrong type-in error. Entering "3x+2x" as a type-in for "CLT" to the *left-hand side* (LHS) by incorrectly applying a disjunct for the skill clt-typein, which should have been applied to the *right-hand* side to copy the expression '4' in the cell above

"5x," which requires an actual combination of like terms, but the right-hand side is "10," which is a straight copy for the original equation. When Cognitive Tutor Algebra I™ tutored SimStudent about the type-in steps for CLT, these two kinds of type-in steps are both labeled as 'CLT-typein.' Thus, CLT-typein requires learning disjunctive productions. As a consequence, the skill CLT-typein caused both types of error—the wrong type-in error by applying an incorrect disjunct that yields an incorrect value to type in, as well as the inconsistent type-in error that applies when the immediate transformation was not CLT. For both cases, it is a flaw in the production precondition that causes an incorrect step.

The wrong type-in error could have been avoided if the author distinguished the skill to actually combine like terms from the one to simply copy terms. However, novice authors might not be fully aware of the subtle differences between these two skills. Therefore, this might be a challenging knowledge-engineering problem for novice authors. We will discuss this issue later in the section "Labeling Issues."

The fact that there were no errors caused by incorrect operator function sequences suggests that the domain dependent operator functions used in the study were strong enough to let SimStudent learn appropriate responses correctly within 20 training problems.

Table 1 Frequency of errors for each type. The numbers show the average counts of the errors made by five SimStudents on the test problems when asked to show all possible (conflicting) production applications

Production	Authoring by Tutoring				Authoring by Demonstration			
	NP	ITR	ITY	WT	NP	ITR	ITY	WT
Add	0.5	0	–	–	12.0	3.0	–	–
Subtract	1.8	0	–	–	23.0	7.0	–	–
Divide	0	0	–	–	1.0	10.8	–	–
Multiply	0	0	–	–	0.5	1.8	–	–
CLT	4.0	0	–	–	7.3	0	–	–
CLT-typein	–	–	5.0	0	–	–	30.5	2.8
Subtract-typein	–	–	0	0	–	–	0.58	0

NP no progress error, *ITR* inconsistent transformation error, *ITY* inconsistent type-in error, *WT* wrong type-in error. Note that, by definition, NP and ITY only apply to transformation skills. Likewise, ITY and WT only apply to type-in skills

Why was it so hard to learn preconditions of particular productions correctly? To illustrate incorrect learning of the preconditions, consider the following example for the skill CLT-typein:

IF the goal is to enter an expression on a side *S* (which is either "right" or "left") of an equation, AND

the expression *M* on the same side *S* of the prior equation is a monomial

THEN enter *M* on the side *S* (for the transformed equation)

This production generates a correct CLT-typein to enter the right-hand side for $2x+3x=5$. However, since this production is overly general, it also allows copying an expression even when the preceding transformation is not to combine like-terms (e.g., copying 8 from equation $2x-5=8$ even though transformation is "add 5"), which is an incorrect type-in error.

When validating the data, it turned out that a feature predicate relevant to testing whether the previous transformation skill was CLT or not (implemented as a Boolean function is-skill-CLT ()) was accidentally excluded from the background knowledge given to SimStudent when we conducted the Simulation Study. For all other type-in skills, the precondition is simply asking whether the previous transformation step is to apply a corresponding transformation skill. For example, the feature predicate is-skill-add () returns the Boolean value TRUE if the previous transformation step is to add a term to both sides of the equation. The lack of is-skill-CLT () should have made it more difficult for SimStudent to learn the preconditions of the production for 'CLT-typein.'

SimStudent with Authoring by Tutoring actually learned more elaborated preconditions even without is-skill-CRT () for 'CLT-typein' as shown below:

IF the goal is to enter an expression on a side *S* of an equation, AND

the expression *M* on the same side *S* of the prior equation is a monomial, AND

the previously applied transformation skill is not subtraction, AND

the previously applied transformation skill is not addition

THEN enter *M* in *S*

This production rule has additional preconditions to exclude inappropriate precedent transformation steps. Authoring by Tutoring allows SimStudent to learn the "when" part more effectively than Authoring by Demonstration to compensate the lack of the key feature predicate.

The above observation implies that it is how FOIL learns the precondition of the production that makes Authoring by Tutoring superior to Authoring by Demonstration. Why does Authoring by Tutoring more effectively drive FOIL than Authoring by Demonstration? To answer this question, we counted the number of examples provided to FOIL. Table 2 shows the average number of positive and negative examples

Table 2 Average number of positive and negative examples provided to FOIL. The counts are averaged across five training sequences

	Positive examples	Negative examples	Total	Positive to negative ratio
Authoring by Demonstration	41.5	226.3	267.8	0.18
Authoring by Tutoring	166.3	951.0	1,117.3	0.12
AbyD to AbyT ratio	0.25	0.24	0.24	

generated for each authoring strategy aggregated across five authoring sessions. It turned out that Authoring by Tutoring (AbyT) generated about four times as many positive and negative examples as Authoring by Demonstration (AbyD). Interestingly, the ratio of positive to negative examples is similar for both strategies.

