A Machine Learning Approach to Student Modeling

Pat Langley
Stellan Ohlsson
Stephanie Sage

CMU-RI-TR-84-7

The Robotics Institute
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213 USA

May 1, 1984

This research was supported by Contract N00014-83-K-0074, NR 154-508, from the Personnel and Training Research Program, Psychological Sciences Division, Office of Naval Research. Approved for public release; distribution unlimited. Reproduction in whole or part is permitted for any purpose of the United States Government.

We would like to thank Derek Sleeman and Kurt VanLehn for numerous discussions about approaches to the student modeling problem, as well as John Laird and John Anderson for comments on an earlier draft. Pat Langley was responsible for implementing the ACM system, Stephanie Sage debugged and tested the system in the subtraction domain, and Stellan Ohlsson proposed the extensions to ACM discussed at the end of the paper.

Table of Contents

1. Introduction

In recent years, the methodology of cognitive simulation has been applied to a problem of great practical interest – the modeling of students' errors in school work. Given some record of the pupil's work in arithmetic or some other domain, one constructs a simulation model of his strategy which will explain his behavior. The concept of a *buggy* procedure is central to such models. A buggy procedure is a strategy for some task domain which is applicable to problems in that domain, but which does not solve all of these problems correctly. In terms of the computer metaphor, a buggy procedure is a "program" which "runs", but which does not deliver the right "output". A "bug" is any feature of a procedure which causes it to produce erroneous answers; "buggy thinking" refers to the activity of applying a buggy procedure.

Buggy procedures are important to psychology because human learning is gradual even in task domains where the dividing line between "correct" and "incorrect" answers is sharp, as in school mathematics. The number, frequency, and severity of errors decrease gradually over time. The currently favored hypothesis about this phenomenon, which is also fundamental to the research reported here, is that students traverse a sequence of buggy procedures on their way to a correct procedure. The pedagogical interest of buggy models comes from the hope that (in conjunction with a learning theory) they can be used as a basis for remedial instruction. However, the construction of simulation models is a time-consuming task. For such models to be practically useful, there must be some way to automate their construction; this is generally known as the *student modeling* problem.

The modeling of buggy thinking poses several problems. First, there are the difficulties associated with any effort to simulate human cognition: the choice of a psychologically motivated representation, the proper representation of capacity limitations, and so forth. Second, even after one has made these higher level commitments, many possibilities remain to be considered, since the set of possible buggy procedures for any task domain is very large. To model buggy thinking, one must systematically consider this set of possibilities, and determine which procedure best accounts for the student's behavior. In some sense, this is equivalent to automating the process of psychological theory formation, though in a restricted domain. Thus, any novel approaches for addressing this challenging problem would be welcome.

We believe that the rapidly developing field of *machine learning* offers a methodology for approaching the task of student modeling. Researchers in this field have proposed several different *learning mechanisms*, that is, methods for changing a procedure to yield different behaviors. These mechanisms are usually thought of as methods for *improving* a procedure, i.e., for making it produce correct answers. However, this need not be so. A general learning system should be able to acquire buggy procedures as easily as correct ones, provided it is given buggy behavior to imitate. If so, then one should be able to use standard machine learning methods to aid in the process of constructing models of student behavior, whether that behavior is correct or incorrect.

The reader is cautioned to keep in mind the two distinct ways in which "learning" enters into the current discourse. On the one hand, there is the learning process of the pupil being modeled. On the other hand, there is the "learning" process which a modeling program goes through in its efforts to construct a valid model for that pupil. These two processes must be kept distinct. Although the human learning process is inherently interesting and has considerable relevance to education, we will not propose a theory of human learning in this paper. Rather, we will employ techniques from machine learning as tools for automating the construction of student models.

Below we review some previous efforts to automate the construction of computer models of buggy thinking. After this, we state our analysis of the student modeling problem, which is based on a view of student modeling as a special case of the task of learning search heuristics. Next we describe ACM, a

computer program which implements this analysis, along with its behavior in the subtraction domain. We then examine some weaknesses of the current system, and propose some extensions to deal with these issues. In the final section, we evaluate our approach along the dimensions of generality, psychological validity, and practical utility.

2. Previous Research on Student Modeling

Before moving on to describe our approach to student modeling, we should review some of the earlier work in this area. It is not surprising that much of the research on student modeling has focused on mathematical skills, since these have a number of advantages over other school subjects. First, domains such as arithmetic and algebra involve relatively algorithmic procedures, so one can easily construct computer models of the correct strategy. Second, students appear to treat these subjects in a syntactic manner, so that cognitive modeling techniques developed on abstract puzzle solving tasks can be easily applied. Finally, despite their transparency to most educated adults, these subjects cause a great deal of trouble for students, resulting in an interesting menagerie of buggy behaviors on which to test student modeling systems. Accordingly, all of the work we will review has been carried out in the context of mathematical skills, and we have tested our own system in the same area.

2.1. The DEBUGGY System

Some of the earliest work on modeling mathematical skills was carried out by Brown and Burton (1978). These researchers focused on the domain of multi-column subtraction problems, and found a variety of strategies that deviated from the standard algorithm. Although the traditional algorithm may seem simple to most adults, Brown and Burton showed its complexity by representing all of its components and subcomponents in a hierarchical procedural network. Errorful behavior was explained by replacing one or more of these components with an errorful or buggy component. Organizing the data in this way, they identified over 100 buggy components that accounted for the majority of student behavior.

Building on this framework, Burton (1982) proceeded to implement DEBUGGY, a student modeling system for diagnosing the cause of subtraction errors. Given a set of test problems and a student's answers to those problems, DEBUGGY went through its list of buggy components, evaluating each in terms of the answers it predicted. Predictions were made by actually running each buggy procedure on the test problems. If no single bug accounted for enough of the errors, bugs accounting for some of the mistakes were composed into two, three, and four-bug combinations. The system relied on a generate-and-test strategy for evaluating individual bugs, but used a more sophisticated method for deciding which compound hypotheses to examine. Using this strategy, DEBUGGY was successful in modeling a high percentage of students in the subtraction domain.

During its diagnostic process, DEBUGGY relied on a user-supplied data-base of bugs, which we will call a *bug library*. The bugs in this library were discovered by the researchers themselves, through extensive analyses of students' answers on subtraction tests. The resulting list of bugs is certainly an impressive accomplishment, and we will return to this list in evaluating our own student modeling system. However, the ability to generate new buggy components would have been a definite asset, and as we shall see, more recent research has addressed this issue.

2.2. Production System Models of Subtraction

Young and O'Shea (1981) have taken a rather different approach to explaining the origin of subtraction errors. Instead of representing arithmetic strategies as procedural networks, these researchers employed a production system formalism. They represented the correct subtraction procedure as a set of condition-action rules. Errors were explained by the removal of one or two rules, or by the inclusion of a few new rules.

Because these rules were relatively independent of one another, the system would continue to run after such modifications had been made, but would lead to incorrect solutions. Using this approach, Young and O'Shea were able to model the most common subtraction bugs observed by Brown and Burton.

These researchers did not attempt to implement an actual student modeling system. However, their approach does suggest some design features that would be useful in constructing such a system. The notion of a modular production system is a very attractive one, and the idea of accounting for errors through missing components is very elegant, since it allows one to avoid the need for an explicit bug library. Unfortunately, they were forced to introduce a few incorrect rules to account for some of the bugs. These would have to be provided by the programmer, as in DEBUGGY, or one must find a way to generate such variant rules automatically.

2.3. The Leeds Modeling System

Sleeman and Smith (1981) have carried out research in much the same spirit as DEBUGGY, though there are some important differences between the two projects. In this case, the domain was algebra problems involving a single variable, and the result was the Leeds Modeling System (LMS). Like Brown and Burton's system, LMS accounted for errorful behavior by replacing components in a correct procedure with buggy ones. However, rather than representing algebra strategies as procedural networks, they used production rules for transforming one equation into another. The researchers used the term "mal-rules" to refer to the buggy counterparts of the correct rewrite rules.

LMS presented a student with problems designed to distinguish between a single algebra rule and the mal-rules that might take its place. As more rules were incorporated into the model, more complex problems were presented to determine the presence of additional rules, until a complete model of the student's algebra strategy had been constructed. Thus, LMS can be viewed as carrying out a heuristic search through the space of possible procedures, with each search step identifying which rule to add to the student model. Like DEBUGGY, Sleeman and Smith's system required the user to specify a bug library.

In a more recent paper, Sleeman (1982) has described an extension of LMS that is capable of discovering new mal-rules to account for behavior that cannot be modeled using known rules. This approach involves searching backwards from a given or observed answer, using the existing (correct and incorrect) re-write rules. If a complete path from the answer to the problem as given can be constructed using only the correct algebraic rules, then the solution is correct. If a path can be found using the correct rules plus the already discovered mal-rules, then the error can be modeled using existing components. However, if the backward search fails to connect the answer with the problem as given, then a new mal-rule is postulated which connects the problem as given with the first state in some path which led to the observed answer.

This work has considerable potential, since it does not rely on the user to provide a complete bug library at the outset of the modeling process. One of its limitations is that only a single new mal-rule can be discovered at a time. If the student is actually using two mal-rules in solving a problem, and if none of these mal-rules has previously been encountered, the system will learn a single mal-rule that masks the true behavior. Thus, the resulting student model will have good predictive power, but will not be very useful for a teacher attempting to design remedial lessons. However, a careful tutor can avoid such confounding by appropriate selection of sample problems.

2.4. Repair Theory and Step Theory

As we have noted, one of the limitations of the DEBUGGY system was that it required a user-specified list of bugs. In addition, more than 100 such bugs were eventually found by painstaking analysis of subtraction data. In response to this bewildering array of behaviors, Brown and VanLehn (1980) developed a

theory to account for the *origin* of these bugs. They called this *repair theory*, and implemented a cognitive simulation program based on the theory that was capable of predicting many of the observed subtraction bugs. Repair theory claimed that arithmetic skills are represented not by a procedural network, but as a goal-oriented production system.

In this framework, buggy procedures were generated in a number of steps. First, a set of *deletion operators* was applied to the correct procedure, thus generating a set of procedures which were incomplete in various ways. Second, each incomplete procedure was run on a set of subtraction problems. While such an incomplete procedure was running, it could encounter a situation where it could not proceed, or an *impasse*. In response to such an impasse, problem solving was used to *repair* the broken procedure. A repair was a local "patch" designed to overcome the impasse and allow the procedure to complete the problem. However, since the repair was local, it could lead to erroneous answers. Thus, by performing all plausible deletions, and then responding to each impasse by repairing the incomplete procedure in all plausible ways, Brown and VanLehn's implementation of repair theory constructed a set of buggy subtraction strategies. The path taken in generating each bug was interpreted as explaining the origin of that bug. These bugs could then be used by DEBUGGY in the diagnosis process.

