Submitted NATO Conference Book

Student Modeling in an Intelligent Programming Tutor

Albert T. Corbett and John R. Anderson

Psychology Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract: The ACT Programming Tutor helps students as they complete short programming exercises. The tutor is constructed around an ideal student model consisting of several hundred production rules. This model allows the tutor to solve exercises along with the student and serves as an overlay model of the student's knowledge. As the student completes exercises, the tutor maintains an estimate of the probability that student has learned each rule, based on a two-state learning model. These estimates are employed to guide remediation. This paper assesses the predictive validity of this modeling process, and examines the implications for the rules in the ideal student model

Keywords: Student Modeling, Intelligent Tutors, Mastery Learning.

Introduction

One of the promises of intelligent tutoring systems is increased learning rates. Tailoring the learning environment to the individual student should enable the student to obtain a higher level of understanding and performance from a given investment of time. While there are other significant characteristics of learning characteristics, e.g., fostering a student's motivation to spend time on a task, it is difficult to quarrel with this goal. Learning rate can be viewed as the issue of time on task defined at the level of cognitive modeling. If we have an accurate cognitive model of the task and a model of the student's knowledge state, we can optimize learning by (1) guiding the student toward activities that lie on the frontier of the student's knowledge and (2) providing help that mimimizes the student's floundering in acquiring new knowledge. The ACT Programming Tutor (APT) is an intelligent programming environment that is intended to further each of these goals. Much of our research over the past five years has examined the issue of error feedback and floundering [5,6]. In this paper we consider the goal of tailoring activities to the individual student. In particular, we examine the tutor's knowledge tracing mechanism which is intended to optimize remedial practice, allowing the student to achieve mastery. We report a recent assessment of this mechanism and focus on the implications for the cognitive model that underlies the tutor.

The ACT Programming Tutor

The ACT Programming Tutor is a practice environment for students learning to program [3]. It presents exercises that require the student to write short programs and monitors the student's behavior on a symbol-by-symbol basis. The program has Lisp and Prolog modules, which are used to teach a self-paced introductory programming course at Carnegie Mellon University. In this report we focus on the Lisp module. Figure 1 displays the tutor interface, near the start of a Lisp exercise. The exercise description appears in the upper left window and the student's solution is displayed immediately below in the code window, which is similar to a structure editor. The student enters code essentially one symbol at a time, either by typing or through the menu on the lower right. In Figure 1, the student has just entered the operator *car*. The tutor has expanded a template for this function call in the code window, providing balanced parentheses and an editor node or goal symbol (in anglebrackets) that represents an argument to the function. The student will replace this node with another Lisp expression.

