Barnes, T., & Stamper, J. (2010). Automatic Hint Generation for Logic Proof Tutoring Using Historical Data. *Educational Technology & Society*, *13* (1), 3–12.

Automatic Hint Generation for Logic Proof Tutoring Using Historical Data

Tiffany Barnes and John Stamper
University of North Carolina at Charlotte, Computer Science Department, Charlotte, NC, USA //
Tiffany.Barnes@gmail.com // John@stamper.org

ABSTRACT

In building intelligent tutoring systems, it is critical to be able to understand and diagnose student responses in interactive problem solving. However, building this understanding into a computer-based intelligent tutor is a time-intensive process usually conducted by subject experts. Much of this time is spent in building production rules that model all the ways a student might solve a problem. In our prior work, we proposed a novel application of Markov decision processes (MDPs) to automatically generate hints for an intelligent tutor that learns. We demonstrate the feasibility of this approach by extracting MDPs from four semesters of student solutions in a logic proof tutor, and calculating the probability that we will be able to generate hints for students at any point in a given problem. Our past results indicated that extracted MDPs and our proposed hint-generating functions will be able to provide hints over 80% of the time. Our results also indicated that we can provide valuable tradeoffs between hint specificity and the amount of data used to create an MDP.

Keywords

Educational data mining, Hint generation, Intelligent tutoring, Propositional logic proofs

Introduction

According to the Joint Task Force on Computing Curricula (2005), discrete mathematics is a core course in computer science, and an important topic in this course is solving formal logic proofs. However, this topic is of particular difficulty for students, who are unfamiliar with logic rules and manipulating symbols. To allow students extra practice and help in writing logic proofs, we are building an intelligent tutoring system on top of our existing proof-verifying program. Results from student surveys and our experience in teaching discrete math indicate that students particularly need hints when they get stuck.

The problem of offering individualized help is not unique to logic proofs. Through adaptation to individual learners, intelligent tutoring systems (ITS) can have significant effects on learning (Anderson & Gluck, 2001). However, building one hour of adaptive instruction takes between 100 and 1000 hours of work for subject experts, instructional designers, and programmers (Murray, 1999), and a large part of this time is used in developing production rules that model student behavior and progress. A variety of approaches have been used to reduce the development time for ITSs, including ITS authoring tools and constraint-based student models. ASSERT is an ITS authoring system that uses theory refinement to learn student models from an existing knowledge base and student data (Baffes & Mooney, 1996). Constraint-based tutors, which look for violations of problem constraints, require less time to construct and have been favorably compared to cognitive tutors, particularly for problems that may not be heavily procedural (Mitrovic, Keodinger, & Martin, 2003). However, constraint-based tutors can only provide condition-violation feedback, not goal-oriented feedback that has been shown to be more effective (Van Lehn, 2006).

Some systems use teacher-authored or demonstrated examples to develop ITS production rules. RIDES is a "Tutor in a Box" system used to build training systems for military equipment usage, while DIAG was built as an expert diagnostic system that generates context-specific feedback for students (Murray, 1999). These systems cannot be easily generalized, however, to learn from student data. In example-based authoring tools, teachers work problems in what they predict to be common correct and incorrect approaches, and then annotate the learned rules with appropriate hints and feedback. The Cognitive Tutors Authoring Tool (CTAT) has been used to develop example-based tutors for genetics, Java, and truth tables (Koedinger, Aleven, Heffernan, McLaren, & Hockenberry, 2004). This system has also been used with data to build initial models for an ITS, in an approach called Bootstrapping Novice Data (BND) (McLaren, Koedinger, Schneider, Harrer, & Bollen, 2004). However, in both of these approaches, considerable time must still be spent in identifying student approaches and creating appropriate hints.

Machine learning has also been used to improve tutoring systems. In the ADVISOR tutor, machine learning was used to build student models that could predict the time students took to solve arithmetic problems, and to adapt

ISSN 1436-4522 (online) and 1176-3647 (print). © International Forum of Educational Technology & Society (IFETS). The authors and the forum jointly retain the copyright of the articles. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear the full citation on the first page. Copyrights for components of this work owned by others than IFETS must be honoured. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from the editors at kinshuk@ieee.org.

instruction to minimize this time while meeting teacher-set instructional goals (Beck, et al., 2000). In the Logic-ITA tutor, student data was mined to create hints that warned students when they were likely to make mistakes using their current approach (Merceron & Yacef, 2005). Another logic tutor called the Carnegie Proof Lab uses an automated proof generator to provide contextual hints (Sieg, 2007).