One drawback of repair theory was that its deletion operators were not very psychologically plausible. VanLehn (1983) has responded to this problem by developing *step theory*, a model of the learning process that accounts for the origin of faulty, impasse-producing procedures without reference to deletion operators. This theory (again implemented as a cognitive simulation) also represents skills in terms of a goal-based production system. In VanLehn's framework, procedures are acquired incrementally in response to successive lessons from a textbook, where each lesson consists of a set of solved sample problems. The acquisition of new goal structures, and subroutines for achieving them, relies on the notion of parsing.

Upon encountering a new sample solution, the learning system uses its current goal hierarchy (initially this is very simple) in attempting to parse the solution sequence. If a complete parse can be found, then the system is capable of solving the problem in its current form, so no modifications are necessary. However, if the parse fails, the system hypothesizes extensions to its goal hierarchy that would have allowed it to succeed. Since many such extensions are usually possible, the alternatives are rank ordered according to a set of general principles, such as preferring simpler goal hierarchies to more complex ones. The most desirable extension is then incorporated into the system's "grammar", so that it will be able to solve similar problems in the future.

The resulting modified procedure is then passed to the implemented version of repair theory. This runs the procedure on a set of test problems, notes any impasses that result, finds the appropriate repairs, and generates a set of bugs. Step theory leads to incorrect procedures in cases where a non-representative set of sample problems are given to the system, so that spurious relations are incorporated into the new goal hierarchy. Thus, for a particular lesson sequence, it makes specific predictions about the bugs that may be observed (and those which should not be observed) for a student presented with those lessons.

Both repair theory and step theory are intended as psychological models of the origin of bugs, and together they are impressive accomplishments on this dimension. However, they do not have as much to contribute toward the goal of a practical yet general student modeling system. One could imagine using a combination of step theory, repair theory, and DEBUGGY to model a particular student, but the requirement that one know the instructional history of that student is a serious drawback. Given our concern with student modeling, we should look elsewhere for new approaches to this problem.

2.5. Comments on the Previous Research

In summary, though some interesting work has been done on the student modeling task, it remains an open problem for which improved methods may be found. We saw that the early DEBUGGY work, though contributing much to the student modeling paradigm in terms of both methods and data, required the user to provide an explicit bug library. Young and O'Shea attempted to deal with this drawback by explaining a number of subtraction bugs in terms of missing rules, but this attempt was only partially successful. Brown and VanLehn's repair theory and step theory provided a more complete explanation of bug origins, but it was also much more complex than Young and O'Shea's theory. Neither group of researchers has proposed any more sophisticated way of diagnosing a particular student's behavior than the approach used in DEBUGGY.

Sleeman and Smith described a more sophisticated heuristic search through the space of procedures, and this approach has definite advantages. However, the initial version of LMS relied on a user-specified list of buggy components, just as DEBUGGY did. More recent work by Sleeman has focused on the generation of new mal-rules, but this approach encounters difficulty when students have multiple bugs. In this paper, we describe an approach to student modeling that overcomes these difficulties. Of course, there are many useful ideas in the earlier work, and we will draw upon them whenever possible. In particular, we will rely on Sleeman's notion of heuristic search through a procedure space. We will also employ Young and O'Shea's notion of modular production systems, and the idea that some bugs result from missing rules. The main difference from the previous work lies in our focus on the generation of new buggy components as the central problem in student modeling. This gives our approach a rather different flavor than earlier ones, which we hope the reader will find refreshing.

3. An Alternate Approach to Student Modeling

In this section, we describe an alternate approach to student modeling. This approach is based on the insight that the student modeling task can be viewed as a special case of the task of learning procedures. This more general task can be stated as follows:

Given: (a) a task domain, (b) a set of problems within that domain, (c) and a set of answers to those problems;

Find: a procedure that is (d) applicable to any problem in the domain, and (e) when applied to the given problems, delivers the observed answers.

In the case of student modeling, the observed answers are those produced by a student. If this student uses the generally accepted "correct" strategy, the student modeling task will be equivalent to the task of learning correct procedures. However, if the student is using a buggy strategy, then one must learn an "incorrect" procedure to account for the student's behavior.[1] In this section, we describe the approach to student modeling that has resulted from this insight. Since there are many directions that such a "learning-based" student modeling system might take, we will devote this section to discussing the major design decisions we have made in constructing our system. In the following section, we provide additional details of the system in the context of its application to the subtraction domain.

---

[1]This basic idea first arose from discussions between Derek Sleeman and Pat Langley, in attempting to understand the relations between the former's work on student modeling and the latter's work on learning search heuristics.

3.1. Student Modeling as Machine Learning

Our first decision was to view the student modeling task as a *heuristic search* problem. In heuristic search (Newell, 1972), one is given an *initial state* from which to begin searching, along with a set of *operators* for generating new states. Taken together, these components define a *problem space* which can be searched in a systematic way. In the case of student modeling, one must search a space in which each state is a *procedure* that is capable of solving (or at least attempting to solve) a class of problems. Many heuristic search methods also require some *test* to determine when the goal state has been reached. In the student modeling task, the goal is to find some procedure that both predicts and explains the answers produced by the student.

In order to carry out heuristic search through any problem space – whether it is a space of procedures, a space of bugs, or any other – one must specify a number of components, including: (a) the representation of individual states in the space; (b) the initial state from which search begins; and (c) a set of operators for moving from one state to another. These components *define* a particular problem space. In addition, we must specify some search control method, which determines which operators to apply at each point in the search, and the states to which they should be applied. In the following pages, we will describe the procedure space we have chosen to search, in terms of these components.

Since the goal of student modeling is to develop a cognitive model of the student's behavior, the states in our procedure space must be psychologically plausible procedures. This also constrains the operators for generating new states, since these must produce plausible procedures. Viewed in this way, we can make a clean separation between the psychologically relevant components of a student modeling system, and the rest of its components. The representation of individual states in the problem space must conform to psychological considerations, since this constitutes a theory of the mental representation of cognitive skills. However, the other components – the search control and the operators – can be implemented in any way that is computationally efficient.

Expressed differently, the student modeling problem separates into two distinct research problems, one belonging to Cognitive Psychology and the other to Artificial Intelligence (AI). The psychological problem is to define the problem space, thus determining the mental representation of procedures. In other words, one must decide on the "programming style" of the mind. The AI problem is: given a space of possible procedures, and given a test to decide when the goal procedure has been found (i.e., a set of answers to be reproduced), implement a means for searching that space in an efficient manner.

3.2. Algorithms as Constrained Search

Let us first address the issue of cognitive skills and their representation. Along these lines, our first basic decision is to adopt Newell's Problem Space Hypothesis (1980). For our present purposes, we can formulate this hypothesis as follows:

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

The significance of this decision in the current context is that (a) we make contact with a large amount of cognitive research which has shown that the problem space concept is a fruitful one in the analysis of complex cognitive processes, and that (b) it commits us to view *algorithms* as *search procedures*. Thus, representing a student's knowledge of a domain involves defining a set of basic arithmetic operators, along with a set of rules or heuristics for when to apply those operators.

This forces us to make a second decision about how to represent this heuristic knowledge. AI has developed two alternate representations for such heuristics. The first involves the notion of an *evaluation function*, and has been extensively used in game-playing domains, where look-ahead tends to be very

important. The second involves the notion of *productions* or condition-action rules, and has been widely used in cognitive simulations of human behavior in a variety of domains. One can imagine the problem space hypothesis being implemented in either of these basic paradigms. However, many arguments have been given for using production system formalisms in modeling human behavior (Newell, 1972). This assumption is significant enough to be stated as another hypothesis:

The Production System Hypothesis. All human cognitive skills can be modeled as a production system.

Since we are concerned with developing psychologically plausible models of students' behavior, the production system representation seems a natural choice.[2] These two representational decisions – viewing behavior as search through a problem space, and explaining behavior in terms of production systems – have dramatic implications for the manner in which we approach the student modeling task.

The reader is urged to distinguish the two ways in which the search concept enters into the present work. On the one hand, we are making the psychological assumption that algorithmic skills are encoded as search procedures which traverse states in a problem space. On the other hand, we are viewing the student modeling task in terms of a higher level search through a space of possible procedures.

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 "move-proposing" rules are responsible for suggesting which operators to apply; we will use the term *proposers* for these rules. Assuming our set of operators includes those operators actually used by the student (we will return to this assumption later), then the task of learning procedures (and thus the student modeling task) can be reduced to the problem of: (1) learning which operators are useful; and (2) learning the conditions under which these operators should be proposed. Since the proposers are independent of one another, one can transform the student modeling task into a number of much simpler subproblems, each concerning *one* of the proposers/operators. Each of these subproblems involves determining whether a given operator was used by the student, 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 viable model of the student's behavior. Since we have described earlier systems in terms of search, we should note that this approach can be viewed as employing the *problem reduction* approach to problem solving described by Nilsson (1971).

Of course, the problem of mental representation is not solved or exhausted by deciding to abide by the problem space and production system hypotheses. For instance, one must still decide on a particular problem space in which to model students, and one must choose a particular production system architecture in which to implement one's models. We will address these details later, in the context of a specific application domain. For the moment, let us move on to other matters.

3.3. Selecting an Initial Procedure

As we have seen, any problem space is defined by an initial state, along with one or more operators for generating new states. In addition, it is often useful to specify some test for recognizing the goal state. In procedure learning and student modeling, this test is obvious – the goal procedure must generate the observed answers for the set of sample problems. However, deciding on an initial state and operators is more problematic. The correct procedure might seem to be a natural choice for the initial state, since one could argue that most bugs are only slight contortions on the correct strategy. One could then move through the

---

[2] The interested reader should see Laird's (1983) discussion of production systems as a framework for implementing the problem space hypothesis.

procedure space by modifying the correct procedure in various ways. This approach is implicit in the DEBUGGY program, where the procedural network mirrors the correct breakdown of the subtraction skill, and the buggy components are viewed as versions of the corresponding correct component. It is more explicit in repair theory, where deletion operators applied to the correct procedure produce the incomplete procedures which lead to impasses. It is also explicit in the work by Young and O'Shea, and is present in Sleeman's approach, where each correct rewrite rule has one or more associated mal-rules.

The above approach is closely linked to the view of procedures as algorithms. At some level, this makes excellent sense, since mathematics students always appear to behave algorithmically. In contrast, the Problem Space Hypothesis advises us to view procedures as search methods, with algorithms being the special case in which search is so constrained that only one move is proposed at each point in the search process. Viewing algorithmic behavior in terms of search leads us to expand our procedure space to include non-algorithmic methods, even though we may never see such methods used by students.