Recall that implicit negative examples are exclusively generated each time a positive example is generated, and, because multiple correct solutions are possible, those implicit negative examples might be incorrect. On the other hand, the negative feedback provided in Authoring by Tutoring always generates "correct" negative examples relative to feedback. We thus hypothesized that Authoring by Tutoring has a higher ratio of correct to incorrect negative examples than Authoring by Demonstration. This hypothesis was supported as shown in Table 3, which shows the number of correct and incorrect negative examples that were classified by a human coder reviewing all the examples. In particular, the average accuracy of negative examples, which is the ratio of correct to all negative examples, is higher for Authoring by Tutoring than Authoring by Demonstration; $M_{AbyD}=0.74$ vs. $M_{AbyT}=0.89$.

The accuracy of negative examples could be increased by either (1) removing more incorrect negative examples by eventually getting replaced with positive feedback or demonstration provided for the same FoA instance, or (2) accumulating more correct negative examples by explicit negative feedback (assuming that authors provide accurate feedback). The analysis of process data showing the interaction between SimStudent and Cognitive Tutor Algebra I™ revealed that this is actually the case. The details follow.

When analyzing the process data, we found that there was an average of 8.4 steps (aggregated across five authoring sessions) demonstrated per problem for Authoring by Demonstration. On the other hand, there was an average of 1.7 steps demonstrated (as a hint) per problem for Authoring by Tutoring. As for feedback, there was an average of 8.6 positive and 7.8 negative feedbacks provided per problem for Authoring by Tutoring. Thus, in average, when Authoring by Tutoring for a sequence of 20 problems, 33.3 positive examples were generated by demonstration, 172 positive examples

Table 3 The number of correct and incorrect negative examples accumulated for 20 training problems averaged across five training sequences. The greater accuracy for authoring by tutoring results from explicit negative feedback that produces explicit negative examples

	Correct negative examples	Incorrect negative examples	Total	Accuracy
Authoring by Demonstration	167.5	58.8	226.3	0.74
Authoring by Tutoring	848.5	102.5	951.0	0.89

were generated by positive feedback, and 155.5 explicit negative examples were generated by negative feedback.

Since there were in average the total of 951.0 negative examples for Authoring by Tutoring (Table 3) and 155.5 of them were explicit negative examples, there should have been 795.5 implicit negative examples provided for Authoring by Tutoring. Compared to Authoring by Demonstration where 26 % of (implicit) negative examples were incorrect (Table 3), there could have been 206.8 (795.5×0.26) incorrect negative examples generated. However, there were only 102.5 incorrect implicit negative examples generated (Table 3). This suggests that Authoring by Tutoring eliminated 104.4 (206.8–102.5, which is about 50 % of) incorrect implicit negative examples that Authoring by Demonstration could hardly eliminate.

What if more positive examples were made available by the author? To answer this question, we modified the algorithm for Authoring by Demonstration by disabling the function call model-trace () shown in Fig. 7. This modification forces SimStudent to always commit to learning regardless of whether the step demonstrated can be explained by an existing production. Therefore, more positive examples were generated by this modification.

We tested the modified version of Authoring by Demonstration using the same training and test problems as the ones used in the Simulation Study. This modification resulted in better problem scores than the original version of Authoring by Demonstration, 0.70 vs. 0.61 on the 20th training problem. However, it was not better than Authoring by Tutoring—it suffered from the same weakness as Authoring by Demonstration, i.e., the lack of explicit negative feedback.

This result implies that it is not only the number of positive examples that matters for accuracy of learning, but receiving positive and negative feedback is an important factor for better learning. We will discuss this issue further in the section "Impact of Feedback."

In addition to producing more accurate negative examples, Authoring by Tutoring may produce better results because authors can review multiple suggestions for individual steps when multiple productions are applicable. Because more than 80 % of such suggestions were correct on the later problems (Fig. 8), Authoring by Tutoring had a higher chance of removing incorrect negative examples by applying the corresponding skill and receiving positive feedback. It is important to note that this work in Authoring by Tutoring of reviewing multiple suggestions does not require more time (as we discussed in the section "Authoring Efficiency Case Study").

