4

Information and Computer Science

RULES AND PRINCIPLES IN COGNITIVE DIAGNOSIS

Pat Langley
James Wogulis
Irvine Computational Intelligence Project
Department of Information & Computer Science
University of California, Irvine, CA 92717

Stellan Ohlsson
The Center for the Study of Learning
Learning Research and Development Center
University of Pittsburgh
Pittsburgh, Pennsylvania 15260

TECHNICAL REPORT

**UNIVERSITY OF CALIFORNIA
IRVINE**

4

RULES AND PRINCIPLES IN COGNITIVE DIAGNOSIS

Pat Langley
James Wogulis
Irvine Computational Intelligence Project
Department of Information & Computer Science
University of California, Irvine, CA 92717

Stellan Ohlsson
The Center for the Study of Learning
Learning Research and Development Center
University of Pittsburgh
Pittsburgh, Pennsylvania 15260

Technical Report 89–02

January 1, 1989

To appear in N. Fredericksen (Ed.), *Diagnostic Monitoring of Skill and Knowledge Acquisition.* Lawrence Erlbaum Associates, 1989.

We would like to thank Wayne Iba for his useful discussions on this work and for his comments on an earlier version of this paper.

This work was supported in part by contract N00014-85-K-0373 from the Personnel and Training Research Group, Office of Naval Research. Preparation of this manuscript was sponsored in part by the OERI institutional grant for The Center for the Study of Learning.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report No. 6	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Rules and Principles in Automated Cognitive Diagnosis		5. TYPE OF REPORT & PERIOD COVERED Interim Report 1/1/88–12/31/88 6. PERFORMING ORG. REPORT NUMBER UCI–ICS Technical Report 89–02
7. AUTHOR*(s)* Pat Langley James Wogulis Stellan Ohlsson		8. CONTRACT OR GRANT NUMBER*(s)* N00014-85-K-0373
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Information & Computer Science University of California, Irvine, CA 92717		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Cognitive Science Program Office of Naval Research Arlington, Virginia 22217		12. REPORT DATE January 1, 1989 13. NUMBER OF PAGES 33
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

To appear in N. Fredericksen (Ed.), *Diagnostic Monitoring of Skill and Knowledge Acquisition*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1989.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

student modeling	cognitive diagnosis
problem space	heuristic search
production systems	machine learning

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

OVER

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT

Cognitive simulation is concerned with constructing process models of human cognitive behavior. Our work on the ACM system (*A*utomated *C*ognitive *M*odeler) is an attempt to automate this process. The basic assumption is that all goal-oriented cognitive behavior involves search through some problem space. Within this framework, the task of cognitive diagnosis is to identify the problem space in which the subject is operating, identify solution paths used by the subject, and find conditions on the operators that explain those solution paths and that predict the subject's behavior on new problems. The work presented in this paper uses techniques from machine learning to automate the tasks of finding solution paths and operator conditions. We apply this method to the domain of multi-column subtraction and present results that demonstrate ACM's ability to model incorrect subtraction strategies. Finally, we discuss the difference between procedural bugs and misconceptions, proposing that errors due to misconceptions can be viewed as violations of principles for the task domain.

A Framework for Cognitive Diagnosis

One of psychology's central goals is the description of human behavior. However, it has become increasingly clear that psychology faces several complications in its pursuit of that goal. The most important of these complications is that individual differences appear to be an essential aspect of the human condition. Different people have different experiences and thus acquire different knowledge and skills. Moreover, the behavior of even a single human varies over time as he gains more experience and more expertise.

In this chapter, we present one response to the problems of individual differences and of change over time. Our approach builds upon three related research methodologies. The first is cognitive simulation, a paradigm for constructing process models of human cognitive behavior. The second is heuristic search, one of the mainstays of artificial intelligence. The third component is machine learning, the subfield of artificial intelligence that focuses on computational methods for improving performance over time. Our goal is to *automate* the process of constructing cognitive simulations. This is the problem of *cognitive diagnosis* (Ohlsson & Langley, 1986), and we employ methods from heuristic search and machine learning to this end.