In turn, this view suggests a different choice for the initial state in the procedure space. Rather than beginning with an algorithmic, completely constrained search scheme, suppose we start searching from a procedure composed of rules that are as loosely constrained as possible. For each operator, we include a rule containing only the information necessary to allow that operator to be instantiated. For instance, in subtraction one must know which digits one is subtracting before one can find their difference. Such an overly general initial procedure will have a very useful feature: when combined with breadth-first search control, each of the component rules will apply in as many situations as possible. As a result, the procedure will generate all states contained in the problem space it is searching. Assuming our initial procedure includes those operators used by the student, an exhaustive search scheme is guaranteed to generate the same answer as the student for each problem. Of course, it will also generate many other answers, and our goal is to find a set of conditions that, when added to the initial rules, will generate only the student's answers and none of the others.

There is an old adage that the best way to learn is to make mistakes, and researchers in machine learning have found this to be true of artificial learning systems as well as human ones. In particular, if one is attempting to learn the conditions under which a rule should apply, it is useful to have *negative* instances of that rule as well as positive ones. Nearly all AI learning systems rely on a strong distinction between good and bad instances, and if we hope to apply condition-finding techniques from machine learning to the student modeling task, we must find some way to determine good and bad instances of the initial procedure's rules.

Fortunately, this is exactly what an exhaustive search through the problem space provides for us. Since each step along the solution path – the path leading to the student's answer – brings the system closer to the goal, these steps must be good instances of the rule that proposed them. On the other hand, steps leading one step off the solution path take the system away from the goal, so they should be classified as bad instances of the responsible rule. Steps lying more than one step off the solution path should be ignored, since the system should never have been there in the first place. This approach to assigning credit and blame in procedure learning has been employed by Mitchell, Utgoff, and Banerji (1983), Anderson (1981), and a number of other researchers. Sleeman, Langley, and Mitchell (1982) have discussed the capabilities of the "learning from solution paths" method.

3.4. Alternate Condition-Finding Methods

Given a set of positive and negative instances for each rule in the original procedure, we can draw upon the entire repertoire of condition-finding methods developed by machine learning researchers. These methods vary along a number of dimensions, and tradeoffs exist between different approaches in terms of their requirements and their capabilities. For instance, much of the work on condition-finding involves

incremental methods (Winston, 1970, Hayes-Roth, 1976, Mitchell, 1977), which accept one positive or negative instance at a time, and modify the current hypothesis accordingly. However, a few researchers have explored methods that accept an entire set of instances at the outset (Hunt, 1966, Michalski, 1980, Quinlan, 1983a). As one might expect, these methods are much more robust with respect to noise than are the incremental methods, since no single instance will have much effect on the overall decision process. Incremental learning methods provide much more plausible accounts of the human learning process, but we are not restricted to human learning techniques in finding ways to automate the construction of student models. Since our overly general procedure generates an entire set of positive and negative instances (for each operator) for each problem, and since student behavior is liable to be noisy, an all-at-once learning scheme is the more attractive alternative.

Another dimension of variation involves the *direction* in which learning systems search the space of conditions. Many condition-finding systems (Hayes-Roth, 1976, Vere, 1975) begin with very specific rules, and make these more general as they search for some set of conditions which predict all of the positive instances but none of the negative. These are called *generalization-based* learning systems. Others begin with very general rules and formulate more specific hypotheses as they progress toward the goal (Brazdil, 1978, Quinlan, 1983a, Langley, 1983a). These are called *discrimination-based* learning systems. Since the desired rule usually lies somewhere between the most general and the most specific hypotheses, it is even possible to carry out a bi-directional search through the rule space (Mitchell, 1977).

One might think, since our initial procedure is composed of overly general rules, that a discrimination-based approach is required. It is true that the final rules will be more specific than the original ones, but this does not say anything about the way in which the additional conditions must be found. Given a set of positive and negative instances, one could determine these conditions through discrimination, through generalization, or through bi-directional search. In fact, we will employ a discrimination learning scheme to find these conditions, but one can imagine using other methods.

We have chosen a discrimination-based strategy for a number of reasons. First, since discrimination learning methods consider simpler rules before more complex ones, they should construct as simple a student model as possible. Second, since such methods do not attempt to find features held in common by all positive instances, they are capable of learning *disjunctive* rules, which students may also employ. Finally, a number of all-at-once discrimination methods have been developed, and we felt it would be relatively straightforward to adapt one of these to the student modeling task. Moreover, these methods had shown themselves capable of dealing with noisy data (Quinlan, 1983b), and we felt this was an important feature to include in a student modeling system. Now that we have described the alternate learning methods and our reasons for choosing discrimination, let us turn to the particular condition-finding method we have implemented.

3.5. Constructing Discrimination Networks

One of the earliest cognitive simulations was Feigenbaum's EPAM model of human verbal learning behavior (Feigenbaum, 1963). EPAM represented associations between nonsense syllables in terms of a discrimination network. Such a network is stored as a tree, with events entering at the root node and being sorted down different branches until they eventually reach a terminal node. Stored at each terminal node is a response, so that when an event reaches that node, the response is evoked. Hunt, Marin, and Stone (1966) applied this approach to the task of learning concepts from examples. Unlike EPAM, their CLS system accepted an entire set of positive and negative instances at the outset, so that it could use simple statistical methods to determine the best discrimination network to account for the data. Hunt, Marin, and Stone's system could deal only with attribute-value based concepts, such as *large and not square*, or *small or blue*. However, Quinlan (1983) has extended their technique to deal with complex relational representations, such as occur in chess.

Figure 1 presents a sample discrimination network representing the concept **((blue and not large) or (not blue, not circle, and large))**. Each terminal node has either a + or − associated with it. If an event is sorted to a + node, then it is classified as a positive instance of the concept; however, if it is sorted to a − node, it is classified as a negative instance. The sorting process is quite simple. An event enters at the root node, and is sent down either the left or right branch, depending on which of the tests it satisfies. If sent down the left branch, the tests leading from that branch are applied to the event, and again it is sent down one branch or the other. This continues until a terminal node is reached. Note that conjunctive conditions (e.g., blue and not large) are represented as sequences of tests, while disjunctive conditions are represented by alternate branches. Also note that discrimination networks can easily represent disjuncts at any level in the tree structure, so conceptually complex rules can be stored in a very efficient manner.

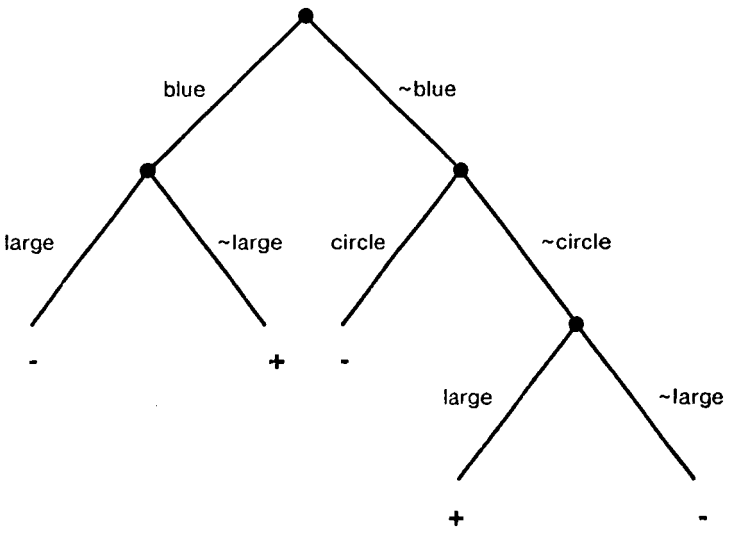

Figure 1. A sample discrimination network.

The process of generating a discrimination network can be viewed as a best-first search through a rule space. One begins with a network consisting of only the root node. All possible tests are applied to the observed positive and negative instances, and the most discriminating of these tests is used to create the first branching point. This test and its negation[3] are placed on the resulting branches, and the various instances are sorted down the branch whose test they satisfy. This process is repeated on the resulting subsets, creating branches lower in the tree, until nodes are reached that contain only positive or only negative instances. This method can be easily modified to deal with noise, simply by allowing it to halt before such pure nodes are achieved.

One constraint on this approach to learning is that one must have some set of tests available at the outset from which to construct a discrimination network. In the case of attribute-value representations, this is trivial to accomplish. Given a list of potentially relevant attributes and their values, it is a simple matter to generate all possible tests that could be placed on the branches of the tree. However, in domains involving relational representations, such as arithmetic or algebra, one must be able to construct more complex relational tests. In order to do this efficiently for the student modeling task, we have placed a constraint on the types of tests that

[3]Some methods for constructing discrimination networks do not assume binary trees; however, for our purposes, this simplifying assumption works quite well.

can be included: the new conditions placed on proposers can only refer to objects that are mentioned in previously existing conditions. We will see the details of the resulting method in the following section, as it is applied to the domain of subtraction. At the present time, it is unclear whether this constraint represents a basic limitation of our approach, a practical measure that can be generalized later, or an important psychological principle.

For the student modeling task, our system must construct a separate discrimination network for each operator, based on the positive and negative instances generated during the exhaustive search process. These networks can be easily transformed into disjunctive sets of conditions by eliminating those branches that lead to nodes containing only negative instances, and taking the conjunction of the remaining tests occurring in sequence. Thus, for the tree in Figure 1, we would obtain two disjunctive sets of conditions, *blue and not large*, and *not blue, not circle, and large*. Of course, features such as color and shape are not very useful in accounting for behavior in mathematics domains, but one can imagine more relevant features that would lead to similar discrimination trees, and thus to similar conditions. These conditions can then be added to the proposers in the initial procedure, leading to a production system model of the student's behavior on a set of test problems.

3.6. The ACM Student Modeling System

To summarize, we represent a strategy as a collection of production rules encoding a search scheme in some problem space. The initial procedure consists of overly general rules that contain only those conditions necessary to instantiate an operator. This procedure is then run on a set of sample problems, generating the observed answers along with many others. Steps along the solution path are labeled as good instances of the responsible rules, while steps leading one step off this path are labeled as bad instances. The set of instances for each operator are then passed to a discrimination learning method, which generates a discrimination network for classifying steps into desirable and undesirable ones. These discrimination nets are transformed into a set of disjunctive rules for each operator. Taken together, these rules constitute a model of the student's behavior.

We have implemented ACM, a student modeling system that incorporates these design decisions. Given a set of operators and a set of test problems, ACM searches the resulting problem spaces until it finds those answers produced by the student. Based on the resulting solution paths, the system classifies steps as positive or negative instances of the initial rules, and constructs a discrimination network for each operator. ACM is an acronym standing for Automated Cognitive Modeler, and the system is a distant descendant of SAGE (Langley, 1983b, Langley, 1983c), an AI system concerned with learning search heuristics.

ACM can be divided into two components, one domain-specific and another quite general. The first component consists of the proposers for searching the problem space, the operators used to generate new states, and the abstract conditions provided by the user. When ACM is working in the subtraction domain, this component will be quite different than when the system is working in the algebra domain. The second, more general component consists of rules for assigning credit and blame, along with the discrimination mechanism for generating variant rules. These will remain the same regardless of the domain in which ACM is operating.