Similar to the goal of BND, we seek to use student data to directly create student models for an ITS. However, instead of feeding student behavior data into CTAT to build a production rule system, our method generates Markov decision processes (MDPs) that represent all student approaches to a particular problem, and uses these MDPs directly to generate hints. This method of automatic hint generation using previous student data reduces the expert knowledge needed to generate intelligent, context-dependent hints and feedback. The system is capable of continued refinement as new data is provided. In this work, we demonstrate the feasibility of our hint generation approach through simulation experiments on existing student data.

Background and Proofs Tutorial context

The Proofs Tutorial is a computer-aided learning tool on NovaNET (http://www.pearsondigital.com/novanet/). This program has been used for practice and feedback in writing proofs in university discrete mathematics courses taught by the first author and others at North Carolina University since 2002 and at UNC Charlotte since 2006. In the Proofs Tutorial, students are assigned a set of 10 problems that range from simpler logical equivalence applications to more complex inference proofs. In the tutorial, students type in consecutive lines of a proof, which consist of four parts: the statement, reference lines, the axiom used, and the substitutions that allow the axiom to be applied. After the student enters these four parts, the statement, reference lines, axiom, and substitutions are verified. If a mistake is made, a warning message is shown, and the line is deleted (but saved for later analysis). In this work, we examine student solutions to Proof 1. Table 1 lists an example of a student solution that includes three errors.

In Barnes (2006), the first author has applied educational data mining to analyze completed formal proof solutions for automatic feedback generation. However, this work did not take into account student errors, and could only provide general indications of student approaches, as opposed to feedback tailored to a student's current progress. In Stamper (2006), the second author performed a pilot study to extract MDPs for a simple proof from three semesters of student data, and verified that the extracted rules conformed to expert-derived rules and generated buggy rules that surprised experts. In Barnes & Stamper (2007), we used visualization tools to explore how to generate hints based on MDPs extracted from student data. In Croy, Barnes, & Stamper (2007), we applied the technique to visualize student proof approaches to allow teachers to identify problem areas for students. This was just one method used to identify students at risk of falling behind in the course.

Table 1. Sample Proof 1 solution

	Statement	Line	Reason
1.	a → b		Given
2.	c → d		Given
3.	¬ (a → d)		Given
	¬ a v d	**3**	**rule IM (error)**
4.	a ^ ¬ d	3	rule IM implication
5.	a	4	rule S simplification
	b	**4**	**rule MP (error)**
	b	**1**	**rule MP (error)**
6.	b	1,5	rule MP modus ponens
7.	¬ d	4	rule S simplification
8.	¬c	2,7	rule MT modus tollens
9.	b ^ ¬c	6,8	rule CJ conjunction

Markov decision processes

A Markov decision process (MDP) is defined by its state set *S*, action set *A*, transition probabilities *P*, and a reward function *R* (Sutton & Barto, 1998). On executing action *a* in state *s*, the probability of transitioning to state *s'* is

denoted $P(s' \mid s, a)$ and the expected reward associated with that transition is denoted $R(s' \mid s, a)$. For a particular point in a student's proof, our method takes the current statements and conclusion as the state, and the student's input as the action. Therefore, each proof attempt can be seen as a graph with a sequence of states (each describing the solution up to the current point), connected by actions. Specifically, a state is represented by the list of statements generated in the student attempt, and actions are the axioms used at each step.