Efficiency of Different Authoring Strategies

To evaluate the efficiency of each authoring strategy, we broke down the computation time spent for each learning algorithm into two key components: the search for a generalization for the if-part (i.e., the information retrieval path and the precondition), and the search for the then-part (i.e., the operator function sequence) of a production. The former corresponds to the time spent on two function calls referred to as generalize-information-retrieval-path () and induce-precondition () shown in Fig. 5, whereas the latter corresponds to search-for-operator-function-sequence ().

Table 4 shows the average time (in milliseconds) spent on learning the if-part and the then-part of productions per problem—i.e., the time shown is a sum of learning

Table 4 The average time (in milliseconds) spent on learning the if-part and then-part of productions per problem (with the standard deviation in parentheses)

	If-part	Then-part
Authoring by Tutoring	40[a] (22)	4990[b] (17,157)
Authoring by Demonstration	46[a] (36)	9408[b] (22,204)

multiple productions. The average is aggregated across five training sets (i.e., 20×5= 100 problems). For both if-part and then-part, Authoring by Tutoring was faster than Authoring by Demonstration, and the difference was statistically significant.

For the then-part learning, Authoring by Tutoring was 47 % faster than Authoring by Demonstration. Because learning often only involved generalization for the if-part (the preconditions), the then-part learning took zero seconds. On average, learning happened about 1.7 times per problem for Authoring by Tutoring and 1.6 times for Authoring by Demonstration. Of those learning opportunities, the then-part learning occurred about 53 % of the time in Authoring by Tutoring and about 50 % of the time in Authoring by Demonstration. We hypothesized that the large variance in the then-part learning was due to the large number of zero-second learning. This hypothesis, however, was not supported—even when we excluded zero-second learning, the standard deviation remained equally large.

As for the tutoring interaction, when Authoring by Tutoring, SimStudent requested a hint 1.7 times per problem in average (as the learning frequency indicated above). Also, SimStudent correctly and incorrectly applied productions 8.5 and 7.7 times per problem, respectively on average.

In sum, the data showed that Authoring by Tutoring is about twice as fast as Authoring by Demonstration with regard to learning speed. Since Authoring by Tutoring requires additional interaction time for feedback, it might take longer when driven by a human author. On the other hand, the author must demonstrate all steps for Authoring by Demonstration. Therefore, the pros and cons of the actual overall authoring time for two authoring strategies are unclear. The next section provides some insights for this issue through a case study.

Authoring Efficiency Case Study

This section describes a case study conducted to evaluate the amount of time that each authoring strategy requires. In particular, we measured the time spent training SimStudent and evaluating the learned expert model, because the difference on these steps would be the most prominent among the different authoring strategies.

There are many human factors to be considered for a rigorous efficiency study (e.g., familiarity with the domain and system, strategy preference, etc.), which is beyond the scope of the current paper. In this case study, we assumed that a hypothetical author knows the domain and is familiar with the authoring system, including CTAT and SimStudent.

Method

There was one participant (one of the authors of this paper who also conducted the Simulation Study) involved in this case study. The participant (called the author hereafter) was asked to author an expert model for an Algebra Equation Tutor twice, once for each of the two authoring strategies—Authoring by Demonstration and Authoring by Tutoring.

For Authoring by Demonstration, the author was instructed to demonstrate five problems at a time and then test the quality of the resulting expert model using a set of test problems. The author was also instructed to repeat this process four times for the total of 20 training problems. The author was given 20 training problems and ten test problems, which were randomly selected from the Simulation Study. The author was asked to create model solutions for the ten test problems by demonstrating their solutions. The demonstrated solutions were saved for later use. The author then repeated the following steps: first, the author demonstrated solutions for five training problems. Each demonstrated solution was saved in an individual file. The author then trained SimStudent with these saved demonstrations. After the training, the author validated the quality of the expert model learned by SimStudent. To run the validation, the author used a given shell script that applies given production rules to the set of ten test problems and reports the results.

For Authoring by Tutoring, the author was given the same 20 training problems as the ones used for the Authoring by Demonstration. The author was then asked to tutor SimStudent using these training problems one after another. There was no test problems used for Authoring by Tutoring, because we hypothesized that the tutoring interaction would provide formative feedback for the author to estimate the quality of the expert model.

Results

Figure 13 shows time spent for Authoring by Demonstration. The solid bars show time (in minutes) spent on demonstrating training problems and saving them into individual files. The hashed bars show SimStudent's learning time using the saved files. The meshed bars show time spent on testing the production rules. The X-axis shows the one-time preparation on the ten test problems (i.e., demonstrating the ten test problems) as well as the four demonstration-training-test iterations.