The ACM system is implemented in PRISM2, a production system formalism designed for modeling learning processes. The domain-specific proposers, as well as the general rules for assigning credit and blame, are stated as PRISM2 productions; the discrimination mechanism is written in Franz Lisp, and is called from the production system. In implementing a production system model, one generally decides on some *conflict resolution* principle, which determines the manner in which rules are selected for application. One common conflict resolution principle prefers rules matching against the most recent elements in memory. Another

prefers the most specific rules that match on a given cycle. In ACM, we decided to avoid the conflict resolution process entirely, so that *any* matched rule would apply. There were two reasons for this decision. First, we wanted the system to carry out a breadth-first search through the problem space defined by its operators, and this was a natural way to implement such a search control scheme. Second, we wanted the final student model to contain all control information in the component rules themselves. In this way, we ensured that any user of the system (such as a teacher) would be able to understand the final model by inspecting the rules themselves, and need not understand such an esoteric process as conflict resolution.

Table 1. The initial production system for subtraction.

---

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

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*,
then add ten to *number1*.

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*,
then decrement *number3* **by one.**

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

The potential for competing hypotheses arises at a number of points in ACM's search through the procedure space. First, one must specify some problem space within which behavior is to be modeled. However, it is possible that more than one problem space can be used to explain a student's behavior. Currently, the programmer defines the problem space, but we will have more to say on this later. Second, one must find a solution path which generates the same answer as the student on each problem. However, more than one path may lead to the same solution. The current system selects the shortest of these paths, and if two or more equally short paths are found, one is selected at random. This is not a very satisfying solution, though it has led to few difficulties in our runs in the subtraction domain. More thought should be given to this issue. Finally, one may find two or more tests to be equally discriminating during the condition-finding process. In some cases, these may be complementary, but in other cases they may be competing explanations of the student's behavior. We will return to this issue in the following section, in the context of a particular example.

4. Modeling Subtraction Strategies

Now that we have discussed ACM's learning methods in the abstract, let us examine how they can be applied to a particular domain – multi-column subtraction problems. We decided on subtraction as the initial testbed for our system precisely because Brown and Burton's, Brown and VanLehn's, and Young and O'Shea's extensive work in this area provided us with a relatively well-understood set of bugs. Ultimately, we expect our approach to be applicable in novel domains, but in the testing stage it is essential to have a familiar bug library, in order to see what percentage of the observed bugs our approach can handle. Thus, we will rely heavily on the earlier empirical analyses of these researchers in the subtraction domain, though we employ a quite different approach to the student modeling problem.

Below we describe ACM's behavior in the domain of multi-column subtraction problems. In order to do this, we must first describe the problem space searched by the system, in terms of the initial states, the operators used for moving through this space, and the initial proposers for these operators. After this, we discuss the conditions that ACM considers in attempting to characterize the student's strategy. Next, we follow the steps taken by the system in modeling correct behavior on a set of test problems. This should help the reader understand the relation between student modeling and the AI problem of procedure learning. We then trace the system's behavior in modeling a buggy subtraction strategy. Finally, we summarize those bugs that the system has successfully modeled, and consider the reasons for its successes and failures.

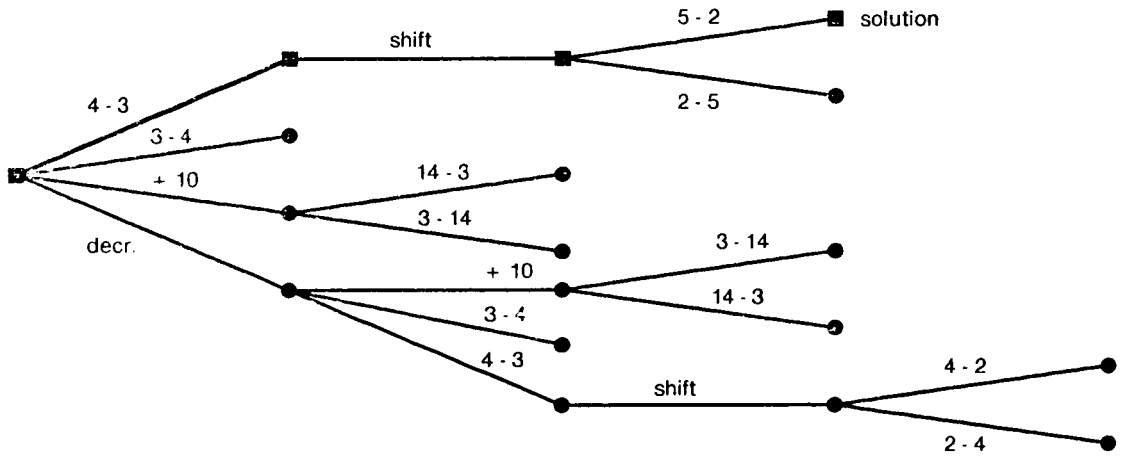

Figure 2. Search tree for the problem 54 – 23 = 31.

4.1. Subtraction as a Search Problem

As we have seen, one of the domain-specific inputs required by ACM is a list of operators for moving through a problem space. In the domain of subtraction, a number of operators are involved. 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 are given in Table 1. In fact, these operators are not even capable of *correctly* solving all subtraction problems (additional operators are required for borrowing across 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.

In the context of the student modeling task, ACM is not given the "legal" conditions on operators, since this would cause it to begin with the correct algorithm for a domain like subtraction. Instead, the system is

presented with the minimum conditions necessary to allow each operator to apply. For instance, the correct rule for finding the difference between N1 and N2 would require that N1 be above N2, and that N1 be larger than or equal to N2. However, including these conditions would mean that ACM could never construct student models in which they were absent, and this would *severely* limit its generality. Rather, the program is told only that two numbers must be in the same column before their difference can be found, as can be seen from the first rule in Table 1. Similar simplifications are made for the add-ten and decrement rule, since a number of common bugs also involve incorrect conditions on these operators. Note that the shift-column rule is correctly stated, since few bugs seem to involve this operator.

Given the very general initial conditions on its proposers, ACM must search when it is given a set of subtraction problems. However, rather than looking for the *correct* answer to these problems, it searches for a set of answers that agree with the student's answers. Figure 2 shows the system's search on the non-borrowing problem 54 − 23, when the correct answer 31 is given by the student that ACM is attempting to model. For this problem-answer pair, the solution involves finding a difference, shifting columns, and finding a second difference. Figure 3 shows ACM's search on the borrowing problem 93 − 25, when the correct answer 68 is given. In this case, the solution path includes decrementing and adding ten, as well as finding a difference, shifting columns, and finding a final difference. States along the solution paths are shown as squares, while other states are represented by circles. Note that a number of dead-end states occur when ACM finds a difference for some column that disagrees with the student's answer for the same column. Given such a partial answer, ACM decides that the current path should not be pursued any further, drastically reducing the amount of search it must carry out.

4.2. Generating Plausible Conditions

Before it can determine the conditions on its operators that will let it model a student's behavior, ACM must be told what form these conditions can take. This information is provided by the programmer in terms of an abstract set of tests that can be instantiated for each rule. ACM generates the set of potentially relevant conditions at the outset of each run.[4] This actually occurs before the system begins to search for solution paths to explain a student's behavior; we have delayed discussion until now because one must understand the nature of the initial proposers in order to understand how specific conditions are generated. Given an abstract condition such as **(greater number number)**, ACM examines each of its rules, and finds those variables having the same type as those in the abstract condition. For example, the find-difference rule has two numeric variables, **number1** and **number2**. These would be inserted into the general form in all possible ways, generating a set of potentially relevant conditions. In this case, there are two such conditions for the find-difference rule, **(greater number1 number2)** and **(greater number2 number1)**. Analogous tests would result from other abstract conditions, with a different set of specific conditions being constructed for each of the initial proposers.

Some caution is necessary in deciding on the set of abstract conditions given to the system, since we want the final model to be psychologically plausible. Thus, conditions which examine the problem state generated by the system ten steps earlier would be unacceptable. However, most tests involving the current problem state are plausible, since even if the student forgets something, he can generally retrieve it by inspecting what he has written down on the paper. For example, since a number is crossed out when it has been decremented, this is a plausible test to give to the system. **Above** relations are another plausible test, since these are available by direct inspection as well. Note that the constraint that the arguments of each predicate be already mentioned in the initial rule considerably limits the types of tests that ACM can consider. From an AI viewpoint, this would be a severe drawback, but from a psychological perspective, this is an asset.

---

[4] This need only be done once for each student, and involves very little computational time.

In the runs described below, ACM was presented with ten abstract condition types, from which the system generated specific potential conditions for each of its initial rules. These condition types were:

- **(Greater number number)**. This condition is satisfied if the first argument is greater than the second argument. Both arguments must be numbers.
- **(Above row row)**. This condition matches if the first argument is above the second. Both arguments must be rows.
- **(Added-ten number)**. This test is met if ten has been added to the predicate's argument, which must be a number.
- **(Decremented number)**. This condition is satisfied if the argument has been decremented. Again the argument must be a number.
- **(Added-ten-to-any)**. This condition is satisfied if ten has been added to any number. No arguments are involved.
- **(Decremented-any)**. This condition is satisfied if any number has been decremented. No arguments are necessary for this test either.
- **(Just-decremented number)**. This test is satisfied if the argument was just decremented on the previous step. This argument must be a number.
- **(Is-zero number)**. This matches if the numeric argument is zero.
- **(Is-one number)**. This matches if the numeric argument is one.
- **(Is-ten number)**. This matches if the numeric argument is ten.

Note that some of these conditions, such as **(greater number number)** and **(above row row)**, relate to features of the problem state, while others, such as **(decremented column)**, describe traces left by certain operators.

Let us consider ACM's response to these conditions in the context of the initial find-difference rule. In this case, the system generates the following specific conditions: **(greater number1 number2)**, **(greater number2 number1)**, **(above row1 row2)**, **(above row2 row1)**, **(added-ten column1)**, **(decremented column1)**, **(added-ten-to-any-column)**, **(decremented-any-column)**, **(just-decremented column1)**, **(is-zero number1)**, **(is-zero number2)**, **(is-one number1)**, **(is-one number2)**, **(is-ten number1)**, and **(is-ten number2)**. However, since ACM has the ability to consider negated conditions as well as positive ones, some of these are redundant. Accordingly, the system is told that **(greater x y)** is the inverse of **(greater y x)**, and that the same relation holds between **(above x y)** and **(above y x)**. As a result, the system eliminates the two specific conditions **(greater number2 number1)** and **(above row2 row1)**, and only considers their inverses.