Within the field of computer-aided instruction (CAI) and intelligent tutoring systems, the problem of cognitive diagnosis goes by another name - *student modeling.* Unlike most CAI systems, human teachers employ some model of the student's knowledge state to determine their instructional actions. Intelligent teaching systems would also gain from such student models, but they must first have some means to infer the student's knowledge from his or her overt behavior. However, this problem is equivalent to that confronting the cognitive psychologist attempting to construct a model of some subject, so we will use the more general term *cognitive diagnosis* in our discussion.

Testing and Cognitive Simulation

Before delving into the problem of cognitive diagnosis and our proposed solution, we should review a major alternative to this approach. The testing tradition in psychology has a long and venerable history, dating back to Galton and Binet. The basic paradigm involves identifying dimensions of intellectual variation and designing tests to measure ability along these dimensions. Thus, test psychology has explicitly focused on the problem of individual differences, and has led to many practical results over the years. Tests are easy to apply and interpret, making them attractive in many settings. This approach has been particularly useful in predicting academic success, making it quite popular within educational psychology.

However, the testing paradigm makes a number of limiting assumptions. First, it does not attempt to generate a theoretical description of an individual's state of knowledge; it is content to summarize a person's behavior in terms of empirically-derived test scores. Second, test psychology interprets these scores as stable properties of the individual, thus ignoring the changes that occur over time as the result of learning. Finally, the testing framework focuses on quantitative descriptions of behavior, and this representation severely limits its descriptive and explanatory power.

Now let us turn to the paradigm of cognitive simulation, a quite different approach with a much shorter history, first formulated by Newell, Shaw, and Simon (1960). Like the testing approach, this framework also focuses on individual analyses. However, it differs in that the goal is a detailed description of the subject's cognitive behavior. This description takes the form of a running computer program that simulates the subject's performance, hence the term *cognitive simulation.* These simulations specify the structures and processes leading to behavior, rather than providing numeric descriptions like those in test psychology. Finally, the approach also has potential for modeling learning, and thus accounting for changes in performance over time.

Despite these advantages, the methodology of cognitive simulation has gained relatively few adherents in the years since its inception. Undoubtedly, one reason is the difficulty of gathering and analyzing data such as verbal protocols, and another is the skill and patience involved in constructing a detailed cognitive simulation that explains these data. In contrast, tests are easy to administer and their results are simple to interpret. Our long-term goal is to automate the process of constructing simulation models, making the framework of cognitive diagnosis as manageable and attractive as the testing approach.

Assumptions for Cognitive Diagnosis

Before we can automate the diagnostic process, we must establish the framework within which we will operate. There exist a variety of schemes for modeling cognitive processes in computational terms, and the more constrained our models, the better chance we will have to automate their construction. Let us consider the assumptions underlying our cognitive models, along with the alternatives.

The most basic assumption is that knowledge is represented using *symbols*, and that these symbols are grouped into propositions. Newell (1980) has referred to this general approach as the *physical symbol system* hypothesis. For instance, we can represent the subtraction problem 43 − 21 with propositions like `(in 4 column-1 row-1)`, `(in 3 column-2 row-1)`, and so forth. In such representations, the symbols themselves have meaning; thus, `4` represents the number 'four' and `row-1` stands for a particular position.[1] One could easily map our propositional scheme onto other symbolic paradigms, such as semantic networks. However, both frameworks contrast sharply with research in the connectionist paradigm that relies on the *subsymbolic* hypothesis – that meaning is represented by configurations of nodes in a network (Anderson & Hinton, 1981). We will not attempt to justify our alignment with the symbolic approach, except to note that this approach has proven very useful in modeling a wide range of cognitive behavior.