We combine all student solution graphs into a single graph by taking the union of all states and actions and mapping identical states to one another. Once this graph is constructed, it represents all of the paths students have taken in working a proof. Typically at this step, value iteration is used to find an optimal solution to the MDP. For the experiments in this study, we set a large reward for the goal state (100), penalties for incorrect states (10), and a cost for taking each action (1). Setting a non-zero cost on actions causes the MDP to penalize longer solutions (but we set this at 1/10 the cost of taking an incorrect step). These values may need to be adjusted for different sizes of MDPs. We apply the value iteration technique using a Bellman backup to assign reward values to all states in the MDP (Sutton & Barto, 1998). The equation for calculating values $V(s)$ for each state s, where $R(s)$ is the reward for the state, γ is the discount factor (set to 1), and $P_a(s,s')$ is the probability that action a will take state s to state s':

$$V(s) = R(s) + \gamma \cdot \max_a \left(\sum_{s'} P_a(s,s') \cdot V(s') \right)$$

For value iteration, V is calculated for each state until there is little change in the value function over the entire state space. Once this is complete, the optimal solution in the MDP corresponds to taking a greedy traversal approach in the MDP (Barnes & Stamper, 2007). The reward values for each state then indicate how close to the goal a state is, while probabilities of each transition reveal the frequency of taking a certain action in a certain state.

Generating individualized help

We propose to provide real-time, individualized hints to support on-going student proof construction efforts. As described in (Barnes & Stamper, 2007), we generate an MDP for each problem and use it to generate hints for new students who are solving proofs. Since our tutorials have been used as a computer-aided instructional tool for a number of years, we have many semesters of data from which to create large MDPs for each proof problem. We first use these MDPs to add intelligent hints to every problem. As a new student works a problem, we match each of their states to those in the MDP. If their state is present in the problem's MDP, we enable a hint button to give contextual help.

In Barnes & Stamper (2007), we have proposed several reward functions that could be used in hint generation, including expert, typical, and least error-prone. The reward function we have described herein reflects an expert reward function, where the value for a state reflects the shortest path to the goal state. Given the current state, when the hint button is pressed, we select a reward function for the current student based on his or her student profile. If we have identified the student as an at-risk student, we may select the "least error-prone" reward function for generating hints. On the other hand, high-performing students would likely benefit from expert hints, while students between these two extremes may benefit from hints reflecting typical student behavior (Barnes & Stamper, 2007). After we've selected a reward function, we select the next state with the highest reward value. We create four levels of hints from this state, as follows:
(1) Indicate a goal statement to derive (goal-setting hint).
(2) Tell the student what rule to apply next (rule hint).
(3) Indicate the statements where the rule can be used (pointing hint).
(4) Tell the student both the rule and the statements to combine (bottom-out hint).

The hint sequence is constructed to provide goal-setting, pointing, and bottom-out hints. Hints that help students set intermediate goals have been shown to be effective in (McKendree, 1990). Pointing hints help focus user attention, while bottom-out hints essentially tell students the answer (Van Lehn, 2006).

We plan to limit the number of hints a student can use and still receive credit for working the problem. We believe that four hints is a fair number, to be used on a single state in sequence as above or on separate states in the same problem. This results in giving the student one full step of the proof, or allowing rule hints up to four times.

If a student's state is not found in the MDP, the hint button will be disabled. The student can then get the tutor's built-in feedback that indicates the correctness of each step, but will not get strategic help. However, we can add the student's action and its correctness to our database and periodically run value iteration to update the reward function values. Before an update is applied, we will test the update to be sure that the instructors agree with the generated hints.

Method

This experiment uses data from the four fall semesters of 2003–2006, during which an average of 220 students took the discrete math course at NC State University. Students in this course were typically engineering and computer science students in their second or third year of college, but most had not been exposed to a course in logic. Students attended several lectures on propositional logic and completed online homework in which students completed truth tables and filled in the blanks in partially completed proofs. Students then used the Proofs Tutorial to solve 10 proofs, directly or indirectly. Sixty percent of students used direct proof when solving proof 1. We extracted 537 of students' first attempts at direct solutions to proof 1 from the Proofs Tutorial.

The data were validated by hand by extracting all statements generated by students and removing those that 1) were false or unjustifiable, or 2) were of improper format. We also removed all student steps using the axioms Conjunction, Double Negation, and Commutative, since students were allowed to skip these steps in the tutorial. After cleaning the data, there were 523 attempts at proof 1. Of these, 381 (73%) were complete and 142 (27%) were partial proofs, indicating that most students completed the proof. The average lengths, including errors, were 13 and 10 steps, respectively, for completed and partial proofs. When excluding errors and removed steps, the average number of lines in each student proof was 6.3 steps.