By providing ACM with a set of abstract conditions, along with a set of initial rules, we are defining the space of rules that the system will search in constructing student models. The conditions given to ACM certainly constrain its search, and different sets of conditions may lead the system to formulate entirely different models. However, we will argue again that this is more elegant and more general than providing the system with a library of standard bugs. In specifying abstract conditions, we only partially determine the course that events will take. The particular conditions ACM selects will depend on the data it encounters during its modeling attempts. Thus, though there is a model-driven component to the system, its final actions are determined by the behavior of the student it is attempting to explain.

4.3. Modeling the Correct Subtraction Strategy

After finding the solution paths for a set of problems, ACM uses the conditions it has generated to formulate more conservative proposers that will let it regenerate those paths without search. Let us examine the search trees in Figures 2 and 3, and the variant rules that are generated from the good and bad instances in these trees. Since most of the interesting learning occurs with respect to the find-difference operator in these

two problems, we shall focus on it here. Upon examining the two search trees, we find four good instances of finding a difference, which lie on the solution path, and eight bad instances, which lie one step off the solution path. A number of additional instances lie multiple steps off the solution path, but these do not concern us, since the system should never have reached those states in the first place.

Table 2 lists the four good and eight bad instances of the find-difference rule from the problems in Figures 2 and 3, along with the tests satisfied by each of these instances. Instances occurring beneath a + are positive, while instances marked with − are negative.[5] Crosses mark tests that are satisfied by a particular instance, while the absence of a cross indicates an unsatisfied test. Consider the two instances 4 − 3 and 3 − 4 at the first level of Figure 2. The first of these steps lies along the solution path, while the second leads off this path. The first instance satisfies the test **(greater number1 number2)**, since 4 is greater than 3; however, the second fails to satisfy the test, since the converse is not true. ACM's goal is to find some set of conditions that will cover all of the positive instances, but none of the negative instances. Upon close examination, we find that no single condition is capable of this, but that a few come close. Considering pairs of tests, we do find one pair that satisfies our criterion − **(above row1 row2)** and **(greater number1 number2)**. In this simple example, one can find the desired conditions by inspection, but for complex cases, a more sophisticated approach is required.

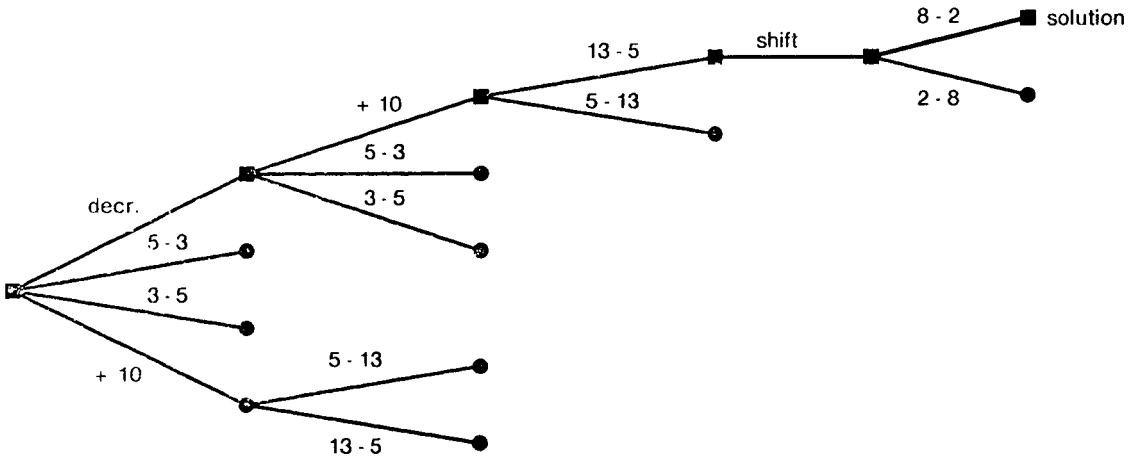

Figure 3. Search tree for the problem 93 − 25 = 68.

As we outlined earlier, ACM attempts to generate a discrimination network that can distinguish the positive instances of a rule from its negative instances. The system begins with a branchless tree, and grows the tree downward, adding the most discriminating tests first. This process continues until nodes are reached containing only positive or only negative instances, or until no useful test can be found. At a higher level, ACM can be viewed as carrying out a search through the space of possible discrimination networks, attempting to find that tree which can account for the data in as simple a manner as possible. Every time the system decides which test to use in distinguishing between positive and negative instances, it can be viewed as taking a step through this space of discrimination networks.

Thus, the process of generating a discrimination network can be viewed as a best-first search through a rule space, with each successive state adding a new set of branches to the network. In order to direct this search, one needs some evaluation function. Hunt, Marin, and Stone's CLS system selected that test matching

[5] The two instances 5 − 3' and 3 − 5' differ from the instances 5 − 3 and 3 − 5 in steps taken before the find-difference rule was applied.

the greatest number of positive instances, while Quinlan's ID3 system computed a complex information theoretic measure. The current system 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 = maximum (S, 2 − S). An optimal test that matches all positive instances and none of the negative instances would thus receive the maximum score S = 1 + 1 = 2. In contrast, a test with no discriminating power would match exactly have the positives and half the negatives, giving the minimum score S = 1/2 + 1/2 = 1. Since negated tests can be useful, we also want tests that match all of the negatives but none of the positives to score highly, and for this reason the final evaluation function E is defined as the maximum of S and 2 − S. To summarize, at each stage in constructing a discrimination network, ACM computes the function E for each test, and extends the tree based on the highest scoring test. When a test with the perfect score of 2 is found, no further discriminations need to be made.

Table 2. Tests and instances for the find-difference operator.[6]

	+	+	+	+	−	−	−	−	−	−	−	−	
	13-5	8-2	4-3	5-2	5-3	3-5	3-4	2-5	5-3'	3-5'	5-13	2-8	E
GREATER N1 N2	X	X	X	X	X				X				1.75
ABOVE R1 R2	X	X	X	X		X				X			1.75
ADD-TEN C1	X												1.25
DECR. C1		X											1.25
ADD-TEN-ANY	X	X									X	X	1.25
DECR-ANY	X	X							X	X	X	X	1.0
JUST-DECR. C1									X	X			1.25

For example, in Table 2 we see that the test **(greater number1 number2)** satisfies all four of the positive instances, but fails to satisfy only six out of eight negative instances. This leads to the score E = 4/4 + 6/8 = 1.0 + 0.75 = 1.75. The test **(above row1 row2)** receives an identical score, though it satisfies a different pair of negative instances. The test **(added-ten column1)** is assigned a lower score; since it matches only one positive instance and none of the negative instances, we get E = 1/4 + 8/8 = 0.25 + 1.0 = 1.25. Thus, one can see that the evaluation function E does reasonably well at finding features that distinguish positive instances from negative instances, leading to simple discrimination networks for summarizing the data.

[6]The six tests based on the is-zero, is-one, and is-ten predicates are not included in the table, but all have a minimal score of 1.0 for the evaluation function E.

Of course, it is still possible for two or more tests to tie on this dimension, and in such cases, the order in which the abstract conditions were originally specified by the programmer is used to break the tie.[7] Since the **greater** condition was listed before the **above** condition in our earlier discussion, the test **(greater number1 number2)** would be preferred. This facility gives the user the ability to incorporate into the student modeling process his own biases about the types of conditions the student is likely to use. On the other hand, if the user has no such preference, he is forced to specify one anyway, and this is undesirable. Later we will propose an alternative to this approach, but for now we are concerned with honestly describing the current system.

Once it has selected a test, ACM adds a branch to its discrimination network, and sends the various instances down the left or right branch depending on whether they pass or fail the test, respectively. If all of the instances sent down a branch are negative, the branch is abandoned, since the system is only interested in generating positive instances of each operator. If only positive instances are sent, then that branch becomes a terminal node in the discrimination network, and a rule containing the tests leading to that node is formulated. If both positive and negative instances are sent, the discrimination process is called recursively to find additional tests which will further subdivide the instances.

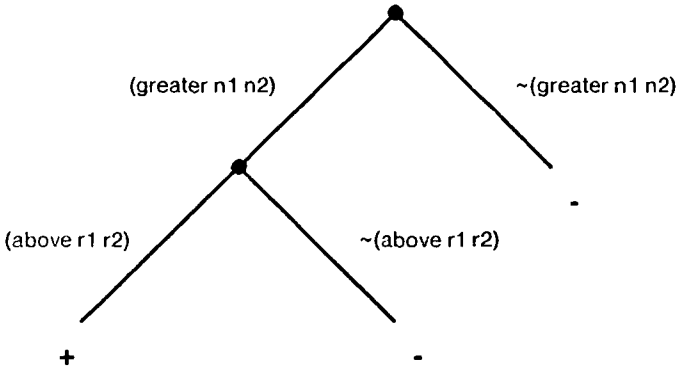

Figure 4. Correct discrimination network for the find-difference operator.

In our example, the **(greater number1 number2)** test was selected in favor of the **(above row1 row2)** condition, due to the order in which the abstract tests were specified by the user. This leads to a right branch containing only negative instances, and a left branch containing four positive instances and two negative instances. When ACM examines the remaining tests, it finds that the **(above row1 row2)** condition achieves the maximal score of 2, faring better than any other test. As a result, two new branches are created, the left one containing only the four positive instances and the right containing both negative instances. Based on the left branch, ACM creates a single rule containing the two conditions **(greater number1 number2)** and **(above row1 row2)**, in addition to the initial conditions. Figure 4 presents the final discrimination network for the find-difference operator. This includes both positive and negative terminal nodes, even though only the former are used in constructing the actual rules. Similar networks are created for each of the other operators. Table 3 presents the rule formulations of these discrimination networks, which can be used by ACM to resolve each of the original subtraction problems correctly without search.

---

[7] In cases where two or more specific conditions are based on the same abstract condition, even this priority ordering is not sufficient. However, this has not yet given us difficulties in our subtraction runs.

Table 3. A 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*.

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*.

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 1.

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

4.4. Modeling a Buggy Strategy

Now that we have considered ACM's learning 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, shown in Table 1. If we present the same two subtraction problems we used in the previous example, we find that the buggy student produces the correct answer 54 − 23 = 31 for the non-borrowing problem, but generates the incorrect answer 93 − 25 = 72 for the borrowing problem. As before, ACM's task is to discover a set of variants on the original proposers that will predict these answers. Obviously, the solution path for the first problem will be identical to that shown in Figure 2. However, the solution path for the second problem (shown in Figure 5) differs considerably from that for the same problem when done correctly (shown in Figure 3).

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 5 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 discovered to assign credit and blame. As in the previous run, only positive instances of the shift-column operator are found, indicating that its conditions need not be altered. And since both positive and negative instances of the find-difference rule are noted, ACM calls on its discrimination process to determine additional conditions for when to apply this operator. The major difference from the earlier run is that only *negative* instances of the add-ten and decrement operators are found. This tells ACM that the proposers for these operators should not be included in the final model, since apparently the student nevers uses them.