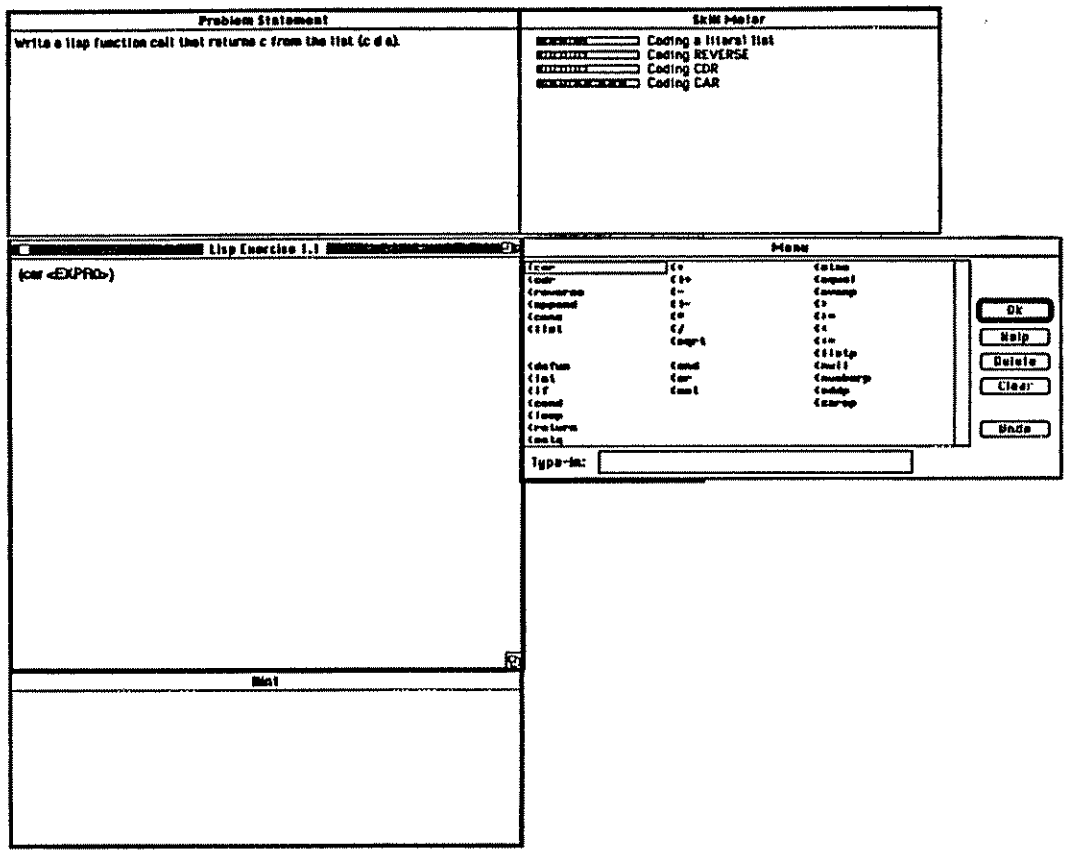

Figure 1. The APT interface near the beginning of an exercise.

The tutor provides immediate feedback on each each code symbol the student enters. If the student makes a mistake, the tutor notifies the student in the help window on the lower left and does not display the symbol. Instead, the student can enter another symbol, ask for help, or

switch to working on another node. Three levels of help are available at each goal node. In response to successive help requests at a node the tutor presents: (1) a description of the current goal, (2) an explanation of how to achieve the goal and (3) a description of the exact action to perform.

The remaining window in the upper right corner is the skill meter, which displays the tutor's model of the student. As described below, the cognitive model that underlies the tutor consists of a set of rules for writing Lisp code. As the student works, the tutor maintains an estimate of the probability that the student has learned each rule in the set, in a process we call *knowledge tracing*. The learning probability for each of these rules is displayed graphically for the student in the skill meter. In Figure 1, four rules have been introduced. The learning probability is approximately 0.80 for one rule and 0.50 for the other three rules. The present report focuses on this knowledge tracing process. A recent assessment indicates that the predictive validity of knowledge tracing is quite good [7]. In this paper we discuss how this knowledge tracing assessment can be used to assess the cognitive model (rule set). In the following sections we describe the curriulum under consideration, the corresponding cognitive model, the tutor's learning and performance model and finally, an evaluation of the cognitive model.

The Curriculum

The knowledge tracing evaluation focused on the first five sections of the APT Lisp curriculum. These five sections introduce two basic data types, atoms (symbols) and lists (symbols grouped in parentheses), function calls (operations) and function definitions. Students learn three extractor functions, *car*, *cdr* and *reverse*, that return information from a list, three constructor functions, *append*, *cons* and *list* that build new lists, and the operator *defun*, which is used to define new functions. An example exercise from each section is displayed in Table 1.

The Cognitive Model

The tutor was developed to test the ACT* theory of skill acquisition [1]. The theory assumes that programming knowledge can be modeled as a set of production rules. The tutor is constructed around an expert system consisting of several hundred such if-then rules for writing programs, called the *ideal student model*. This ideal model plays two roles in the tutor. First, it allows the tutor to solve the exercise step-by-step along with the student in a process we call *model tracing*. As the student enters each code symbol, the tutor attempts to match the action to an applicable rule in the model. If a match is found, the tutor accepts the symbol and updates its internal representation of the problem state. If not, the tutor requires the student to try

another action. Second, in knowledge tracing, the student is represented as an overlay of the ideal model [8]. As the student works through the exercises, the tutor maintains an estimate of the probability that the student has learned each rule in the ideal model.