Of course, a cognitive simulation requires more than a representation of knowledge – it also requires some processes to manipulate that knowledge. These processes can be organized in various ways, and computer science has traditionally focused on *algorithmic* control schemes. However, this is not the only such organization, and artificial intelligence has emphasized the alternative organization of *heuristic search.* In this framework, one system-

[1] These are examples of declarative knowledge, but as we will see shortly, we also assume that procedural knowledge is represented symbolically.

atically searches a space of possible states, deciding at each point which is most likely to lead to a problem solution. Many AI tasks lend themselves to this approach, including problem solving and language understanding.

A central concept in the heuristic search framework is the *problem space*, which can be defined in terms of an initial problem state and a set of operators for generating new states. Newell and Simon (1972) have put forward the *problem space* hypothesis - that all human cognition involves search through some problem space. In this view, even algorithmic behavior can be viewed as search, though this search is so constrained that behavior is determined at each point. We will rely heavily on the problem space hypothesis in the following pages. Again, we will not argue for this assumption, except to mention that it has been successfully applied to many aspects of intelligent behavior.

Although the problem space hypothesis significantly constrains one's models of cognition, it does not specify the representation of operators or the heuristics used to control search. One response is to represent operators as opaque procedures (e.g., Lisp functions) that are applied to a given state. The alternative is to represent operators as transparent *production rules* in which the conditions and results are stated as abstract propositions. One can represent heuristic knowledge as some numeric evaluation function that ranks states according to predicted distance to the goal. This approach has a sizable following within artificial intelligence, partially because it is amenable to formal analysis (Pearl, 1984). However, one can also represent control knowledge as heuristic conditions on rules, and this scheme is a natural companion to the transparent operator representation.

Newell (1972), Newell (1973), Anderson (1976), and others have proposed production rules as a general framework for modeling human cognition. A *production system* is a collection of production rules that operates in cycles. On each cycle, the conditions of each rule are matched against the contents of a dynamic working memory. From those rules with conditions that successfully match, some are selected for application and their actions are carried out. These actions alter the contents of working memory, causing new rules to match on the next cycle. This recognize/act cycle continues until no rules are matched or an explicit halt action is evoked. We will assume that all procedural knowledge is represented as such condition-action rules. This *production system* hypothesis constitutes our third assumption. Taken together, our three assumptions significantly restrict the class of models that we must consider in constructing cognitive simulations.

Three Stages of Cognitive Diagnosis

Newell and Simon (1972) identified three successive stages in the diagnostic process, each providing a more detailed model of behavior. First, one must identify the problem space in which a given subject is operating. This involves selecting some representation for states and operators, including the legal conditions on the latter. One must also specify some termination criterion; this is usually represented as one or more tests for detecting when a goal state has been reached. Identifying the subject's problem space is a very difficult task, and we have not attempted to automate this aspect of cognitive diagnosis.

Second, one must identify the path followed by the subject while working on each problem. This involves examining the subject's behavioral record and inferring a sequence of state selections and operator applications. Newell and Simon (1972) employed the method of protocol analysis to this end, but verbal protocols are difficult to analyze and not always available, especially in educational settings.[2] Another approach is to focus on *answer data*, such as a student's responses to test questions. Although such data have a lower temporal density than verbal protocols, they are much easier to collect and may limit the diagnostician to one or a few path hypotheses.

Finally, one must identify some production system model of the subject's behavior that accounts for his or her performance across a number of problems. This set of condition-action rules makes up a running cognitive simulation, and constitutes an hypothesis about the strategy the subject used to solve the problems. Of course, knowledge of the problem space and path traversal can be used to constrain this model. Unfortunately, one cannot always assume that a subject's behavior is consistent across different problems, and this makes the inference process more difficult. Also, we will see later that for educational applications, such models may be less useful than the path hypotheses themselves. Thus, it is not clear that one always need generate a complete simulation of the subject's behavior.