We performed several experiments to explore the capability of our method to generate automated hints. In each experiment, we isolated the data into training and test sets, where the training set was used to generate the Markov Decision Process (MDP) as described above, and the test set was used to explore hint availability. The process for comparing the test set to the MDP consisted of several steps. Because of the structure of the tutorial, we first removed all error states from the MDP and from student attempts before comparison, since the tutorial provides error messages and deletes the corresponding error from the student proof. Then, each attempt in the test set was mapped onto a sequence of states. For each test state, there are two requirements for a hint to be available: 1) there must be a "matching" state in the MDP, and 2) the "matching" state must not be a leaf in the MDP. The closer the match between a test state and the corresponding MDP state, the more context-specific the hint based on that match is.

Leaves

Because we included partial solutions in our training datasets, there are leaves in the MDPs or, in other words, statements with no subsequent actions taken. Therefore, we potentially had a higher percentage of matches than the number of states where hints could be generated. To investigate this, we examined all leaf nodes, and found that the overwhelming majority of leaves occurred in only one semester, for one student. This means that these states are never matched, and therefore do not count toward being able to provide hints. There were a total of three leaf states that occurred in multiple semesters. For two of these states, they occurred twice in one semester, and once in another. For the remaining leaf, it occurred once in one semester and once in another. Therefore, we have over-counted the number of times we can give hints to students by at most two matches in any semester. Since the minimum number of state visits in any semester is 304, this represents an error of at most 2/304 (0.06%) in any reported statistic.

Matching functions

In our experiments, we considered four matching functions that would allow us to select a source statement for hint generation: 1) ordered, 2) unordered, 3) ordered minus the latest statement, and 4) unordered minus the latest statement. An ordered, or exact, state match means that another student has taken the same sequence of steps in solving the proof. An unordered state match means that there is a state with exactly the same statements, but the states were not necessarily reached in the same order. An "ordered-1" match looks for an exact match between the

student's previous state and an MDP state. An "unordered-1" match looks for an unordered match between the student's previous state and an MDP state. Once a match is made, we generate a hint based on knowing the next optimal (highest reward value) step from the matching state. The more specific the match, the more contextualized the hint.

To determine hint availability, we calculated two numbers. The first was the "move matches," the number of test set states or "moves," including duplicates, that had matches in the MDP, divided by the total number of test set states. The second was the "unique matches," where we determined all unique test states and calculated the percentage of these that have matches in the MDP. Move matches gave us a measure of the percentage of the time a particular student was able to attain a hint while working a proof. Unique matches gave us a measure of the percentage of overlap in the states in the test set and the MDP.

We conducted two experiments to test the feasibility of automated hint generation. The first was something like a cross-validation study, comparing the hints we could generate using various semesters of data for MDP creation. The second was a simulation of creating MDPs incrementally as students worked proofs and calculating the probability of being able to generate hints as new attempts were added to the MDP.

Experiment 1: Comparing classes

In this experiment, we explored the ability of our system to provide hints after one, two, three, or four semesters of data were used to build MDPs. Table 2 shows that each semester was used as a test set (denoted by f and the semester), while all the remaining semesters were used as training sets for MDPs. For example, when fall 2003 was used as test set f3, it was compared with MDPs created from one semester of data each (e.g., M4 = fall 2004), two semesters of data each (e.g., M45 = fall 2004 and 2005), and three semesters of data (e.g., M456 = fall 2004 to 2006).

Table 2. Experimental design for comparing classes

Test	1-sem. MDP	2-sem. MDP	3-sem. MDP
f3	M4, M5, M6	M45, M46, M56	M456
f4	M3, M5, M6	M35, M36, M56	M356
f5	M3, M4, M6	M34, M36, M46	M346
f6	M3, M4, M5	M34, M35, M45	M345

This experiment provides us insight into the number of semesters of data we might need to provide hints a reasonable percentage of the time while students are solving proofs. Table 3 presents the data for each semester. We note that semester fall 2005 was unusual: there was a small number of states, but a large number of moves, suggesting that students solved these proofs in very similar ways.