Table 1

Five example exercises from the APT Lisp tutor.

Section	Exercise
1 Extractors	(car '(c d e))
2 Combiners	(list 'x 'y 'z)
3 Extractor Algorithms	(car (cdr '(horse dog cat)))
4 Function Definitions[a]	(defun second-to-last (lis) (car (cdr (reverse lis))))
5 More Definitions[b]	(defun replace-first (itm lis) (cons itm (cdr lis)))

[a] extractor algorithms, [b] constructor/extractor algorithms

The cognitive model for the present curriculum consists of a total of twenty-one rules. Four rules introduced in the first section govern the coding of the three extractor functions, *car*, *cdr* and *reverse*, and literal lists. The first and last of these rules apply in succession in coding the first exercise in Table 1. Four rules introduced in section 2 govern the coding of the three constructor functions, *append*, *cons* and *list* and literal atoms. The rule that codes a literal atom applies three times in coding the second exercise in Table 1, following an application of the *list* rule. Four rules introduced in section 4 concern the coding of function definitions. One codes the operator *defun*, one codes the name of the new function, a third rule applies for each variable being declared in the parameter list, and the fourth applies each time a variable is referenced in the body of the function.

In section 3 the student learns to code algorithms involving nested extractor functions. For example, in exercise 3 of Table 1, students learn to extract the second element of a list by (1) applying *cdr* to remove the first element of the list, then (2) applying *car* to return the new first element. The cognitive model assumes that the student is essentially learning to reason about nested function calls in this section and that knowledge about each of the three extractor functions will generalize from section 1. As a result, two rules in the model govern the coding

of all extractor algorithms in this section. One codes the outermost function call, e.g., *car* in our example. The second rule applies in coding each nested function call, e.g., *cdr* in our example. In section 4, students learn to code function definitions involving nested extractor functions. The model assumes that the two rules learned in section 3 will generalize to the function body in section 4. In section 5, students learn to code definitions involving constructors and extractors. Three separate rules are introduced that govern the coding of the three constructors in this context, while a single rule governs the coding of an extractor or the initiation of an extractor algorithm as the argument to a constructor.

Finally, three rules model interface actions, specifically the deletion of extra editor nodes. One rule deletes an extra argument node in a constructor function call, one deletes of an extra variable declaration node in the parameter list and one deletes an extra argument node in a call to *defun*. The complete set of rules is displayed in Table 2 below.

The Learning and Performance Model

In addition to a cognitive model, knowledge tracing requires learning and performance assumptions. For this purpose, the tutor assumes a simple two-state learning model with no forgetting. That is, each coding rule is either in the learned or unlearned state. A rule can make the transition from the unlearned to the learned state at each opportunity to apply the rule, but rules do not make the transition in the other direction. Within this framework, the task is to maintain an estimate of the probability that the student has learned each rule, conditionalized on the student's performance. The Bayesian computational procedure used is a variation of a one described by Atkinson [4] and is reported elsewhere [7]. This procedure employs two learning parameters and two performance parameters:

pL_0 the probability a rule is in the learned state prior to the first opportunity to apply the rule (i.e., from reading the text)

pT the probability a rule will make the transition from the unlearned to the learned state following an opportunity to apply the rule

pG the probability a student will guess correctly if a coding rule is not in the learned state

pS the probability a student will slip (make a mistake) if a coding rule is in the learned state

The values of these parameters are estimated empirically from earlier tutor data.

The goal of knowledge tracing is to implement mastery learning. In each section of the curriculum, students complete a set of required exercises that covers the set of rules introduced, then continue working on remedial exercises in the section until the learning probability of each of these rules rules has reached 0.95. However, the underlying model also allows us to predict accuracy at each goal (step) in completing the tutor exercises. The probability of a correct response at a goal is the sum of two probabilities (a) the probability that an applicable production rule is in the learned state times the probability the student will not slip and (b) the probability the rule is not in the learned state times the probability a student will nevertheless guess correctly.