Demonstrating a solution for a single problem took 4.4 min on average, which is about the same as tutoring a single problem as discussed below. The time for SimStudent's learning on a single training problem took 4.3 min on average, but the learning time increased as the number of iterations increased due to the increase of positive and negative examples—SimStudent needed to find appropriate generalizations and specializations across all solutions demonstrated thus far. On the fourth iteration, the time for learning took 50 min (on average, the learning time took 21.3 min per iteration). The time for testing took 6.5 min on average.

Figure 14 shows time spent for Authoring by Tutoring. The average time to tutor on a single problem was 4.0 min. The time spent on each tutoring problem was fairly constant across 20 training problems.

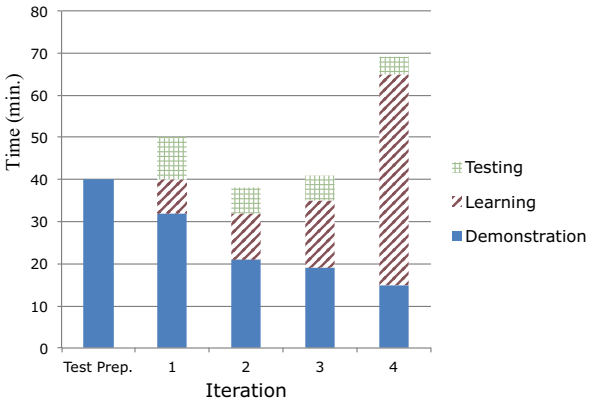

Fig. 13 Time spent to author with authoring by demonstration. The *Y*-axis shows time in minutes. The first bar on the *X*-axis shows the time spent to demonstrate the ten test problems to be used for the following training-test cycles. The first (1) through fourth (4) iterations are the training-test cycles that consisted of demonstrating five problems as well as training and testing SimStudent

It turned out that for Authoring by Tutoring, there was no extra time cost other than tutoring. Even though the author had to provide feedback on the steps performed by SimStudent (an extra task compared to Authoring by Demonstration), 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 to the time for demonstration, because the number of steps demonstrated during tutoring (i.e., providing "hints") decreases as learning progresses, and also because providing flagged feedback on the correctness of the steps is much quicker than demonstrating steps. On average, the author demonstrated 8.4 steps per problem for Authoring by Demonstration. In contrast, for Authoring by Tutoring, the author demonstrated 4.9 steps per problem in addition to providing 5.7 instances of positive and 6.6 instances of negative feedback. Authoring by Tutoring took 77 min to tutor 20 problems, whereas Authoring by Demonstration took 238 min to demonstrate 20 training and ten test problems. In

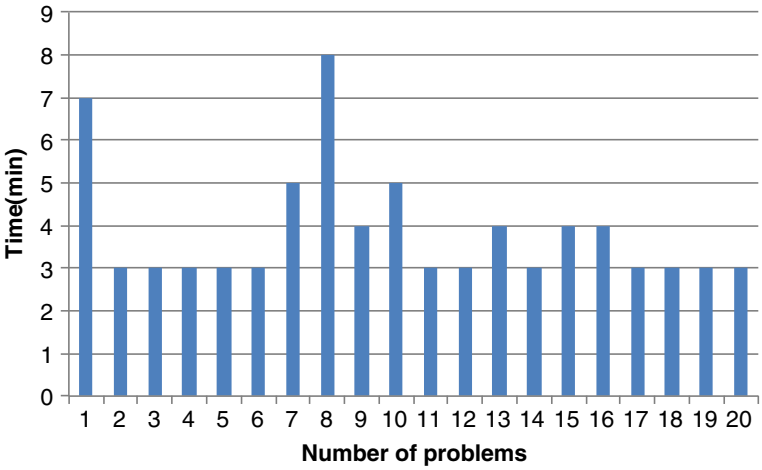

Fig. 14 Time spent authoring by tutoring SimStudent. The *x*-axis shows the number of training problems tutored. The *y*-axis shows time (min.) spent on tutoring

sum, it turned out that Authoring by Tutoring was about 3 times faster than Authoring by Demonstration to author an expert model for the Algebra Equation Tutor.

Discussion

Impact of Learning Strategy

The Simulation Study showed that Authoring by Tutoring is a better authoring strategy to create an expert model with SimStudent, both in terms of producing a more accurate expert model and in terms of drastically reducing the time required for authoring. Even when the same number of training problems in the same order is used, Authoring by Tutoring requires less time than Authoring by Demonstration. In particular, for Authoring by Demonstration, even though the cost for demonstration generally declines over time (as the author becomes familiar with the process of demonstration), the computational cost for SimStudent's learning increases as more and more training problems are accumulated for generalization. On the other hand, for Authoring by Tutoring, the cost for tutoring remains almost constant regardless of the number of problems tutored.