Table 3. Semester data, including attempts, moves, and MDP states

Semester	# Attempts	MDP states	# Moves
f3	172	206	711
f4	154	210	622
f5	123	94	500
f6	74	133	304

We hypothesized that we could provide hints a majority of the time, using just one semester as our MDP training data. Table 4 shows the percentage of ordered matches between each semester and the remaining combinations of training sets. We were very encouraged by these data, which suggest that our system would provide highly contextualized hints over 66% of the time, in the worst case, after just one semester of training. In all cases, adding more data increased the probability of providing hints, though we do see diminishing returns when comparing the marginal increase between 1–2 (6.8%) and 2–3 (2.8%) semesters of data.

Table 4. Average % move matches using the ordered function

Test set	1-sem. MDPs	2-sem. MDPs	3-sem. MDPs
f3	68.73%	75.67%	78.62%
f4	69.77%	77.71%	81.03%
f5	86.33%	90.80%	92.00%
f6	66.34%	74.12%	77.63%
Average	72.79%	79.57%	82.32%

Tables 5–7 show the results of this experiment using the unordered, ordered-1, and unordered-1 matching techniques. These results show consistent increases within each table, going from 1-semester MDPs up to 2-semester MDPs, as expected. However, the increases between 2- and 3-semester MDPs are decreasing, suggesting consistent diminishing returns for adding more data to the MDPs.

Table 5. Average % move matches using the unordered function

Test set	1-sem. MDPs	2-sem. MDPs	3-sem. MDPs
f3	76.62%	82.16%	84.37%
f4	75.35%	81.99%	84.41%
f5	91.93%	94.40%	95.40%
f6	74.56%	82.35%	84.87%
Average	79.62%	85.22%	87.26%

Table 6. Average % move matches using the ordered-1 function

Test set	1-sem. MDPs	2-sem. MDPs	3-sem. MDPs
f3	76.92%	85.14%	89.00%
f4	76.26%	85.69%	90.35%
f5	90.78%	96.19%	97.80%
f6	75.55%	84.32%	89.14%
Average	79.88%	87.84%	91.57%

Table 7. Average % matches using the unordered-1 function

Test set	1-sem. MDPs	2-sem. MDPs	3-sem. MDPs
f3	82.63%	89.19%	91.99%
f4	81.73%	90.14%	93.41%
f5	94.60%	97.00%	98.00%
f6	81.03%	89.69%	92.43%
Average	85.00%	91.50%	93.96%

Table 8 lists the average percentage of matches for each of our experiments using the four match functions. This table gives an indication of the tradeoffs between using multiple semesters of data versus multiple techniques for matching. Here, we see that, on average, for 72% of moves, we can provide highly contextualized (ordered) hints using just one semester of data. With two semesters of data, we can provide these hints almost 80% of the time, but this only increases to 82% for three semesters of data. If we wished to provide hints after collecting just one semester of data, we could provide less contextualized hints for those who didn't have ordered matches in the MDP. There is a nearly identical benefit to providing hints using unordered versus ordered-1 searches, increasing the match rate to almost 80%. We did not calculate the marginal benefit of providing one of these over the other. However, we can provide hints to an additional 5% of students if we add the unordered-1 match function.

When analyzing these data, we observed a skew in all statistics because of the unusual distribution of states and moves in f5. We therefore repeated all experiments excluding f5, and the results are given in Table 9. The differences caused by skew in f5 had a smaller effect as you move from top left to bottom right, suggesting that more data or less sensitive matching can mitigate the effect of unusual training data.

Table 8. Comparison of % move matches and matching techniques

Matching	1-sem. MDPs	2-sem. MDPs	3-sem. MDPs
Ordered	72.79%	79.57%	82.32%
Unordered	79.62%	85.22%	87.26%
Ordered-1	79.88%	87.84%	91.57%
Unordered-1	85.00%	91.50%	93.96%

Table 9. Comparison of % move matches, excluding f5

Test set	1-sem. MDPs	2-sem. MDPs
Ordered	70.97%	78.05%
Unordered	78.69%	83.59%
Ordered-1	79.02%	87.99%
Unordered-1	85.77%	91.86%

Table 10 shows the marginal increase of each matching technique, for each MDP size, to illustrate the tradeoffs between additional data and matching technique. When considering matching functions, the easiest technical change is from ordered to ordered-1, where one statement is removed from the test state before comparison with the MDP states. In all cases, the benefit of providing these hints is higher than that of providing hints based on unordered matches. This is probably because there is some inherent partial ordering in proofs, so only limited benefit is seen from reordering statements. When an ordered hint cannot be matched, it is perhaps more likely that the student has just performed a step that no one else has done before, rather than generating a new ordering of steps, so the benefit of ordered-1 can exceed that of unordered. Providing the unordered search requires us to maintain two separate MDPs (one ordered and one unordered) to make the search more efficient, so there are both time and space tradeoffs to using unordered matching. However, adding unordered-1 after adding unordered provides a very large difference in our capability to provide hints, with little investment in time.