Student Modeling Assessment

The knowledge tracing mechanism satisfied a weak validity test in an early assessment [2]. Students who worked through the tutor exercises with knowledge tracing in operation performed better on posttests than students who only completed the required exercises. In the present research, however, we undertook a more detailed assessment of just the first five sections of the curriculum. We examined the results of forty-one students enrolled in our programming course in a recent semester. Knowledge tracing was in operation and the four learning and performance parameters were held constant across the twenty-one production rules as follows: $pL_0 = 0.50$, $pT = 0.40$, $pG = 0.20$ and $pS = 0.20$. Students completed an average of 14 remedial exercises, with a range of 1 to 38 exercises, in addition to twenty-five required exercises. Since knowledge tracing drives the learning probability for each production rule for each subject to a minimum of 0.95, there is insufficient variability to assess how well these learning probabilities predict posttest performance. Instead, we focused on the model's predictions for accuracy in completing the tutor exercises.

We computed actual error rates across the forty-one subjects for each of the 158 goals in the twenty-five required tutor exercises and correlated these with the predicted error rates derived from the knowledge tracing mechanism. This correlation was reliable, $r = 0.47$, $p < .05$. However, the expected values deviated systematically from the actual values. As a result, we refit the data allowing the four learning and performance parameters to vary across the twenty-one programming rules (best fitting parameter estimates were generated with a curve fitting program). This revised set of parameter estimates yielded a much better fit to the error rate data, $r = 0.85$. One consequence of refitting the data with revised parameters is that the final learning probabilities for the production rules no longer necessarily exceed 0.95. The mean learning probability for the twenty-one rules ranged from 0.89 to 1.0 across the forty-one subjects.

These mean learning probabilities for the rules in the cognitive model correlated reliably with posttest performance, $r = 0.47$, $p < .05$. See [7] for more details.

Parameter Estimates: Implications for the Cognitive Model

The best fitting parameter estimates for each of the twenty-one production rules in the cognitive model are displayed in Table 2. The correlation of empirical and theoretical error rates for each rule is also displayed. The average correlation coefficient in Table 2 is 0.66, suggesting that the model is fitting individual rules reasonably well (while the correlation of 0.85 for the overall set of 158 goals is achieved in part by fitting differences among rules). Nevertheless, inspection of the correlation coefficients and parameter estimates in the table suggest that some of the rules lack psychological validity

Goodness of fit is the strongest indicator of the validity of a rule. By this criterion, two rules are obviously weak: Start-extractor-algorithm and Code-cons-with-extractor-argument(s). However, the parameter pT is also an important indicator of psychological validity. A low estimate suggests that as students practice they are not learning a rule that generalizes across the expected contexts. By this criterion, the three extractor algorithms rules introduced in sections 3 and 5 are suspect. Clearly, the model's assumptions concerning extractor algorithms are incorrect. Similarly, the three constructor rules introduced in section 5 are suspect.

The performance parameters pG and pS, also provide some secondary information. In curve fitting these parameters were constrained not to exceed 0.33 and 0.10 respectively. If these parameter estimates hit these constraints, it suggests that model does not fit the data well. The pattern of r, pT, pG and pS values as a whole suggest that Code-literal-list rule in section 1 and the three combiner rules in section 2 are suspect. This result may reflect difficulties that students have in grasping the hierarchical structure of lists.

Finally, consider the three rules involving variables in section 4 (ncluding the rule that deletes an extra node in the parameter list). The r and pT values for these rules are respectable, but the performance parameter estimates tend to be high. When we plot the learning curve for the rule that declares a variable, as in Figure 1, there is evidence of the difficulty some students are encountering. This figure displays error rates for successive opportunities to apply the rule. We would expect such a learning curve to be monotonically decreasing and error rates do decline fairly regularly across the first four opportunities. However they rise at the fifth and sixth goals. These two goals represent the first exercise in which students must declare two variables in a function (exercise 5 of Table 1) and the sixth goal in particular is where that second variable is declared. The figure suggests that instead of learning the ideal rule - declare

one variable for each argument to the function - some students are learning a buggy rule - always declare one variable - that happens to work for the exercises in section 4. While the curve is generally well fit by the model, the performance parameters are driven up in trying to fit the peak at goal 6 and an analogous peak at goal 11.

