From: AAAI-84 Proceedings. Copyright ©1984, AAAI (www.aaai.org). All rights reserved.

AUTOMATED COGNITIVE MODELING*

Pat Langley
Stellan Ohlsson
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213 USA

Abstract

In this paper we describe an approach to automating the construction of cognitive process models. We make two psychological assumptions: that cognition can be modeled as a production system, and that cognitive behavior involves search through some problem space. Within this framework, we employ a *problem reduction* approach to constructing cognitive models, in which one begins with a set of independent, overly general condition-action rules, adds appropriate conditions to each of these rules, and then recombines the more specific rules into a final model. Conditions are determined using a discrimination learning method, which requires a set of positive and negative instances for each rule. These instances are based on inferred solution paths that lead to the same answers as those observed in a human subject. We have implemented ACM, a cognitive modeling system that incorporates these methods and applied the system to error data from the domain of multi-column subtraction problems.

1. Introduction

The goal of cognitive simulation is to construct some process explanation of human behavior. Towards this end, researchers have developed a number of methods for collecting data (such as recording verbal protocols, observing eye movements, and measuring reaction times), analyzing these data (such as protocol analysis and linear regression) and describing cognitive processes (such as production systems and neo-Piagetian structures). Unfortunately, there are inherent reasons why the task of cognitive simulation is more difficult than other approaches to explaining behavior. Cognitive simulators must infer complex *process* descriptions from the observed behavior, and this task is quite different from searching for a simple set of equations or even a structural description.

Given the complexity involved in formulating cognitive process models, it is natural to look to Artificial Intelligence for tools that might aid in this process. Along these lines, some researchers have constructed AI systems that generate process models to explain errorful behavior in mathematics. For instance, Burton [1] has described DEBUGGY, a system that diagnoses a student's behavior in the domain of multi-column subtraction problems, and creates a procedural network model of this behavior. In addition, Sleeman and Smith [2] have developed LMS, a system that diagnoses errorful algebra behavior, and formulates process models to explain that behavior.

The task of constructing cognitive models makes contact with two other areas of current interest within Artificial Intelligence. The first of these is concerned with formulating *mental models*. This research has focused on process models of physical phenomena, and though this work faces problems similar to those encountered in cognitive modeling, we will not pursue the connections here. The second area of contact is the rapidly growing field of *machine learning*, and it is the relation between cognitive simulation and machine learning that we will discuss in the following pages. Let us begin by proposing some constraints on the cognitive modeling task that will enable the application of machine learning methods in automating this process.

*This research was supported by Contract N00014-83-K-0074, NR 154-508, from the Office of Naval Research.

2. A Framework for Cognitive Modeling

Before a researcher can begin to construct a cognitive model of human behavior, he must decide on some representation for mental processes. Similarly, if we ever hope to *automate* the formulation of cognitive models, we must select some representation and work within the resulting framework. To constrain the task of cognitive modeling, we will draw on the following hypothesis, first proposed by Newell [3]:

- *The Production System Hypothesis.* All human cognitive behavior can be modeled as a production system.

A production system is a program stated as a set of condition-action rules. Combined with a production system *architecture*, such programs can be used to simulate human behavior. We will not argue here for the psychological validity of the production system approach, except to mention that it has been successfully used in modeling behavior across a wide variety of domains. For our purposes, we are more interested in another feature of production system programs: they provide a well-defined framework for *learning* procedural knowledge. We will discuss this feature in more detail later.

Although the production system hypothesis considerably limits the class of models that must be considered by an automated system, additional constraints are required. Based on years of experience in constructing such models, Newell [4] has proposed a second general principle of human behavior:

- *The Problem Space Hypothesis.* All human cognition involves search through some problem space.

This proposal carries with it an important implication. This is that if we plan to model behavior in some domain, we must define one or more problem spaces for that domain. Such a definition will consist of a number of components:

- A *representation* for the initial states, goal states, and intermediate states in the space;
- A set of *operators* for generating new states from existing states;
- A set of rules that state the legal conditions under which operators may be applied; we will refer to these move-suggesting rules as *proposers*.

For any given task domain, there may be a number of possible spaces, and the cognitive modeler must be willing to entertain each of these in his attempt to explain the observed behavior. However, by requiring that these components be specified, the problem space approach further constrains the task of formulating cognitive process models. The problem space hypothesis also carries with it a second interesting implication: *algorithmic behavior* should be viewed as "frozen" search through a problem space in which the proposers suggest only one move at each point in the search process.