Authoring by Tutoring also has the advantage of having SimStudent suggest multiple production applications that facilitate pruning incorrect implicit negative examples. Providing explicit negative feedback is beneficial to SimStudent, because such negative feedback generates explicit negative examples that are always correct as opposed to implicit negative examples that can be incorrect. Furthermore, giving feedback on production applications, as is done in Authoring by Tutoring, provides a natural way for the author to gauge the quality of the expert model learned by SimStudent. Also, observing errors made by SimStudent provides the author with insights into the type of training problem that should be tutored next. All these features contribute to the advantages for Authoring by Tutoring.

The Simulation Study showed that the expert model learned by SimStudent, regardless of the authoring strategy, would have incorrect productions that might result in both the error of commission (i.e., making incorrect steps) and the error of omission (i.e., cannot perform a step correctly). Therefore, the current SimStudent technology might require the author to manually correct the production rules. Even though the author needs to modify an incorrect expert model, modifying an existing expert model is much easier than writing one from scratch.

The results from the Simulation Study also provide a theoretical account for the benefit of learning by guided-problem solving over learning from examples (Atkinson et al. 2003; Kalyuga et al. 2001). Our data predict that learning by guided-problem solving would better facilitate learning than learning from examples. This is because when students receive feedback on their production applications, they are more likely to catch incorrect productions and correct them. Our data suggest that human tutors should let tutees solve problems and then provide feedback on their incorrect performance. Our theory conjectures that when providing negative feedback on incorrect skill applications, students should be able to revise their productions if they can correctly carry out inductive reasoning to generalize feedback and hints.

Impact of Feedback

Our data suggest that the superiority of Authoring by Tutoring for the accuracy of learned production rules is due to the positive and negative feedback on production applications. Making errors and getting explicit negative feedback is crucial for successful authoring with SimStudent. Also, applying productions correctly and getting positive feedback is crucial to prune incorrect implicit negative examples. As a consequence, an expert model resulting from Authoring by Tutoring is better, as we saw in the Simulation Study, not in its recall but in its precision—it is not being more correct, but is being less incorrect that matters.

Despite the importance of negative examples, programming by demonstration in most cases only produces positive examples, or sometimes demonstrating a positive example on a particular concept serves as a negative example for other concepts, which we call 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 able to create a program to do a certain task (i.e., to learn what to do), but it also needs to reinforce the program so that it learns when to invoke the program (i.e., to learn when to do)—"otherwise, of what use is the program?" (p.424). We further claim the importance of explicit negative examples in order to effectively learn when to invoke the program, as well as the crucial role of positive examples in eliminating incorrect implicit negative examples. Therefore, programming by tutoring is a better alternative than programming by demonstration.

Interactive machine learning (Fails and Olsen 2003; Olsen and Pace 2005) 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) implemented the Model Inference System in a natural way to let the system run a program to find "conjectures that are too strong," which means letting the system make errors, and then have the user provide feedback including an anticipated behavior so that the system can debug the error.

The number of negative examples per se (when their correctness is not guaranteed) is not necessarily a predictor of a better learning outcome. Our study showed that implicit negative examples are not as useful as the explicit negative examples obtained by feedback on an incorrect rule application, and pruning incorrect implicit negative examples is a key for a successful learning.

Labeling Issues

The wrong type-in error particularly observed for CLT-typein (Table 1) could have been avoided if the author had labeled two skills for CLT-typein differently—one that actually combines like terms and the other one that simply copies terms. However, it might be troublesome for authors to differentiate potentially different skills. This is indeed a key issue of knowledge acquisition and even domain experts are generally not good at consciously accessing (as is necessary for authoring) their mostly tacit procedural knowledge.

One potential idea to deal with this issue is to have SimStudent make suggestions to differentiate skills. When disjunctive productions are generated for a particular skill,

SimStudent may tell the author that multiple productions have been generated (perhaps by showing how those productions differ from each other), and suggest that the author might re-label them accordingly.

Theoretically speaking, since SimStudent learns disjunctive productions, SimStudent can be tutored without skill labels. This is equivalent to tutoring SimStudent by labeling all skills with the same name, and then having SimStudent learn different skills as different disjuncts. Doing so certainly slows down (and perhaps diminishes) SimStudent's learning, and requires further study to fully explore the challenges and benefit of programming by tutoring without skill labels.