The Subtraction Domain

Early work in cognitive diagnosis focused on puzzle solving and reasoning tasks. However, one of the most carefully analyzed areas is that of multi-column subtraction problems, and we have chosen this domain to test our approach to automated diagnosis. Below we consider the nature of these problems and the types of errors that arise when students attempt to solve them. After this, we review some previous approaches to diagnosing these errors.

The Standard Subtraction Algorithm

At first glance, the task of multi-column subtraction seems relatively straightforward: given two multi-digit integers, one must simply find their difference. For instance, $345-211 = 134$, $642-13 = 629$, and $406-138 = 268$. However, the standard algorithm for such problems contains unexpected complexity, and this complexity causes difficulty for many students of arithmetic. Close examination of the three examples above reveals some of the reasons.

The first example, $345 - 211$, is the simplest of the set. In this case, one merely finds the difference in the rightmost column (4) and writes down this result. One then moves to the second column, finds the difference there (3) and records the result. Finally, one moves to the leftmost column, finds the last difference (1), and writes the final result, giving the complete answer 134.

The second problem, $642 - 13$, is more difficult because it involves borrowing. One cannot directly subtract 3 from 2, since this would give a negative result. Instead one moves

[2] Waterman and Newell (1972) made some progress towards automating the analysis of verbal protocols, but more work must be done on this problem before it reaches the applications stage.

to the adjacent column, decrements the 4 by one to give 3, and then adds ten to the original column, transforming the existing 2 into 12. Now one can subtract 3 from 12, producing the result 9. Next one shifts to the second column and finds the difference between 3 (the decremented 4) and 1, records the result of 2 in this column, and moves to the leftmost column. Since no digit exists in the second row, the digit in the top row (6) is carried down into the result, giving the final answer 629.

Our third example (406 − 138) is even more complicated, since it involves borrowing from zero. As before, one cannot subtract 8 from 6, so the only option is to borrow from the top digit in the adjacent column. However, one cannot borrow from zero, so one must move over yet another column. Borrowing from the 4 in the leftmost column transforms this digit into 3 and the 0 in the middle column into 10. Now one can borrow from the 10, replacing this number with 9 and the 6 (the original cause of the problem) with 16. These actions let one compute the results for each column in turn, giving the final result 268.

Table 1
Common Subtraction Bugs

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

Errorful Subtraction Algorithms

Brown and Burton (1978) and VanLehn (1982) have carried out detailed analyses of student's subtraction behavior, noting the different types of errors that occur on various problems. They have identified over 100 basic 'bugs' that students make in this domain. Table 1 lists the eleven most common subtraction bugs, along with their relative frequency and with an example of each error type. Let us consider some of these procedural misconceptions.

Nearly all subtraction errors occur on borrowing problems. One of the most common borrowing bugs is the 'smaller-from-larger' strategy. In this algorithm, the student subtracts the smaller digit from the larger in a given column, regardless of which is above the other.

Thus, the answer generated for 81 − 38 would be 57 (rather than the correct answer 43). Note that this method bypasses the need for decrementing digits and adding ten, making it much simpler than the correct strategy.

Some types of errors occur only when borrowing from zero is involved. For instance, in the 'borrow across zero' bug, the student decrements the digit left of the zero twice to avoid borrowing from zero. Thus, this algorithm generates 577 for the problem 904 − 237, rather than the correct answer 667. The 'borrow across zero over zero' is a variation on this bug, in which the errorful behavior is only invoked when two zeros occur in the same column. This strategy generates 408 (instead of 498) as the answer for 802 − 304, while producing the correct answer for 904 − 237.

Young and O'Shea (1981) have used the label *pattern errors* to refer to another class of misconceptions. One of these is the '$0 - N = N$' bug, in which the student avoids borrowing by using the digit in the second row as the answer for the current column. The '$0 - N = 0$' bug has a very similar flavor. For the problem 50 − 23, the first strategy generates the answer 33, while the second method produces 30 as the final result. Special cases of these algorithms also exist, such as '$0 - N = N$ except after borrow'.