In addition to being psychologically plausible, the combination of the problem space hypothesis and a production system representation has an additional advantage. In this framework, relatively independent condition-action rules are responsible for suggesting which operators to apply. Assuming one's set of operators includes those operators actually used by the subject being modeled, then the task of cognitive modeling can be reduced to the problem of: (1) determining which operators are useful; and (2) determining the conditions under which each operator should be applied. Since the operators are independent of one another, one can divide the cognitive modeling task into a number of simpler problems, each concerning *one* of the operators. We may formulate this as a basic approach to cognitive modeling:

- *The Problem Reduction Approach to Cognitive Modeling.* Taken together, the production system and problem space hypotheses allow one to replace search through the space of cognitive models with several independent searches through much simpler rule spaces.

To reiterate, the problem reduction approach lets one factor the cognitive modeling task into a number of manageable subproblems. Each of these subproblems involves determining whether a given operator was used by the subject, and if so, determining the conditions under which it was used. Once each of these subtasks has been completed, the results are combined into a complete model of the subject's behavior.

This approach is closely related to recent work in the field of machine learning. A number of the researchers in this area – including Anzai [5], Langley [6], and Ohlsson [7] – have applied the problem reduction approach to the task of learning search heuristics. However, this work has focused on acquiring a *correct* search strategy for some domain of expertise. Our main contribution has been to realize that the same basic approach can also be applied to automating the construction of cognitive models, and to explore the details of this application. Now that we have laid out our basic framework for stating process models of cognition, let us turn to one method for implementing the approach.

3. The Automated Cognitive Modeler

As we seen, our approach to cognitive modeling requires two basic inputs: the definition for a problem space (consisting of state descriptions, operators, and proposers) and some information about the behavior of the person to be modeled. This information may take the form of problem behavior graphs, error data, or reaction time measurements. Given this information, the goal is to discover a set of additional conditions (beyond the original legal ones) for each of the proposers that will account for the observed behavior. Fortunately, some of the earliest work in machine learning focused on a closely related problem; this task goes by the name of "learning from examples", and can be easily stated:

- *Learning from Examples.* Given a set of positive and negative instances for some rule or concept, determine the conditions under which that rule or concept should be applied.

A number of methods for learning from examples have been explored, and we do not have the space to evaluate the advantages and disadvantages of them here. However, all of the methods require a set of positive and negative instances of the concept/rule to be learned, so let us consider how such a set can be gathered in the context of automated cognitive modeling (or learning search heuristics).

Recall that we have available a problem space within which the behavior to be modeled is assumed to have occurred. Since the proposers are more general than we would like them to be, their unconstrained application will lead to breadth-first search through the problem space. If the observed behavior actually occurred within this space, then one or more of the resulting paths will successfully "explain" this behavior.

For example, if partial or complete problem behavior graphs are available, then one or more paths will have the observed sequence of operator applications. If only error data are available, then one or more paths will lead to the observed response. Since we have been working primarily with error data, we shall focus on this latter case in our discussion. Presumably, the subject has been observed working on a number of different problems, so that we will obtain one or more "solution paths" for each of these problems. For now, let us assume that only one such path is found for each problem; we will return to this assumption later.

Given a solution path for some problem, one can employ a quite simple method for generating positive and negative instances for each of the rules used in searching the problem space. We may summarize this method as follows:

- *Learning from Solution Paths.* Given a solution path, label moves lying along the solution path as positive instances of the rules that proposed them, and label moves leading one step off the path as negative instances of the rules that proposed them.

This method allows one to transform a solution path into the set of positive and negative instances required for learning from examples. Note that not all moves are classified as desirable or undesirable; those lying more than one step off the solution path are ignored, since these states should never have been reached in the first place. Sleeman, Langley, and Mitchell [8] have discussed the advantages and limitations of this approach in the context of learning search heuristics. The most notable limitation is that one must be able to exhaustively search the problem space, or be willing to chance the possibility of misclassifications, thus leading to effective "noise". Fortunately, in many of the domains to which cognitive simulation has been applied, the problem spaces allow exhaustive search.