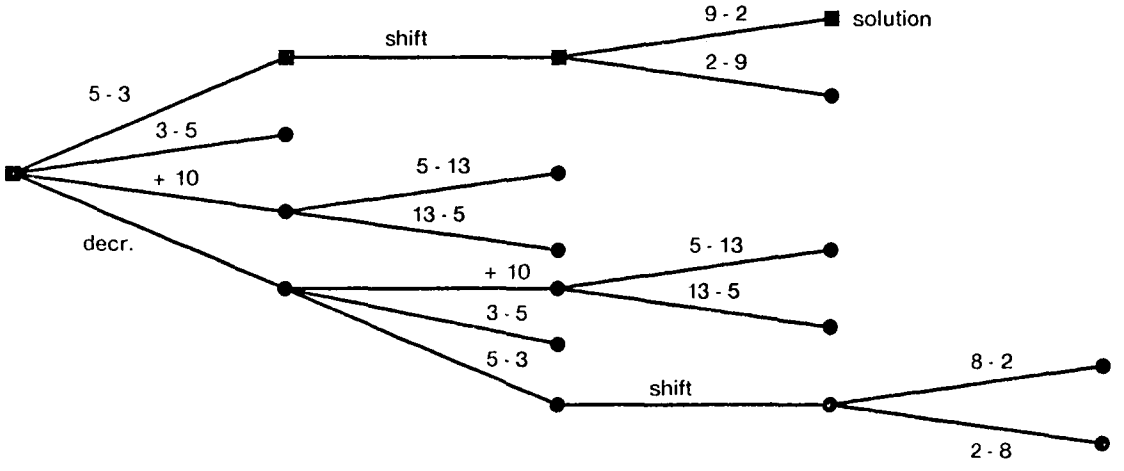

Figure 5. Search tree for the problem 93 − 25 = 72.

For this idealized student, ACM finds four positive instances of the find-difference operator and four negative instances. When these instances are passed to the discrimination mechanism, the system finds the **(greater number1 number2)** condition to be the most discriminating test, with a maximal score of 2.0. However, the **(above row1 row2)** condition, which tied with the **greater** test in modeling the correct strategy, does not fare so well on this data set, receiving the score E = 0.75 + 0.75 = 1.5. Moreover, the **greater** test completely discriminates between the positive and negative instances, leading ACM to a very simple discrimination net for the find-difference rule. This is because the idealized student is 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 4 presents the variant rules that ACM generates 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.

In our examples, ACM was presented with an idealized student's behavior on only two test problems. Although these were sufficient for modeling the correct strategy and the "smaller from larger" bug, more complex errors will clearly require additional test problems. VanLehn (1982) has reported a diagnostic set of 20 test problems, which let the DEBUGGY system distinguish between every bug in its library, even in the presence of noise. Since ACM searches a larger space of student models than DEBUGGY, it is possible that our system will sometimes require an even larger set of test problems in order to completely disambiguate a student's behavior. One alternative to designing a standard set of test problems would be to let ACM generate its own diagnostically useful problems when necessary; such an extended system could model some students based on a only a few problems, and produce more if the need arose. We discuss one approach to automatically generating diagnostic problems in a later section.

Table 4. A 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*.

4.5. Initial Results in the Subtraction Domain

ACM has been implemented on a Vax 750, and successfully run on a number of the more common subtraction bugs. Although the system must still be considered in the testing stage, let us review some tentative results we have obtained from these runs. Table 5 presents eleven common bugs reported by VanLehn (1982), along with their observed frequencies. ACM has successfully modeled each of these bugs, given idealized behavior on a set of representative test problems. A number of these bugs involve borrowing across 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 near-exhaustive search.

We have focused on the most common bugs first, since if ACM had serious difficulty with these errors, we would not need to bother with less common mistakes. Fortunately, the system ran into only minor difficulties in modeling these bugs, which were easily corrected. Although we have so far tested ACM on only eleven bugs, we estimate that the current version will be able to model another 20 known bugs without modification. Moreover, we estimate that ACM will be able to model approximately 30 additional bugs, provided the system is augmented with a few more operators (such as *incrementing* rather than decrementing) and a few more abstract conditions (such as **(left-most column)**). We plan to test these predictions in the near future. We feel these are conservative predictions, and that ultimately ACM will be able to account (with only minor modifications) for 80 percent of the systematic subtraction errors that have been observed. Since a few of the known bugs appear to lie outside the subtraction problem space searched by ACM, we would not expect the system to handle these without major changes in its representation of states and operators.

We should also mention the computational resources required by the student modeling system. Let us consider ACM's behavior in modeling the correct subtraction strategy, based on the two examples discussed earlier. On this run, the system required 39 CPU seconds to generate a set of specific conditions for each operator, and 52 CPU seconds to exhaustively search the two problem spaces and reproduce the student's answers. In addition, ACM took 18 CPU seconds to retrieve the solution paths and to assign credit and blame, and 17 CPU seconds to use the resulting positive and negative instances in constructing discrimination networks. Taken together, these sum to a total of 126 CPU seconds, or slightly more than two CPU minutes. This is not very long, but recall that ACM was given only two simple test problems in this run.

Table 5. Subtraction bugs successfully modeled by ACM.

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

Since the exhaustive search and credit assignment phases should each increase linearly with the number of problems, we can predict that ten similar problems will require 760 CPU seconds, or nearly 13 CPU minutes. Moreover, the search spaces in our example were relatively small, since the short solution paths kept the combinatorics to a minimum. However, the solution paths for problems with additional columns, especially those involving borrowing across zero, are much longer, and can considerably slow down the system's progress. The condition generation stage should be unaffected by the number of problems, and the discrimination stage should be only slightly affected, so efforts to increase efficiency should focus on the search and credit assignment phases.

5. Extending the Approach

In an earlier section, we mentioned three ways in which competing explanations of a student's behavior could arise, and it is now time to address these possibilities in greater detail. The first of these involved the occurrence of alternate problem spaces, each capable of generating the student's answers. Since ACM currently works in a single problem space, we discuss below a more comprehensive approach to student modeling that deals with this possibility. Two other possibilities for competing explanations involve the discovery of multiple solution paths, and the occurrence of conditions that score equally well on both of ACM's evaluation functions. We discuss these complications together, along with ways to deal with them in future implementations.

5.1. Considering Alternate Problem Spaces

The perspective which underlies the ACM program allows a clear separation of the system's psychological assumptions and its computational devices. The restrictions which define the procedure space constitute a set of assumptions about the mental representation of skills, but the computational techniques which are used to search that space have no psychological content. Each node in the procedure space is a collection of productions which work in the same basic problem space.

Until now, we have avoided the issue of how one determines in which problem space the student is operating. In obtaining the computational results reported in the previous section, we chose a particular, rather simple arithmetic space. We advanced no psychological reasons for selecting that space. However, *the problem space in which the student model works is no less a psychological hypothesis than is the collection of productions itself.* The specification of a problem space constrains the notation for encoding a given problem, the operators used in solving it, and the criterion for having reached a solution. These aspects remained constant throughout the runs discussed earlier. Let us pause and pursue a psychological interpretation of these formal constructs.

The notation used to encode a problem presupposes a set of hypotheses about what aspects of the problem the pupil pays attention to. A notation employs some list of predicates which can be used to describe the state of the problem. For example, in the subtraction problem space we have been using, predicates like in-column, in-row, above, left-of, added-ten, decremented, and greater-than came into play. Each predicate introduced into the notation constitutes a hypothesis that (a) the pupil has that concept available, and (b) he is actively using it when thinking about arithmetic problems.

In the examples discussed in the previous section, ACM employed four basic arithmetic operators in searching the subtraction space:

- **(Find-difference digit-1 digit-2).** Takes two digits as input, determines the difference between them, and writes that result in the appropriate column.
- **(Decrement digit).** Takes a digit, subtracts unity from it, scratches out the digit, and writes the new result in its place.
- **(Add-ten digit).** Takes a digit, adds ten to it, scratches out the digit, and writes the new result in its place.
- **(Shift-column column-1 column-2).** Shifts attention from the first column to the second column.

As one can see, most of these operators are rather complex, consisting of a sequence of steps. What does it mean to use these as "primitive" operators in a student model?

Clearly, different operators can be defined for one and the same task. In the above example, the **decrement** operator can be broken down into a number of "smaller" components: **cross-out**, **subtract-unity**, and **write-result**. In other words, operators can be defined at different levels of *aggregation.* Some operators are powerful, contain a lot of machinery, and perform a large portion of the task; others are simple, and make a very modest contribution to the complete computation. Whatever machinery is packed away in the operators will not be explicitly represented in the final model; only the rules for directing search through the problem space will be explicitly represented. To select a problem space is therefore to select a *level of analysis.*

In student modeling, the focus of interest is on the bugs, the conceptual mistakes that cause the pupil to produce the wrong answers on test problems. Therefore, these decisions and mistakes should be represented in the heuristic rules for when to apply operators, rather than in the operators themselves. This is the basic representational principle underlying the ACM system. In other words, an operator represents a hypothesis about a subskill which the pupil is able to execute without making errors. The set of operators for the arithmetic problem space is an assumption about what the pupil knows about arithmetic; they are the parts of arithmetic that he has mastered. The rules about when and how to apply those operators, on the other hand, represents those parts of the skill in which he might be making mistakes.

Since individual differences in cognitive processes are ubiquitous in general, and since pupils are at different skill levels in particular, it is clear that a robust student modeler cannot consider just a single problem space, as does ACM. Not every pupil (every bug) can be modeled in a single space, unless that space

incorporates *very* low-level operators. As an illustration, let us consider a very high-level problem space for subtraction that includes only two operators:

- **(Process column).** Takes a column as input, computes the column difference (if and only if the bottom number is not greater than the top number), writes the answer in the appropriate place, and shifts attention to the column immediately to the left of the column just processed.
- **(Borrow column-1 column-2).** Takes a pair of columns as input, subtracts one from the top-number in column-1 (if and only if it is greater than zero), scratches out the old digit, writes the new result in its place, shifts attention to column-2, adds ten to its top number, scratches out the previous digit in that place, and writes the new number instead.

In this space, we represent the child as knowing how to carry out single column subtraction, and how to borrow. Could there be anything left to know about subtraction? Which bugs could possibly be modeled in this highly aggregated space, at this level of analysis? At least one error *can* be modeled in this space. It is the "always borrow" bug (VanLehn, 1982), in which the child borrows in every column, whether it is needed or not.

We would not expect too many other bugs to be representable at this level of analysis. Assuming that the pupil knows how to process a column and how to borrow correctly does not leave much room for subtraction errors to occur. As we "pick apart" these subskills into their components, we free up the possibilities for interaction between those components, thus producing new ways in which they can be incorrectly applied. In other words, the more fine-grained the level of analysis, the more bugs one can represent. We will use the term *expansion* to refer to the process of taking an operator and decomposing it into its components, representing it as a set of rules which call other, lower-level, operators. The result seems important enought to be designated as a principle:

- *The Bug Distribution Principle.* If S is a problem space in which the set of bugs B can be modeled, and space S' is generated from S by expansion of one of the operators of S, then the set of bugs B' which can be modeled in S' contains B as a subset.