Table 10. Marginal increases when comparing matching techniques to ordered

Technique	1-sem. ordered	2-sem. ordered	3-sem. ordered
Unordered	6.83%	5.65%	4.94%
Ordered-1	7.09%	8.27%	9.25%
Unordered-1	12.21%	11.93%	11.64%

As part of this study we also compared the unique states across semesters, as shown in Table 11. This gives us a measure of the percent overlap between MDPs. Three semesters of data with ordered matching or one semester of data with unordered-1 matching will give us over 50% matching of states across MDPs.

Table 11. Unique state % matches across semesters and techniques

Test set	1-sem. MDPs	2-sem. MDPs	3-sem. MDPs
Ordered	34.55%	45.84%	51.93%
Unordered	43.62%	55.23%	59.90%
Ordered-1	48.25%	63.07%	71.39%
Unordered-1	57.28%	71.98%	77.87%

Experiment 2: Exploring the "cold start" problem

One critique of using data to generate hints has been the expected time needed for the method to be applied to a new problem, or in other words, the "cold start" issue. Our hypothesis was that a relatively low number of attempts would be needed to build an MDP that could provide hints to a majority of students. One method for building our hint MDP would be to incrementally add MDP states as students solve proofs. This experiment explores how quickly such an MDP is able to provide hints to new students, or in other words, how long it takes to solve the cold start problem. For one trial, the method is given below.

Let Test = {all 523 student attempts}
Randomly choose and remove the next attempt a from the Test set.
Add *a*'s states and recalculate the MDP.
Randomly choose and remove the next attempt *b* from the Test set.
Compute the number of matches between *b* and MDP.
If Test is non-empty, then let *a* = *b* and go to step 3. Otherwise, stop.

For this experiment, we used the ordered and unordered matching functions, and plotted the resulting average matches over 100,000 trials, as shown in Figure 1. These graphs show a very quick rise in ability to provide hints to students, which can be fit using power functions, whether the system uses ordered or unordered MDP states and matching.

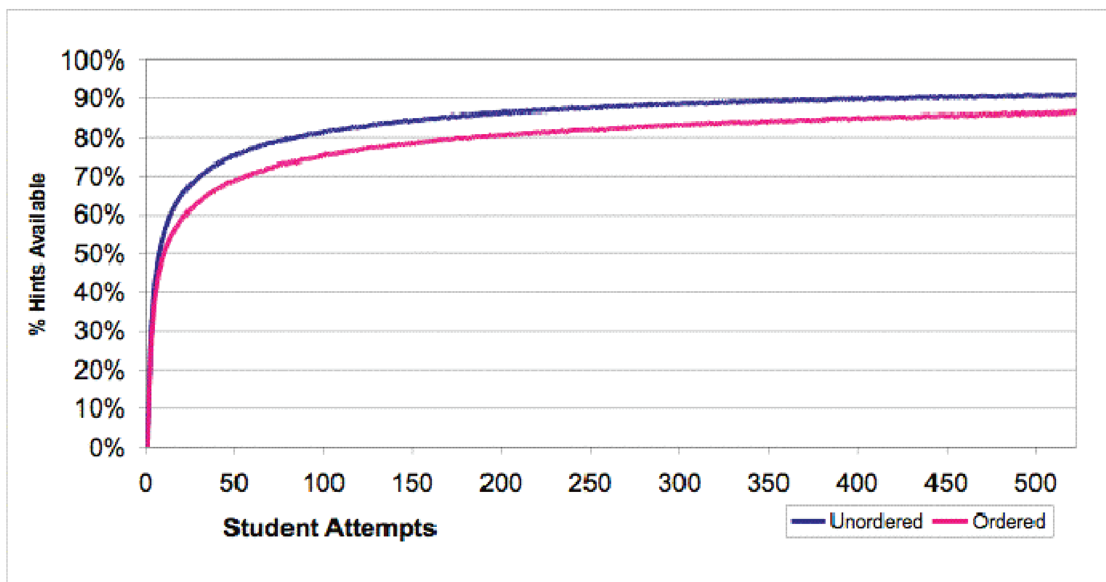

Figure 1. Hints available when the MDP is constructed using a given number of attempts, averaged over 100,000 random orderings of the attempts selected for the MDP.