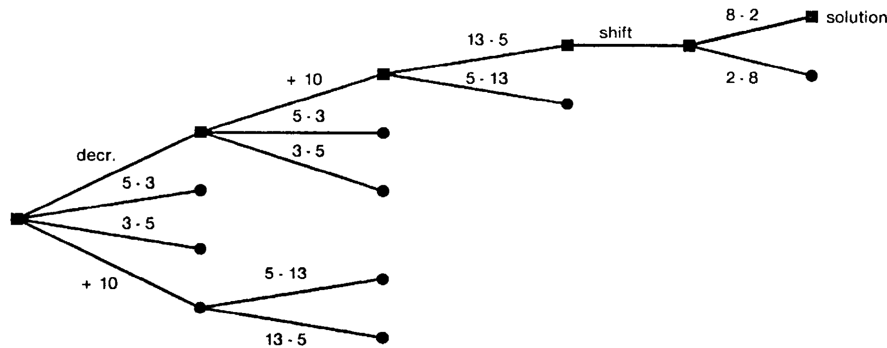

Figure 1. Search tree for the problem 93 – 25 = 68.

Given a set of positive and negative instances for each of the proposers, one can employ some method for learning from examples to determine additional conditions for these rules. The resulting set of more specific rules are guaranteed to regenerate the inferred solution path for each problem, and thus constitute a second level explanation of the observed behavior. Taken together, these rules constitute a cognitive process model stated as a production system.

We have implemented the Automated Cognitive Modeler (ACM), an AI system that instantiates the approach outlined above. Given a set of positive and negative instances for each proposer, the system constructs a discrimination network for each rule, using an approach similar to that described by Quinlan [9]. Once a network has been found for a proposer, it is transformed into a set of conditions which are then added to the original rule. These additional conditions let the proposer match against positive instances, but not against negative ones, and in this sense explain the observed behavior. The details of this process are best understood in the context of an example, to which we now turn.

Table 1. Production system model for the correct subtraction strategy.

find-difference
If you are processing *column1*,
and *number1* is in *column1* and *row1*,
and *number2* is in *column1* and *row2*,
[and *row1* is above *row2*],
[and *number1* is greater than *number2*],
then find the difference between *number1* and *number2*,
and write this difference as the result for *column1*.

decrement
If you are processing *column1*,
and *number1* is in *column1* and *row1*,
and *number2* is in *column1* and *row2*,
and *row1* is above *row2*,
and *column2* is left of *column1*,
and *number3* is in *column2* and *row1*,
[and *number2* is greater than *number1*],
then decrement *number3* by one.

add-ten
If you are processing *column1*,
and *number1* is in *column1* and *row1*,
and *number2* is in *column1* and *row2*,
and *row1* is above *row2*,
[and *number2* is greater than *number1*],
then add ten to *number1*.

shift-column
If you are processing *column1*,
and you have a result for *column1*,
and *column2* is left of *column1*,
then process *column2*.

4. Modeling Subtraction Errors

Our initial tests of ACM have focused on modeling errors in the domain of multi-column subtraction problems. We selected this domain as a testbed because substantial empirical analyses of subtraction errors were available, and because other efforts had been made to model subtraction behavior, to which we could compare our approach. In particular, VanLehn and his colleagues have compiled descriptions of over 100 systematic subtraction errors, and have used this analysis to construct DEBUGGY, a system capable of diagnosing students' subtraction strategies. Although our work relies heavily on this group's analysis of subtraction errors, our approach to automating the process of cognitive modeling differs considerably from their scheme. The most obvious difference is that DEBUGGY made significant use of a "bug library" containing errors that students were likely to make, while ACM constructs explanations of errorful behavior from the same components used to model correct behavior. As a result, ACM carries out no more search in modeling behavior involving multiple bugs than it does in modeling errors due to single bugs; we believe this is a very desirable feature of our approach to cognitive modeling.

In order to model subtraction behavior, ACM must be provided with a problem space for subtraction. This may seem counterintuitive, since we tend to think of subtraction strategies as algorithms, but recall that the problem space hypothesis implies that even algorithmic behavior can be described in terms of "frozen" search. In addition, different students clearly use different subtraction procedures, so one may view this space as the result of generalizing across a set of quite distinct algorithms. In order to define a problem space, we must specify some representation for states, a set of operators for generating these states, and a set of proposers. We will not go into the details of our representation here, and for the sake of clarity, we will focus on only the four most basic operators – **finding a difference** between two numbers in a column, **adding ten** to a number, **decrementing** a number by one, and **shifting attention** from one column to another.* The initial rules for proposing these operators can be extracted from Table 1 by ignoring the conditions enclosed in brackets. We will see the origin of the bracketed conditions shortly.