Generality of Findings

To apply the SimStudent technology to a new domain, the author needs to write background knowledge, i.e., feature predicates and operator functions. In the current SimStudent implementation, the background knowledge must be written in Java, which may be challenging for some novice authors. We argue, however, that writing background knowledge is not a burden required only by the SimStudent technology. When writing an expert model of a cognitive tutor by hand, authors need to provide background knowledge within the framework of a production rule description language.[6]

Background knowledge can be re-used for different tasks, if they are written at an appropriate grain size and abstract level. The background knowledge for arithmetic, for example, is likely to be usable for algebra and other algebra related domains (e.g., physics). We have made some effort to facilitate the cross-domain generalization and reduce the constraint of writing domain specific background knowledge. For example, Li et al. (2011) used a grammar induction technique to learn domain concepts from examples. Further studies are necessary to investigate an authoring aid to write background knowledge, and more importantly, how transfer of background knowledge from one domain to another domain can best be achieved.

The current implementation of SimStudent authoring technology relies heavily on the cognitive tutor architecture for a few essential points: (1) it assumes a strict tutoring interaction as cognitive tutors do (i.e., solution steps must be broken down into fine grained skills; feedback and hint must be provided for each step, etc.), (2) it assumes that the expert model is represented as production rules, and (3) it assumes that other tutoring components (problem selection, instructional strategy, feedback, hint, etc.) will function automatically once the expert model is provided. Therefore, applying the SimStudent technology to author other types of intelligent tutoring systems (e.g., constraint-based tutors) requires further work, although the theory of programming by demonstration would apply to broader domains.

We have demonstrated that the SimStudent technology facilitates authoring expert models for skill-acquisition tasks. In theory, if one could build a tutoring interface upon which necessary skills can be explicitly demonstrated, then SimStudent should be able to learn those skills. An example of a challenging task is the one that requires a mental

[6] It is also a common practice to invent invisible working memory elements to keep track of internal states that are otherwise represented with feature predicates. However, creating and maintaining such a scheme is as technically hard as writing the corresponding background knowledge in a programming language.

computation to provide rationale for an action to perform a step (e.g., planning). Investigating the issues regarding applying SimStudent to other type of domains (e.g., language acquisition, pattern recognition, and categorization, etc.) is an interesting and important future research agenda.

In sum, the SimStudent technology facilitates authoring expert models for cognitive tutors. It amplifies the effectiveness of CTAT by adding a significant value of automated cognitive modeling as a generalization of demonstrated solutions. Authors must provide background knowledge for new domains though it is also true that the domain specific background knowledge must be provided to write an expert model by hand. Carefully designed background knowledge for SimStudent might be shared across different domains.

Conclusion

We found that for the purpose of authoring an expert model, Authoring by Tutoring is a better alternative to Authoring by Demonstration. Our data showed that when authoring an expert model with SimStudent, Authoring by Tutoring better facilitates authoring both in the quality of the expert model generated and the time spent for authoring. The study also showed that the benefit of Authoring by Tutoring in creating a high-quality expert model is largely due to the feedback on the correctness of learned predictions applied proactively when solving problems.

We have demonstrated that programming by tutoring is a practical and effective technology for automated cognitive modeling. Further study is necessary to explore the broader benefits of programming by tutoring realized by a simulated learning technology.

In a broader context, the application of simulated learners has recently been growing rapidly (McCalla and Champaign 2013), partly due to recent advances in the study of computational models of learning and their application for education. Many important issues remain regarding the use of simulated learners in improving education including the impact of the cognitive fidelity of simulated learners, and the generalities and limitations of the many different possible learning theories that can be embedded in simulated learners.

References

Ainsworth, S., & Grimshaw, S. (2004). Evaluating the REDEEM authoring tool: can teachers create effective learning environments? *International Journal of Artificial Intelligence in Education, 14*, 279–312.

Aleven, V., McLaren, B. M., Sewall, J., & Koedinger, K. R. (2006). The cognitive tutor authoring tools (CTAT): Preliminary evaluation of efficiency gains. In M. Ikeda, K. D. Ashley, & T. W. Chan (Eds.), *Proceedings of the 8th international conference on intelligent tutoring systems* (pp. 61–70). Berlin: Springer Verlag.

Aleven, V., McLaren, B. M., Sewall, J., & Koedinger, K. R. (2009). A New paradigm for intelligent tutoring systems: example-tracing tutors. *International Journal of Artificial Intelligence in Education, 19*, 105–154.

Anderson, J. R., & Pelletier, R. (1991). A development system for model-tracing tutors. *Proc. of the International Conference on the Learning Sciences*, 1–8.

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.

Angros, R., Jr., Johnson, W. L., Rickel, J., & Scholer, A. (2002). *Learning domain knowledge for teaching procedural skills proceedings of the first international joint conference on autonomous agents and multiagent systems: Part 3* (pp. 1372–1378). New York: ACM Press.