Clearly, the availability to give hints ramps up very quickly. Table 12 lists the number of attempts needed in the MDP versus target hint percentages. For the unordered matching function, the 50% threshold is reached at just 8 student attempts and the 75% threshold at 49 attempts. For ordered matching, 50% occurs on attempt 11 and 75% on attempt 88. These data are encouraging, suggesting that instructors using our MDP hint generator could seed the data to jump-start new problems. By allowing the instructor to enter as few as 8–11 example solutions to a problem, the method might already be capable of automatically generating hints for 50% of student moves.

Table 12. Attempts needed to achieve threshold % hints levels

	50%	55%	60%	65%	70%	75%	80%	85%	90%
Un-Ordered	8	11	14	20	30	46	80	154	360
Ordered	11	15	22	33	55	85	162	362	N/A

Pilot study on hint generation and availability

We have constructed a hint generator using the methods described herein to add hints to Deep Thought, a visual logic tutor created by Marvin Croy (2000). When a student presses the new hint button, our hint generator searches for the current state in the MDP and checks that a successor state exists. If it does, the successor state with the highest value is used to generate a hint sequence as described above. In our pilot experiment as described in Barnes, Stamper, Lehman, & Croy (2008), hints were added to four logic proof problems in a spring 2008 deductive logic course with 40 students enrolled. MDPs were created for these four problems with 16 to 26 training examples, and the percent hint availability was calculated for all problem states. Based on the results in this study, we predicted that hints (using ordered matching) would be available approximately 56–62% of the time. In the pilot study, if a student had pressed the hint button after every move taken, a hint would have been available about 48% of the time. Although this percentage seems low, we found that when students requested hints, they were available 91% of the time. This suggests that hints are needed precisely where we have data in our MDPs from previous semesters. There are several potential explanations for this: students may be avoiding using hints; hints may be most needed in only a few key steps; or the students may have felt very confident in solving proofs before working these problems. We plan to investigate the reasons for this surprising result in future experiments.

Conclusions and future work

We have proposed and explored the feasibility of an approach to mining Markov decision processes from student work to automatically generate hints. This approach differs from prior work in authoring tutoring systems by mining actual student data, rather than relying on teachers to add examples the system can learn from. Our tutor can already classify many errors students make. Adding the MDP to this tutor enables it to provide hints. This MDP can constantly learn from new student data. We note that on cold start for a new problem that has no student data, the system will still act as a problem-solving environment, but after even one semester of data is collected, a significant number of hints can be generated. As more data are added, more automated hints can be generated. We have implemented this hint system in the Deep Thought logic tutor and are currently testing whether our hints affect overall learning. In our future work, we will continue to explore ways to learn general rules to build intelligent feedback and help with greater coverage and robustness. For instance, we plan to group students according to their proofs behavior and class performance, and create tailored MDPs for each group of students. In an interesting extension, we are investigating ways to determine the usefulness of a problem step based on its frequency in the MDP. This work will allow us to detect and prevent hints that may guide students to perform steps that others have taken, but that did not contribute to the problem solution. This work on measuring the utility of a problem step can also be used to generate help in ill-defined domains.

We believe the feasibility studies presented in this paper provide an important methodology for predicting the reliability of data-driven methods for deriving feedback and hints for students. These methods address teacher concerns regarding the availability of sufficient and accurate help while addressing the need to support student learning with low-cost, scalable methods.

References

Anderson, J., & Gluck, K. (2001). What role do cognitive architectures play in intelligent tutoring systems? In D. Klahr & S. Carver (Eds.), *Cognition & Instruction: 25 years of progress* (pp. 227–262). Mahwah, NJ: Erlbaum.