At first glance, the Bug Distribution Principle might be thought to imply that student modeling should proceed in the most fine-grained problem space possible, since only such a space is guaranteed to contain all observed bugs. In fact, one could proceed in this manner, though searching such a space would be quite expensive. However, this conclusion ignores the psychological interpretation of the level of analysis: that the operators represent those parts of the skill that the pupil has mastered. Any student modeling system should produce viable psychological models; therefore, we should be prepared to model a pupil in a space which represents as accurately as possible what he knows. The resulting conclusion is that the student modeler should consider a number of different problem spaces in which the student might be working. In other words, the student modeler should *search* a space of problem spaces for the one in which a particular pupil best can be modeled.

Suppose that we have a space of arithmetic problem spaces. The relation of expansion (i.e., "less aggregated than") imposes a partial ordering on this space. At the top, there is the most aggregated space conceivable. This space has only a single operator, namely the operator which takes a subtraction problem as input, and solves it correctly. It represents those pupils who do not make any errors. Below that, we have first-level decompositions of the the subtraction skill (such as the two-operator space we used in the illustration above); for each of these, the decomposition proceeds further, until we reach some extreme level of fine-grained analysis, in which processes like the writing of a single digit are represented as operators.

We propose to search such a partially ordered space of problem spaces from the top down, until we find

the most highly aggregated space in which a particular pupil can be modeled. The justification of this approach is that it is maximally conservative in its assumptions about the gaps in the pupil's knowledge. Remember that the operators represent the subskills which the pupil has mastered correctly. The more aggregated the operators that we use in the model of the pupil, the bigger the "chunks" of the subtraction skill we are assuming he has mastered. The idea is that we should not postulate, in our accounts for buggy behavior, more things wrong with the pupil's thinking than necessary. This idea, too, seems important enough to be designated as a principle of student modeling:

- *The Principle of Conservatism in Student Modeling.* A pupil should be modeled as working in the highest-level problem space that contains the bug he is exhibiting.

The next question is, of course, how the space of problem spaces is to be generated. A system which invents different problem spaces for some task on the basis of an abstract task analysis is currently beyond the state of the art in AI. Some interesting initial work has been done on constructing new representations (Hayes, 1974), as well as on modifying existing representations (Korf, 1980, Lenat, 1982), but this research has not yet reached the applications stage. Our temporary response is therefore to hand-craft a set problem spaces, drawing upon whatever psychological knowledge is available. This work is currently under way.

One might object that providing the modeling system with a set of problem spaces is no better than providing it with a bug library. The main answer to this objection is that the problem spaces impose a *structure* on the set of bugs. For each arithmetic problem space, there is a set of bugs which can be represented in that space. The decision to model a pupil in a particular space carries with it the prediction that all the bugs he is exhibiting are included in the set belonging to that space; if they do not, the selection of that space was incorrect. Choosing a set of problem spaces, we also choose a particular clustering of the bugs. This leads us to a third principle:

- *The Bug Clustering Principle.* The space of problem spaces being considered by a student modeling program should be such that for any pupil, if N is the set of bugs exhibited by that pupil, there must exist a space S such that N is a subset of the bugs that can be expressed in S.

Expressed more simply, all the pupil's bugs must be representable within one and the same problem space. This gives us a criterion for selecting the space of problem spaces, and for empirically falsifying it. VanLehn (1982) has noted that bugs do not cooccur with equal frequencies; some types of errors are more likely to occur together than others. With the appropriate hierarchy of problem spaces, we should be able to *predict* which bugs should occur together. As far as we know, no other mechanism has yet been proposed to explain these data.

5.2. Generating Diagnostically Useful Problems

In the previous section, we described ACM's search for solution paths to account for a student's answer to a set of test problems. We noted the possibility for multiple paths to explain the observed answer on a given problem, and mentioned the system's current means for dealing with this possibility – selecting the shortest of these paths. However, this is not a very satisfactory solution. One might argue that the shortest path constitutes the most elegant explanation, and so should be preferred over others. However, since we are trying to explain student's *errors*, there is no special reason to suspect that these students are being as efficient as as possible. A truly robust student modeling system should have a more satisfactory method for dealing with such competing explanations.

One might consider all solution paths as complementary, rather than as competitors, and label all steps along any solution path as positive instances. However, if we really believe that the student is using an *algorithm*, then this approach cannot be used, since it may lead to a model that considers multiple paths

simultaneously. Another method would be to accept the ambiguity inherent in multiple solution paths, and to develop alternate student models based on different paths. In this framework, one would first assign credit and blame based on one of the paths, pass the resulting positive and negative instances to the discrimination component, and then do the same for the second path. Unfortunately, if we present ACM with ten test problems, and it discovers two solution paths for each of these problems, then $2^{10} = 1024$ alternate models will result. Such a combinatorial explosion is clearly unacceptable.

For the moment, let us consider the second case in which ACM must deal with competing explanations, and return to the problem of multiple solution paths later. Recall that the method for generating discrimination trees employs two evaluation functions to direct search through the space of possible rules. In an earlier example, we saw two tests, **(greater number1 number2)** and **(above row1 row2)**, which scored equally well on both of these evaluation functions. As a result, ACM was forced to fall back on a somewhat arbitrary priority ordering to select between them. In this case, no harm was done because the unselected test was chosen the second time around. In other words, the **greater** and **above** tests were complementary in this example.

However, there are situations in which such tests must be viewed as *competing* hypotheses rather than complementary ones. Such situations arise when ACM does not have enough data to determine that one test is better than another. For instance, if we had given ACM only correctly answered non-borrowing problems, it would have been unable to determine (as would anyone) whether the student had used the **greater** condition, the **above** condition, or both in deciding when to apply the find-difference operator. This results from the fact that either condition is sufficient for proposing the correct steps on non-borrowing problems. However, borrowing problems (such as 93 − 25) give the system additional information by providing new negative instances of the find-difference rule. In such cases, the obvious solution is to generate "critical experiments" in which one of the tests is satisfied but the other is not, and see which best predicts behavior.

Such a critical experiment cannot be carried out in isolation, but must be embedded within a particular problem. Such *diagnostically useful* problems could be presented to the student, and then to the system, leading to additional positive and negative instances which could be used to disambiguate the student model. Such a diagnostic problem might be generated in the following way. Suppose we have two tests that the discrimination method is unable to distinguish between, such as the **greater** and **above** conditions discussed earlier. Given the instantiations of the operator in question, we retrieve some positive instance for which both tests were satisfied. Upon examining the instance and the tests, we generate a new situation in which only one of the tests (say **greater**) is satisfied.

We then retrieve the problem in which the original instance occurred, and use the new situation to generate a slightly modified problem. If the original problem were 54 − 23, in which both conditions match, this method would produce a new problem like 93 − 25, in which only the **above** condition is matched. If the student solves this problem using the same operators (in the same sequence) as for the original problem, then the **above** test will acquire a new positive instance. On the other hand, if the student takes a different path, then a new negative instance will result. In either case, one of the tests will now outperform the other, so a more informed decision can be made. The current version of ACM is incapable of such subtleties, but this is an obvious direction for future research.

A related approach might be used to distinguish between competing solution paths. If one finds two paths that lead to the same student answer, then they must have a common state from which they diverge. This may be the initial problem state, or it may be far into the search tree. By examining the instantiations of the rules responsible for this divergence, one should be able to generate a variant problem in much the same way as for competing conditions. If only one of these solution paths has an analog in the variant problem,

then that path constitutes the valid explanation of the student's behavior, and the other is rejected. The details of this approach remain to be worked out, but the basic method appears promising.

6. Discussion

In the preceding pages we reviewed some of the previous work on student modeling, and described an alternate approach that employed techniques from the field of machine learning. We examined ACM, a student modeling system based on this analysis, and discussed its application to the domain of multi-column subtraction problems. We also noted some limitations of the current system, and suggested methods for overcoming these problems. Several issues present themselves for discussion, the most important being the generality of the techniques used, the psychological validity of the models that are produced, and the practical utility of the type of system we have discussed.

6.1. Generality of the Approach

We can divide student modeling systems into three categories, based on the amount of domain-specific knowledge they employ. First, there are systems that rely on a user-supplied bug library, listing the types of errors that are likely to occur. These can be used in a generate and test manner, or as part of a heuristic search scheme, as in LMS. However, such systems rely on an extensive analysis of student data by humans in order to generate the bug library. The generality of such a system is limited to the particular task domain in which the empirical study was done. This may not be a very serious limitation, since the number of school subjects is rather small. Also, the educational system develops rather slowly, compared to the time it takes to perform such a study. Thus, it may be quite feasible, in the long run, to perform empirical "bug studies" for each domain.

Other systems still rely on a bug library, but are capable of automatically extending this library when new bugs are encountered. Sleeman's extension of LMS falls into this category. The most natural way to use such a system is to "prime" it with an incomplete initial bug library, and let it fill in the missing error types. It could also be used interactively, aiding the programmer in searching through masses of student data for places where unfamiliar bugs occur. Sleeman has used his program in both of these modes. Such systems seem to be transitional in nature, though they have considerable potential for true generality.

The third type of system does not rely on a bug library. It does not even use a list of self-generated bugs in formulating models of student behavior. Instead, it *constructs* a procedure to account for the pupil's answers, without explicitly representing the relations between that procedure and the correct procedure for the task. This is the kind of system we are trying to build. We favor such an approach because we believe that dependence on *any* kind of bug library, user-generated or system-generated, is undesirable. The human mind is not easily bounded; for every bug library, there will always be a pupil somewhere that transcends it.

One could argue that in student modeling, only the most common bugs are important. The evidence to date indicates that the frequency distribution of bugs is highly skewed. The ten most common bugs account for nearly 90 percent of the errors made in the subtraction domain. Therefore, one could argue, not only will systems with bug libraries prove useful, but rather small bug libraries will probably be sufficient for practical work. We want to question this line of reasoning. From a *practical* point of view, it is not clear that the most frequent bugs are the most important. On the contrary, teachers may be familiar enough with the most frequent bugs that they can recognize them by direct observation of pupil behavior. Because they are frequent, they may very well be exactly those bugs with which the educator needs the least help from an automatic modeling system. In contrast, infrequent or unusually complex bugs or bug combinations may overtax the diagnostic capabilities of the teacher, making their diagnosis by computer a primary pedagogical objective. From the *research* point of view, there seems to be no reason to expect frequent bugs to be more

informative with respect to the mental representation of skills than infrequent bugs. A psychological theory of mental skills should explain the variation in the frequency of bugs – which implies that infrequent bugs are not to be dismissed from the analysis.