Arruarte, A., Fernández-Castro, I., Ferrero, B., & Greer, J. (1997). The IRIS shell: how to build ITSs from pedagogical and design requisites. *International Journal of Artificial Intelligence in Education, 8*, 341–381.

Atkinson, R. K., Renkl, A., & Margaret Merrill, M. (2003). Transitioning from studying examples to solving problems: effects of self-explanation prompts and fading worked-out steps. *Journal of Educational Psychology, 95*(4), 774–783.

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

Booth, J. L., & Koedinger, K. R. (2008). Key misconceptions in algebraic problem solving. In B. C. Love, K. McRae, & V. M. Sloutsky (Eds.), *Proceedings of the 30th annual conference of the cognitive science society* (pp. 571–576). Austin: Cognitive Science Society.

Burton, R. R., & Brown, J. S. (1982). An investigation of computer coaching for informal learning activities. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 79–98). Orlando: Academic.

Chi, M. T. H. (1997). Quantifying qualitative analyses of verbal data: a practical guide. *Journal of the Learning Sciences, 6*(3), 271–315.

Clark, R. E., & Estes, F. (1996). Cognitive task analysis for training. *International Journal of Educational Research, 25*(5), 403–417.

Corbett, A. T., Anderson, J. R., & O' Brien, A. T. (1995). Student modeling in the ACT programming tutor. In P. D. Nichols & S. F. Chipman (Eds.), *Cognitively diagnostic assessment* (pp. 19–41). Hillsdale: Lawrence Erlbaum Associates, Inc.

Corbett, A. T., McLaughlin, M., & Scarpinatto, K. C. (2000). Modeling student knowledge: cognitive tutors in high school and college. *User Modeling and User Adapted Interaction, 10*(2–3), 81–108.

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

Fails, J. A., & Olsen, D. R., Jr. (2003). *Interactive machine learning Proceedings of IUI2003* (pp. 39–45). Miami Beach: ACM.

Gott, S. P., & Lesgold, A. M. (2000). Competence in the workplace: How cognitive performance models and situated instruction Can accelerate skill acquisition. In R. Glaser (Ed.), *Advances in instructional psychology: Educational design and cognitive science* (Vol. 5, pp. 239–327). Mahwah: Erlbaum.

Heffernan, N. T., Turner, T. E., Lourenco, A. L. N., Macasek, M. A., Nuzzo-Jones, G., & Koedinger, K. R. (2006). The ASSISTment Builder: Towards an Analysis of Cost Effectiveness of ITS creation *Proc of FLAIRS2006*.

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: Springer.

Jonassen, D. H., Tessmer, M., & Hannum, W. H. (1999). *Task analysis methods for instructional design*. Mahwah: Lawrence Erlbaum Associates, Inc., Publishers.

Kalyuga, S., Chandler, P., Tuovinen, J., & Sweller, J. (2001). When problem solving is superior to studying worked examples. *Journal of Educational Psychology, 93*(3), 579–588.

Kieras, D. E. (1988). Towards a practical GOMS model methodology for user interface design. In M. Helander (Ed.), *Handbook of human-computer interaction* (pp. 135–157). New York: Elsevier.

Kiyama, M., Ishiuchi, S., Ikeda, K., Tsujimoto, M., & Fukuhara, Y. (1997). Authoring methods for the Web-based intelligent CAI system CALAT and its application to telecommunications service. *AAAI Technical Report FS, 97*(01), 44–52.

Koedinger, K. R., & Aleven, V. (2007). Exploring the assistance dilemma in experiments with cognitive tutors. *Educational Psychology Review, 19*(3), 239–264.

Koedinger, K. R., & Corbett, A. T. (2006). Cognitive tutors: Technology bringing learning sciences to the classroom. In R. K. Sawyer (Ed.), *The Cambridge handbook of the learning sciences* (pp. 61–78). New York: Cambridge University Press.

Koedinger, K. R., & Nathan, M. J. (2004). The real story behind story problems: effects of representations on quantitative reasoning. *Journal of the Learning Sciences, 13*, 129–164.

Koedinger, K. R., Aleven, V., & 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.

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., Cunningham, K., Skogsholm, A., & Leber, B. (2008). An open repository and analysis tools for fine-grained, longitudinal learner data *Proceedings of the international Conference on Educational Data Mining.*

Koedinger, K. R., Baker, R. S. J. D., Cunningham, K., Skogsholm, A., Leber, B., & Stamper, J. (2010). A data repository for the EDM community: The PSLC Datashop. In C. Romero, S. Ventura, M. Pechenizkiy, & R. S. J. D. Baker (Eds.), *Handbook of educational data mining*. Boca Raton: CRC 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: MIT Press.