Baffes, P., & Mooney, R. J. (1996). A novel application of theory refinement to student modeling. *AAAI-96: Proceedings of the 13th National Conference on Artificial Intelligence* (pp. 403–408), Menlo Park, CA: AAAI Press.

Barnes, T. (2006). Evaluation of the q-matrix method in understanding student logic proofs. In G. Sutcliffe & R. Goebel (Eds.), *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference*, Menlo Park, CA: AAAI Press.

Barnes, T., & Stamper, J. (2007). Toward the extraction of production rules for solving logic proofs, In *Proc. 13th Intl. Conf. on Artificial Intelligence in Education, Educational Data Mining Workshop.* Online proceedings retrieved March 25, 2009, from http://aied.inf.ed.ac.uk/AIED2007/AIED-EDM_proceeding_full2.pdf.

Barnes, T., Stamper, J, Lehman, L., & Croy, M. (2008). A pilot study on logic proof tutoring using hints generated from historical student data. *Proceedings of the 1st Annual International Conference on Educational Data Mining*. Montreal, CA, June 20–21, 2008, Retrieved March 25, 2009, from http://www.educationaldatamining.org/EDM2008/index.php?page=proceedings.

Beck, J., Woolf, B., & Beal, C. (2000). ADVISOR: A machine learning architecture for intelligent tutor construction. In *AAAI-96: Proceedings of the 7th National Conference on Artificial Intelligence* pp. (552–557). Menlo Park, CA: AAAI Press.

Croy, M. J, (2000). Problem solving, working backwards, and graphic proof representation, *Teaching Philosophy 23,* 169–187.

Croy, M., Barnes, T., & Stamper, J. (2007). Towards an intelligent tutoring system for propositional proof construction. In P. Brey, A. Briggle, & K. Waelbers (Eds.), *Proceedings of the 2007 European Computing And Philosophy Conference*, Amsterdam, Netherlands: IOS Publishers.

Joint Task Force for Computing Curricula, ACM, AIS, & IEEE. (2005). Computing curricula 2005. Retrieved March 26, 2009, from http://www.acm.org/education/education/curric_vols/CC2005-March06Final.pdf

Koedinger, K., Aleven, V., Heffernan, N., McLaren, B., & Hockenberry, M. (2004, August). Opening the door to non-programmers: Authoring intelligent tutor behavior by demonstration. *Proceedings of the 7th Intl. Conf. ITS-2004: Intelligent Tutoring Systems.* Maceió, Brazil: Springer.

McKendree, J. (1990, December). Effective feedback content for tutoring complex skills. *Human-Computer Interaction, 5*(4), pp. 381–413.

McLaren, B., Koedinger, K., Schneider, M., Harrer, A., & Bollen, L. (2004, August). Bootstrapping Novice Data: Semi-automated tutor authoring using student log files, In *Proc. Workshop on Analyzing Student-Tutor Interaction Logs to Improve Educational Outcomes*, *Proceedings of the 7th Intl. Conf. ITS-2004: Intelligent Tutoring Systems.* Maceió, Brazil: Springer.

Merceron, A., & Yacef, K. (2005). Educational data mining: A case study. In *12th Intl. Conf. Artificial Intelligence in Education*. Amsterdam, Netherlands: IOS Press.

Mitrovic, A., Koedinger, K. & Martin, B. (2003). A comparative analysis of cognitive tutoring and constraint-based modeling. *User Modeling 2003: Proceedings of the 9th International Conference, UM 2003, Johnstown, PA, USA, June 22–26, 2003, Lecture Notes in Computer Science* (pp. 313–322). Berlin: Springer.

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

Sieg, W. (2007). The AProS Project: Strategic thinking & computational logic. *Logic Journal of IGPL*. Retrieved March 26, 2009, from http://jigpal.oxfordjournals.org/cgi/content/abstract/jzm026v1

Stamper. J. (2006, September). Automating the generation of production rules for intelligent tutoring systems. *Proc. Intl. Conf. Computer-Aided Learning (ICL2006)*, Villach, Austria: Kassel University Press.

Sutton, R., and Barto, A. G. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press.

Van Lehn, K. (2006). The behavior of tutoring systems, *International Journal of Artificial Intelligence in Education 16,* 227–265.