Although we have applied ACM to modeling errorful subtraction procedures, the system can best be explained by examining its response to correct subtraction behavior. As we have seen, the overly general initial conditions on its proposers leads ACM to search when it is given a set of subtraction problems. Figure 1 shows the system's search on the borrowing problem 93 – 25, when the correct answer 68 is given by the student that ACM is attempting to model. States along the solution path are shown as squares, while other states are represented by circles. Dead ends occur when the program generates a partial answer that does not match the student's result. The system is also given other problems and the student's answers to those problems, and ACM also searches on these until it find acceptable solution paths.

After finding the solution paths for a set of problems, ACM uses the instances it has generated to formulate more conservative proposers that will let it regenerate those paths without search. Let us examine the search tree in Figure 1, and some of the good and bad instances that result. Since most of the interesting learning occurs with respect to the find-difference operator, we shall focus on it here. Upon examining the search tree, we find two good instances of finding a difference, 13 – 5 and 8 – 2 (which lie on the solution path), and six bad instances, two cases of 5 – 3, two cases of 3 – 5, and one case each of 5 – 13 and 2 – 8 (which lie one step off the solution path).

Given these instances and others based on different problems, ACM proceeds to construct a discrimination network that will let it distinguish the desirable cases of the find-difference rule from the undesirable ones. The system iterates through a list of tests, determining which tests are satisfied for each instance. For the subtraction domain, we provided ACM with ten potentially relevant tests, such as whether one number was greater than another, whether one row was above another, whether ten had been added to a number,

*Actually, these operators are not even capable of *correctly* solving all subtraction problems (additional operators are required for borrowing from zero, as in the problem 401 – 283), and they are certainly not capable of modeling all buggy subtraction strategies. However, limiting attention to this set will considerably simplify the examples, so we ask the reader to take on faith the system's ability to handle additional operators.

and whether a number had already been decremented. For example, the negative instance 5 − 3 satisfies the **greater** test, since 5 is larger than 3, but fails the **above** test, since the 5's row is below the 3's row.

Given this information, ACM determines which of its tests has the best ability to discriminate positive from negative instances. In determining the most discriminating test, ACM computes the number of positive instances matching a given test (M_+), the number of negative instances failing to match that test (U_-), the total number of positive instances (T_+), and the total number of negative instances (T_-). Using these quantities, ACM calculates the sum $S = M_+/T_+ + U_-/T_-$, and computes $E = \text{maximum}(S, 2 - S)$. The test with the highest value for E is selected.

In a 20 problem run involving the correct subtraction strategy, the **greater** test achieved the highest score on the function E, although the **above** test scored nearly as well. As a result, ACM used the former test in the top branch of its discrimination tree. Since all of the positive instances and some of the negative instances satisfied the **greater** test, the system looked for another condition to further distinguish between the two groups. Again the most discriminating test was found, with the **above** relation emerging as the best. Since these two tests completely distinguished between the positive and negative instances, ACM halted its discrimination process for the find-difference rule, and moved on to the next proposer.

Once it has generated a discrimination network for each of its proposers, ACM translates these networks into condition-action rules. To do this for a given network, it first eliminates all branches leading to terminal nodes containing negative instances. For each of the remaining terminal nodes, ACM constructs a different variant of the proposer by adding each test as an additional condition. Thus, if more than one terminal node contains positive instances, the system will produce a disjunctive set of condition-action rules to represent the different situations in which an operator should be applied. Once it has generated the variants for each proposer, ACM combines them into a single production system model. This program will regenerate the student's inferred solution paths without search, and can thus be viewed as a cognitive simulation of his subtraction strategy. Table 1 presents the rules that are generated when correct subtraction behavior is observed; the conditions enclosed in brackets are those added during the discrimination process.