Another advantage of the approach we have taken with ACM relates to the issue of multiple bugs. Since DEBUGGY modeled a student's errors in terms of deformations from the correct algorithm, it encountered combinatoric possibilities when attempting to model students with multiple bugs. In contrast, ACM takes no longer to model students with three or four bugs than it does to model students with single bugs. In fact, the system takes as long to model *correct* subtraction behavior as it does to model complex *buggy* behavior. Of course, this last feature of the system could be easily removed by including a simple check for correct answers, but the main point remains. Rather than constructing buggy models in terms of deformations from the standard procedure, ACM constructs its models from the ground up, and in doing so, bypasses the combinatorics usually associated with multiple bugs.

In summary, there seem to be clear reasons why approaches that do not depend on a bug library should be preferred over those that do. The techniques we have been using to achieve this are exhaustive search and discrimination learning. The exhaustive search through the problem space finds the desired solution paths, and discrimination generates the rules which reproduce those paths. The generality of discrimination as a learning technique has already been demonstrated elsewhere (Langley, 1983a). It has been used to model concept learning and language development, as well as strategy acquisition. Thus, with respect to this part of our system, we feel confident that the technique we are using has broad generality.

The exhaustive search component does not fare as well. Although exhaustive search is a very general method, it cannot be used efficiently when the search space is too large. Thus, the *practical* generality of the current approach hinges on an estimate of which task domains generate problem spaces small enough to allow exhaustive search to be useful. In this context, we believe that most aspects of high-school level mathematics should fall within the scope of our methods. Moreover, Mitchell, Utgoff, and Banerji (1983) have applied very similar techniques to learning search heuristics for symbolic integration, so our approach should be extendable to this domain as well. Thus, though limited in principle, our basic approach may well be sufficient for a broad range of school subjects.

The notion of searching a set of successively more fine-grained problem spaces also contributes to alleviating this difficulty. The more aggregated the space, the smaller it will be. Thus, even in task domains in which the most fine-grained space would be too large to search exhaustively, it may still be true that most of the higher-level spaces are small enough. In such situations, one could still allow diagnosis to proceed for some proportion of the pupil population, namely those who can be modeled in the smaller spaces.

6.2. Psychological Validity

A second important issue is the psychological validity of the models produced by the student modeling system. If a model is going to serve as the basis for remedial instruction – presumably the objective of student modeling – then it must be as accurate a representation of the pupil's knowledge state as possible. A procedure which is dramatically different from the student's actual strategy, but which happens to give the same answer on the test problems, will be of little aid in designing remedial instruction, and may actually hinder this process.

Unfortunately, the field of computer simulation has not developed any commonly agreed upon criteria for psychologically valid simulation programs. This aspect of a student modeler is therefore difficult to evaluate. Nevertheless, a few observations present themselves. First, the intuition of most researchers in cognitive psychology is that human cognition is heavily goal-oriented, and that skills are hierarchically structured. This aspect is *not* represented in the computational approach we have described. The student

models generated by ACM consist of "flat" production systems, without hierarchical decomposition of the subtraction skill and without use of explicit goal-expressions.

Despite ACM's success in the subtraction domain with flat models, the absence of goals is clearly an oversimplification which should be corrected in future versions of the system. On the encouraging side, Laird (1983) has shown that the problem space hypothesis has no difficulty incorporating goal-based search methods, so our basic approach to student modeling should remain intact. On the other hand, the field of machine learning does not provide an abundance of tools for moving through a procedure space where the procedures are hierarchically structured. Indeed, only a few learning systems deal with goals in any explicit way. One of these is Ohlsson's (1982) work on learning search heuristics in the framework of means-ends analysis. Another is VanLehn's (1983) recent work on step theory, which models the process of acquiring a goal hierarchy. In our future work, we could do much worse than borrow ideas from these research efforts.

Another aspect of psychological validity is the way in which the *allocation of attention* is represented in the models. Human visual processing involves a small area of *focal attention*, in which the information is fully available, surroundered by a larger area of *peripheral attention*, within which information is only partially available. The content of focal awareness is manipulated by the near-constant movement of the eyes. This structure is not represented in the ACM program. We note with interest a principle adopted in VanLehn's step theory. In this framework, attention is locked to goals, so that one can attend only to objects used as parameters in the current goal. Attention can be shifted only by creating a new goal, or by popping the current goal from the goal stack.

A related issue involves the limitations on short term memory capacity, and their representation in student models. Unfortunately, research on short-term memory has failed to produce any commonly agreed upon explanation of these limitations, so it is not clear how to include them in such models. Nevertheless, this phenomenon is significant enough to deserve some discussion. As an illustration of the importance of attention and memory limitations in understanding subtraction errors, consider the "borrow-add-is-ten" bug described by VanLehn (1982). In this bug, rather than adding ten to a number, the student *replaces* it with ten. Such an error might occur in the following way. The child begins the process of adding ten to a number by scratching out that number. He may then become involved in some other activity, such as decrementing, and only later return to the goal of adding ten. At this point, he may perceive the scratch mark as a signal that the digit has been "taken out" or replaced by zero. Upon adding the borrowed ten to the column, he will exhibit the observed bug.

The plausibility of this particular "bug story" is not especially relevant. The main point is that attentional considerations may be very important in the modeling of bugs. This means, in turn, that the procedures considered by a student modeler should include some explicit mechanism for the allocation of attention. In other words, the attention mechanism should be represented in such a way that bugs can occur in the allocation of attention alone.

In summary, the psychological validity of ACM's student models can be criticized on three dimensions – the absence of goal structures, the poor modeling of attention processes, and the failure to address short term memory limitations. Each of these areas should be the subject of further work.

6.3. Practicality of the Approach

It is perhaps a bit premature to judge any student modeling system on its practical utility, since this paradigm is still in its initial stages. Nevertheless, it may be worthwhile to consider the *potential* practicality of the present approach. There are many dimensions along which a student modeling system may be practical or impractical. The most obvious involves the speed at which the system can diagnose a student. All systems will improve on this dimension as hardware becomes faster, but there are limits to this improvement, and one can

imagine approaches that lead to such combinatorial explosions in search that they could never be used in real time. On this dimension, our approach fares well, since even the current system is capable of dealing with ten or so subtraction problems in a few hours of CPU time. Combined with the methods we have outlined for reducing the search requirements, a factor of 100 increase in processor speed would put our system well within the practical range.

A second level of practicality relates to the ability of teachers to use the system. Our approach also does well on this aspect. Obviously, one cannot expect grade school teachers to read the production rules currently output by the system, but English paraphrases of these rules should be quite easy to understand, and simple to generate automatically. Moreover, these rules could be contrasted with those composing the correct strategy, in terms of missing operators and missing or incorrect conditions on existing operators. In this way, the teacher could see the difference between the student's procedure and the desired one, and this in turn might suggest particular remedial instruction.

A final dimension of utility involves the percentage of problems on which the system arrives at a useful diagnosis. Student modeling systems that rely on an explicit bug library will seldom score well on this criterion, since some creative student will always be inventing new ways to revise the standard algorithm. This is precisely the problem that motivated us to explore methods for student modeling that synthesized their own buggy explanations of errorful behavior. Of course, ACM's ability to model a particular student does require that he solve problems in the same basic problem space that the system is searching. However, we would argue that no conceivable system can do better than this, unless it is capable of searching the space of problem spaces, and we have started preliminary work on this method as well. To summarize, we believe that our basic approach to student modeling has excellent potential for eventual application in the schools, though much work will be necessary before we reach that stage.

References

Anderson, J. R. Tuning the search of the problem space for geometry proofs. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence,* 1981 , 97-103.

Brazdil, P. Experimental learning model. *Proceedings of the Third AISB/GI Conference,* 1978 , 46-50.

Brown, J. S. and Burton, R. R. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science,* 1978, *2,* 155-192.

Brown, J. S. and VanLehn, K. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science,* 1980, *4,* 379-427.

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

Feigenbaum, E. A. The simulation of verbal learning behavior. In E. A. Feigenbaum and J. Feldman (Eds.), *Computers and Thought,* New York: McGraw-Hill, Inc., 1963.

Hayes, J. R. and Simon, H. A. Understanding written problem instructions. In L. W. Gregg (Ed.), *Knowledge and Cognition,* Hillsdale, N. J.: Lawrence Erlbaum Associates, 1974.

Hayes-Roth, F. and McDermott, J. Learning structured patterns from examples. *Proceedings of Third International Joint Conference on Pattern Recognition,* 1976 , 419-423.

Hunt, E. B., Marin, J., and Stone, P. J. *Experiments in Induction.* New York: Academic Press 1966.

Korf, R. E. Toward a model of representation change. *Artificial Intelligence,* 1980, *14,* 41-78.

Laird, J. and Newell, A. *A universal weak method.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1983.

Laird, J. E. *Universal Subgoaling.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1983.

Langley, P. A general theory of discrimination learning. In D. Klahr, P. Langley, and R. Neches (Eds.), *Production System Models of Learning and Development,* Cambridge, Mass.: MIT Press, 1983.

Langley, P. Learning search strategies through discrimination. *International Journal of Man-Machine Studies,* 1983, *18,* 513-541.

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

Lenat, D. B., Sutherland, W. R., and Gibbons, J. Heuristic search for new microcircuit structures: An application of artificial intelligence. *AI Magazine,* Summer 1982 , 17-33.

Michalski, R. S. Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 1980, *2,* 349-361.

Mitchell, T. M. Version spaces: A candidate elimination approach to rule learning. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* 1977 , 305-310.

Mitchell, T. M., Utgoff, P., and Banerji, R. B. Learning problem solving heuristics by experimentation. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach,* Palo Alto, CA: Tioga Press, 1983.

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

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

Nilsson, N. J. *Problem Solving Methods in Artificial Intelligence.* New York: McGraw-Hill 1971.

Ohlsson, S. On the automated learning of problem solving rules. *Proceedings of the Sixth European Meeting on Cybernetics and Systems Research,* 1982 .

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

Quinlan, R. Learning from noisy data. *Proceedings of the International Machine Learning Workshop,* 1983 , 58-64.

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

Sleeman, D. H. Inferring (mal) rules from pupils' protocols. *Proceedings of the European Conference on Artificial Intelligence,* 1982 , 160-164.

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

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

VanLehn, K. Human procedural skill acquisition: Theory, model, and psychological validation. *Proceedings of the National Conference on Artificial Intelligence,* 1983 , .

Vere, S. A. Induction of concepts in the predicate calculus. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence,* 1975 , 281-287.

Winston, P. H. *Learning structural descriptions from examples.* Technical Report AI-TR-231, Massachusetts Institute of Technology, 1970.

Young, R. M. and O'Shea, T. Errors in children's subtraction. *Cognitive Science*, 1981, *5*, 153-177.