Langley, P., Bradshaw, G. L., & Simon, H. A. (1987). Heuristics for empirical discovery. In L. Bolc (Ed.), *Computational models of learning*. Berlin: Springer.

Lau, T. A., & Weld, D. S. (1998). *Programming by demonstration: An inductive learning formulation proceedings of the 4th international conference on intelligent user interfaces* (pp. 145–152). New York: ACM Press.

Li, N., Matsuda, N., Cohen, W. W., & Koedinger, K. R. (2011). A machine learning approach for automatic student model discovery. In M. Pechenizkiy, T. Calders, C. Conati, S. Ventura, C. Romero, & J. Stamper (Eds.), *Proceedings of the international conference on educational data mining* (pp. 31–40). Eindhoven: International Educational Data Mining Society.

London, B., & Clancey, W. J. (1982). *Plan recognition strategies in student modeling: Prediction and description*: Department of Computer Science, Stanford University.

Matsuda, N., Cohen, W. W., & Koedinger, K. R. (2005a). *Applying programming by demonstration in an intelligent authoring tool for cognitive tutors aaai workshop on human comprehensible machine learning (Technical report WS-05-04)* (pp. 1–8). Menlo Park: 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.

Matsuda, N., Cohen, W. W., Sewall, J., & Koedinger, K. R. (2006). *Applying machine learning to cognitive modeling for cognitive tutors machine learning department technical report (CMU-ML-06-105)*. Pittsburgh: School of Computer Science, Carnegie Mellon University.

Matsuda, N., Cohen, W. W., Sewall, J., Lacerda, G., & Koedinger, K. R. (2008). Why tutored problem solving may be better than example study: Theoretical implications from a simulated-student study. In B. P. Woolf, E. Aimeur, R. Nkambou, & S. Lajoie (Eds.), *Proceedings of the international conference on intelligent tutoring systems* (pp. 111–121). Heidelberg: Springer.

McCalla, G., & Champaign, J. (Eds.). (2013). *Proceedings of the AIED workshop on simulated learners*. Memphis: International AIED Society.

McDaniel, R., & Myers, B. A. (1998). *Building applications using only demonstration proceedings of the 3rd international conference on intelligent user interfaces* (pp. 109–116). New York: ACM Press.

Mitchell, T. M. (1997). *Machine learning*. Boston: McGraw-Hill.

Mitrovic, A., & Ohlsson, S. (1999). Evaluation of a constraint-based tutor for a database language. *International Journal of Artificial Intelligence in Education, 10*, 238–256.

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

Muggleton, S. (1999). Inductive logic programming: issues, results and the challenge of learning language in logic. *Artificial Intelligence, 114*(1–2), 283–296.

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

Munro, A., Johnson, M. C., Pizzini, Q. A., Surmon, D. S., Towne, D. M., & Wogulis, J. L. (1997). Authoring simulation-centered tutors with RIDES. *International Journal of Artificial Intelligence in Education, 8*, 284–316.

Murray, T. (1998). Authoring knowledge-based tutors: tools for content, instructional strategy, student model and interface design. *Journal of the Learning Sciences, 7*(1), 5–64.

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.

Murray, T. (2003). An overview of intelligent tutoring system authoring tools. In T. Murray, S. Ainsworth, & S. B. Blessing (Eds.), *Authoring tools for advanced technology learning environment* (pp. 491–544). Netherlands: Kluwer.

Olsen, D. R., Jr., & Pace, A. (2005). Human-guided Machine Learning. *Proceedings of UIST*, 1–9.

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

Ritter, S., & Koedinger, K. R. (1996). *An architecture for plug-in Tutor Agents*.

Ritter, S., Anderson, J. R., Koedinger, K. R., & Corbett, A. (2007). Cognitive tutor: applied research in mathematics education. *Psychonomic Bulletin & Review, 14*(2), 249–255.

Sammut, C., & Banerji, R. B. (1986). Learning concepts by asking questions. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2). Los Altos: Kaufmann.

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

Shute, V. J., & Psotka, J. (1994). *Intelligent tutoring systems: past, present, and future. (AL/HR-TP-1994-0005)*. Brooks Air Force Base: Armstrong Laboratory.

Sparks, R., Dooley, S., Meiskey, L., & Blumenthal, R. (1999). The LEAP authoring tool: supporting complex courseware authoring through reuse, rapid prototyping, and interactive visualizations. *International Journal of Artificial Intelligence in Education, 10*, 75–97.

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.

Wenger, E. (1987). *Artificial intelligence and tutoring systems: Computational and cognitive approaches to the communication of knowledge*. Los Altos: Morgan Kaufmann.