Now that we have considered ACM's discovery methods applied to modeling the correct subtraction algorithm, let us examine the same methods when used to model a buggy strategy. Many subtraction bugs involve some form of failing to borrow. In one common version, students subtract the smaller of two digits from the larger, regardless of which is above the other. In modeling this errorful algorithm, ACM begins with the same proposers as before (i.e., the rules shown in Table 1, minus the bracketed conditions). If we present the same subtraction problems as in the previous example, we find that the buggy student produces the incorrect answer 93 − 25 = 72, along with similar errors for other borrowing problems. As a result, the solution path for the borrowing problem shown in Figure 2 differs from that for the same problem when done correctly, shown in Figure 1. In contrast, the student generates the correct answers for non-borrowing problems, such as 54 − 23 = 31. As before, ACM's task is to discover a set of variants on the original proposers that will predict these answers.

Table 2. Model for the "smaller from larger" subtraction bug.

find-difference
If you are processing ***column1*****,**
and ***number1*** **is in** ***column1*** **and** ***row1*****,**
and ***number2*** **is in** ***column1*** **and** ***row2*****,**
and ***number1*** **is greater than** ***number2*****,**
then find the difference between ***number1*** **and** ***number2*****,**
and write this difference as the result for ***column1*****.**

shift-column
If you are processing ***column1*****,**
and you have a result for ***column1*****,**
and ***column2*** **is left of** ***column1*****,**
then process ***column2*****.**

In the correct subtraction strategy, the decrement and add-ten operators are used in problems that require borrowing. However, the solution path for the borrowing problem shown in Figure 2 includes only the find-difference and shift-column operators. Apparently, the student is treating borrowing problems as if they were non-borrowing problems, and the student model ACM develops should reflect this relationship. As before, the system uses the solution paths it has inferred to produce positive and negative instances. As in the previous run, only positive instances of the shift-column operator were found, indicating that its conditions need not be altered. And since both positive and negative instances of the find-difference rule were noted, ACM called on its discrimination process to determine additional conditions for when to apply this operator. The major difference from the earlier run was that only *negative* instances of the add-ten and decrement operators are found. This informed ACM that these rules should not be included in the final model, since apparently the student never used these operators.

For this idealized student, ACM found the **greater** test to have the best discriminating power. However, the **above** test, which was so useful in modeling the correct strategy, does not appear in the final model. In fact, the **greater** test completely discriminated between the positive and negative instances, leading ACM to a very simple variant

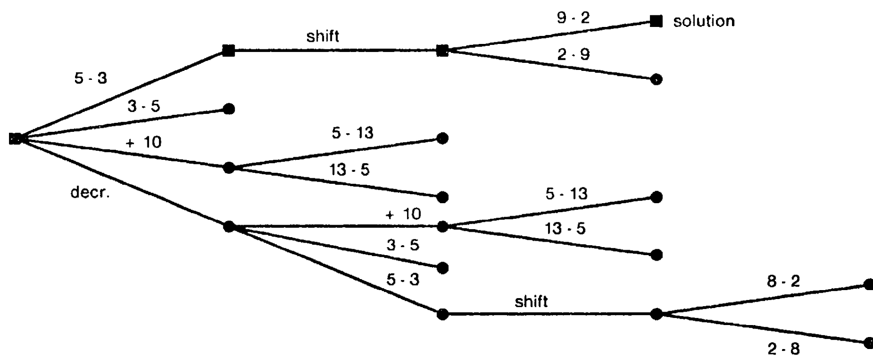

Figure 2. Search tree for the problem 93 − 25 = 72.

of find-difference rule. This was because the idealized student was always subtracting the smaller number from the larger, regardless of the position, and this is exactly what the resulting student model does as well. Table 2 presents the variant rules that ACM generated for this buggy strategy. This model is very similar to that for the correct strategy, except for the missing condition in the find-difference rule, and the notable absence of the rules for decrementing and adding ten, since these are not needed.

ACM has been implemented on a Vax 750, and successfully run on a number of the more common subtraction bugs. Table 3 presents eleven common bugs reported by VanLehn [10], along with their observed frequencies. ACM has successfully modeled each of these bugs, given idealized behavior on a set of 20 representative test problems. A number of these bugs involve borrowing from zero, and so required some additional operators beyond those described in the earlier examples. These operators shift the focus of attention to the left or to the right, in search of an appropriate column from which to borrow. Introducing these operators considerably expanded the search tree for each problem, though ACM was still capable of finding a solution path using exhaustive search.