Table 2
Parameter Estimates and Correlation Coefficient for
the Twenty-One Rules in the Cognitive Model

	pL_0	pT	pG	pS	r
Section 1					
Code Car	0.53	1.00	0.00	0.03	0.91
Code Cdr	0.93	1.00	0.32	0.00	1.00
Code Reverse	0.78	0.68	0.00	0.00	0.99
Code Literal List	0.15	0.27	0.33	0.10	0.78
Section 2					
Code Append	0.45	0.12	0.00	0.10	0.44
Code Cons	0.73	0.28	0.33	0.01	0.64
Code List	0.56	0.54	0.33	0.10	0.39
Code Literal Atom	0.17	0.43	0.00	0.00	0.92
Delete Extra Argument Node	0.13	0.96	0.03	0.04	0.97
Section 3					
Start Extractor Algorithm	0.31	0.07	0.33	0.10	0.09
Complete Extractor Algorithm	0.76	0.03	0.33	0.00	0.63
Section 4					
Code Defun	0.80	0.99	0.10	0.05	0.73
Declare Function Name	0.86	0.44	0.04	0.07	0.54
Declare Variable in Parameter List	0.43	0.76	0.33	0.09	0.65
Code Variable	0.55	0.64	0.33	0.09	0.73
Delete Extra Parameter Node	0.53	0.09	0.33	0.08	0.67
Delete Extra Top Level Node	0.65	0.72	0.33	0.01	0.94
Section 5					
Code Append with Extractor Arg(s)	0.46	0.03	0.00	0.00	0.76
Code Cons with Extractor Arg(s)	0.56	0.04	0.00	0.10	-0.46
Code List with Extractor Arg(s)	0.12	0.39	0.06	0.10	0.87
Start Extraction Arg to Constructor	0.57	0.12	0.08	0.00	0.65

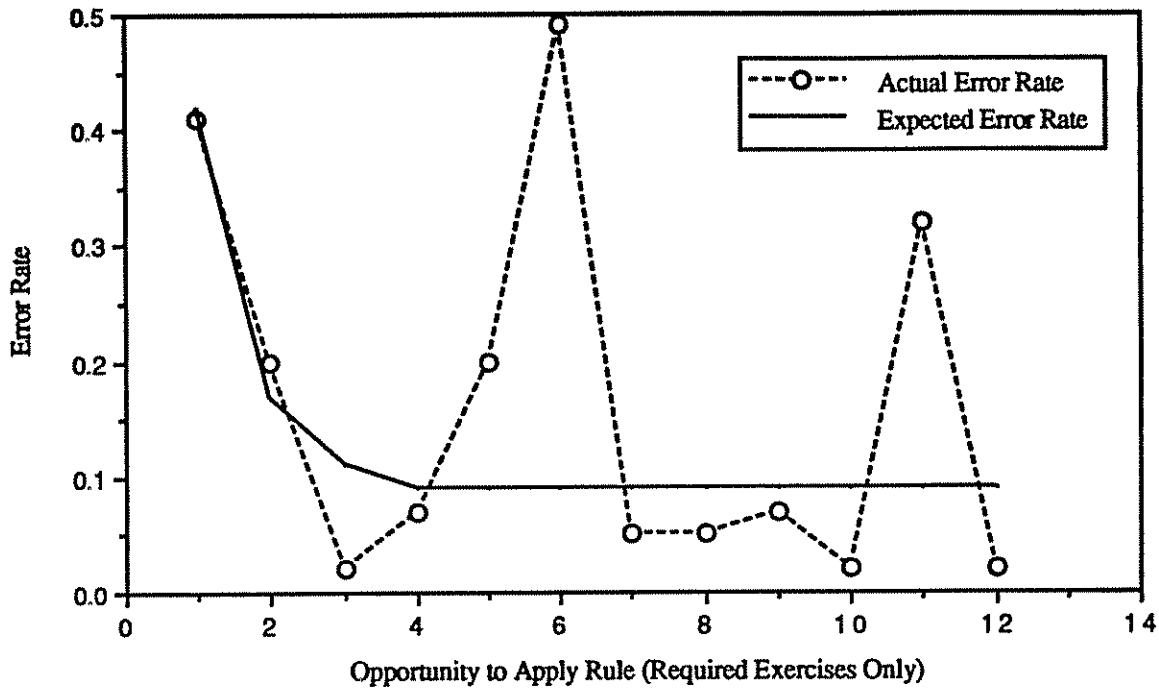

Figure 2. Actual and expected error rate for the rule that declares a variable in a function definition across successive opportunities to apply the rule.