Table 3. Subtraction bugs successfully modeled by ACM.

BUG	EXAMPLE	FREQUENCY
CORRECT STRATEGY	81 − 38 = 43	
SMALLER FROM LARGER	81 − 38 = 57	124
STOPS BORROW AT 0	404 − 187 = 227	67
BORROW ACROSS 0	904 − 237 = 577	51
0 − N = N	50 − 23 = 33	40
BORROW NO DECREMENT	62 − 44 = 28	22
BORROW ACROSS 0 OVER 0	802 − 304 = 408	19
0 − N = N EXCEPT AFTER BORROW	906 − 484 = 582	17
BORROW FROM 0	306 − 187 = 219	15
BORROW ONCE THEN SMALLER FROM LARGER	7127 − 2389 = 5278	14
BORROW ACROSS 0 OVER BLANK	402 − 6 = 306	13
0 − N = 0	50 − 23 = 30	12

5. Discussion

In evaluating the problem reduction approach to cognitive modeling and its implementation in ACM, we must examine three characteristics of the approach – generality, potential difficulties, and practicality. On the first of these dimensions, ACM fares very well. One can readily see the system being used to model behavior described in terms of a problem behavior graph; in fact, this task should be considerably easier than working only with error data, since the process of inferring solution paths will be much more constrained. The approach might even be adapted to reaction time data, though this would certainly be a more challenging application.

However, there are some difficulties with our approach to automating the construction of cognitive models, relating to the three levels at which explanation occurs in the system. First, it is possible that a subject's behavior can be explained in terms of search through more than one problem space. We have avoided this issue in the current system by providing ACM with a single problem space. However, we have described elsewhere [11] our progress in extending the system to handle multiple spaces, and we plan to continue our work in this direction. Second, it is possible that more than one solution path can account for the observed behavior. The current version simply selects the shortest path, but more plausible heuristics are desirable. However, this problem is greatest when only error data are available; providing ACM with additional information about the order of operator application (a partial problem behavior graph) eliminates this ambiguity. Finally, for some sets of positive and ngative instances, two or more tests may appear to be equally discriminating. The current system selects one of these at random, but future versions should be able to generate diagnostically useful problems to resolve the conflict.

In terms of practicality, the existing version of ACM does not operate quickly enough to be useful in diagnosing student behavior in the classroom. For a set of 20 subtraction problems, the system takes some 2 CPU hours to generate a complete cognitive model. However, most of the effort occurs during the search for solution paths, which can be as long as 20 steps for a five-column subtraction problem. There are many domains which involve substantially smaller spaces, and for these ACM's run times should be much more acceptable. In addition to continuing to test the system on subtraction, our future work will explore the ACM's application to other domains, showing the approach's generality and its practicality in automating the process of modeling cognitive behavior.

References

1. Burton, R. R. Diagnosing bugs in a simple procedural skill. In *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown, Eds., Academic Press, London, 1982.

2. Sleeman, D. H. and Smith, M. J. "Modeling students' problem solving." *Artificial Intelligence 16* (1981), 171-187.

3. Newell, A. and Simon, H. A.. *Human Problem Solving*. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

4. Newell, A. Reasoning, problem solving, and decision processes: The problem space hypothesis. In *Attention and Performance*, R. Nickerson, Ed.,Lawrence Erlbaum Associates, Hillsdale, N. J., 1980.

5. Anzai, Y. Learning strategies by computer. Proceedings of the Canadian Society for Computational Studies of Intelligence, 1978, pp. 181-190.

6. Langley, P. Learning effective search heuristics. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 1983, pp. 419-421.

7. Ohlsson, S. A constrained mechanism for procedural learning. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 1983, pp. 426-428.

8. Sleeman, D., Langley, P., and Mitchell, T. Learning from solution paths: An approach to the credit assignment problem. AI Magazine, Spring, 1982, pp. 48-52.

9. Quinlan, R. Learning efficient classification procedures and their application to chess end games. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds., Tioga Press, Palo Alto, CA, 1983.

10. VanLehn, K. "Bugs are not enough: Empirical studies of bugs, impasses, and repairs in procedural skills." *Journal of Mathematical Behavior 3* (1982), 3-72.

11. Ohlsson, S. and Langley, P. Towards automatic discovery of simulation models. Proceedings of the European Conference on Artificial Intelligence, 1984.