Discussion

The knowledge tracing procedure was quite successful at predicting tutor performance, accounting for about 70% of the variance. However, we have begun revising the cognitive model in light of the present analysis. In particular, we have restructured the literal list and the constructor rules in the early sections, the extractor algorithm rules and the rules concerning variables. The resulting cognitive model, which only covers seven Lisp operators and two data structures, has expanded to thirty-five rules. Clearly even the relatively elementary knowledge acquired in the initial phase of a programming course is complex.

The knowledge tracing procedure was reasonably successful at predicting posttest performance, accounting for 20% of the variance. There are a number of reasons we would not expect a stronger relationship is this assessment. Since it was conducted in the context of a course, for example, the retention interval between the tutor exercises and posttest and the amount of studying for the posttest were uncontrolled. We plan to incorporate retention into knowledge tracing, (i.e., allowing rules to move from the learned to the unlearned state). The current assumption that there is no forgetting may be a reasonable simplication when students are completing a short sequence of tutor exercises, but it is obviously not a general model of learning.

There are other theoretically interesting reasons that knowledge tracing does not strongly predict posttest performance. The posttest environment in this evaluation is substantially different from the tutor environment. Students enter code with a screen editor and can test their solutions with a Lisp interpreter. Thus, while the tutor provides practice in one programming subskill, code generation, students can bring to bear code inspection and debugging knowledge in the posttest. The predictive validity of the tutor's knowledge tracing mechanism can be assessed more rigorously if the posttest environment more nearly approximates the tutor environment. Conversely, it should be possible to apply knowledge tracing to other programming subskills. Knowledge tracing does not depend on a unique solution path for exercises. The exercises in this evaluation have unique solutions, but that does not characterize the tutor exercises more generally. Nor does knowledge tracing strictly depend on immediate error feedback. What is required, is a cognitive model of the task consisting of rules that map onto observable behavior. As a result, it would at least be possible to trace a student's actions in debugging incorrect exercise solutions and to monitor the student's use of any reference materials that are brought on-line.

References

1. Anderson, J.R.: The architecture of cognition. Cambridge, MA: Harvard University Press, 1983.
2. Anderson, J.R., Conrad, F. G. and Corbett, A.T.: Skill acquisition and the Lisp Tutor. Cognitive Science, 13, 467-505, 1989.
3. Anderson, J.R., Corbett, A.T., Fincham, J.M., Hoffman, D. and Pelletier, R.: General principles for an intelligent tutoring architecture. In V. Shute and W. Regian (eds.) Cognitive approaches to automated instruction, Hillsdale, NJ: Erlbaum, (in press).
4 Atkinson, R.C.: Optimizing the learning of a second-language vocabulary. Journal of Experimental Psychology, 96, 124-129, 1972.
5. Corbett, A.T., Anderson, J.R., and Patterson, E.G.: Student modeling and tutoring flexibility in the Lisp Intelligent Tutoring System. In C. Frasson and G. Gauthier (eds.) Intelligent tutoring systems: At the crossroads of artificial intelligence and education, Norwood, NJ: Ablex, 1990.
6. Corbett, A.T. and Anderson, J.R. Feedback control and learning to program with the CMU Lisp Tutor. Paper presented at the Annual Meeting of the American Educational Research Association, 1991.
7. Corbett, A.T. and Anderson, J.R. Student modeling and mastery learning in a computer-based programming tutor. Proceedings of the Second International Conference on Intelligent Tutoring Systems, Montreal, 1992.
8. Goldstein, I.P. The genetic graph: A representation for the evolution of procedural knowledge. In D. Sleeman and J.S. Brown (eds.) Intelligent tutoring systems. New York: Academic Press, 1982.