These examples give only a flavor of the errors that can occur on multi-column subtraction problems, but they should help the reader understand the complexity of the behaviors we hope to diagnose. Moreover, the various bugs may occur in combination, producing even more complex errors than is possible by each in isolation. Finally, students are not always consistent in their errorful behavior, sometimes using different algorithms at the beginning and the end of the same test; VanLehn (1982) has called these *bug migrations.*

DEBUGGY and Bug Libraries

In addition to empirical studies of subtraction errors, a number of researchers have developed computational models of this behavior. The earliest and best known effort along these lines was Brown and Burton's (1978) DEBUGGY system. This system used a hierarchical procedural network to represent the correct algorithm for multi-column subtraction. Associated with some nodes in this network were 'buggy' versions of that subroutine. If one of these buggy routines were used in place of the standard node, the algorithm would still run, but it would generate an incorrect answer

Langley, Ohlsson and Sage (1984) call this the *bug library* approach to cognitive diagnosis. One of its main advantages is that it decomposes behavior into a number of relatively independent components. These can be used in isolation or in combination to explain observed deviations from correct behavior. The key word here is 'deviations'. The bug library framework attempts to model errors as minor variations on the correct strategy. This approach has also been successfully applied to other domains; for instance, Sleeman and Smith (1981) have used bug libraries to model errors in algebra problem solving.

The main disadvantage of bug libraries is that they require extensive empirical analysis to identify bugs that actually occur in the domain. For example, VanLehn (1982) has reported over 100 distinct procedural errors that occur in multi-column subtraction, and DEBUGGY would represent each of these misconceptions as another alternative subroutine in its procedural network.[3] In addition, the farther a subject's behavior from the standard algorithm, the more difficult it will be to develop a buggy model. Different bugs can interact with one another, making it difficult to diagnose algorithms involving multiple misconceptions.

However, DEBUGGY did much more than represent errorful subtraction algorithms; the system also *inferred* a student's strategy from his observed answers. The simplest approach would involve a generate-and-test scheme in which each bug is inserted into the procedural network and checked to see whether it explains the student's behavior. If none of the bugs are sufficient by themselves, then all pairs of bugs would considered, then triples, and so forth. Although this scheme is guaranteed to work if the subject's behavior is covered by the bug library, it is combinatorial in nature and thus very expensive. In fact, DEBUGGY used a more sophisticated diagnostic method based on a discrimination network. Particular types of errors would suggest different bugs, and the system would then gather the evidence for each competing hypothesis.

Production System Models of Subtraction

Young and O'Shea (1981) have taken a quite different approach to explaining subtraction errors. Rather than representing arithmetic algorithms as procedural networks, they employed a production system formalism. The researchers modeled the standard subtraction strategy as a set of condition-action rules, explaining errors in terms of slight variations on this model. In many cases, they were able to model errors simply by deleting one or two production rules. Using this approach, they explained many of the most common subtraction bugs described by Brown and Burton.

Although Young and O'Shea did not implement an automated diagnostic system, the rule deletion approach provides a natural foundation for such a system. Given a set of production rules for the correct algorithm, one can easily check to see whether removing various rules will explain a subject's errors. This approach is possible because production rules are relatively independent of each other and, when a production system model is carefully formulated, it will continue to run (though incorrectly) when rules are removed. Although the nodes in a procedural network are independent, they do not have this latter feature.

Unfortunately, Young and O'Shea were forced to introduce additional production rules to model certain subtraction errors, and this complicates the diagnostic method we just outlined. These rules contained standard actions for the subtraction domain, but applied those actions under the incorrect conditions. One could generate a set of such incorrect rules and employ them in diagnosis, but with this modification the approach begins to take on the flavor of a bug library.[4]

[3] Brown and VanLehn (1980) have tried to remedy this drawback with their *repair theory*, which attempts to explain the origin of subtraction bugs.

[4] In fact, Sleeman and Smith (1981) used just such a rule-based approach in their LMS system for algebra diagnosis. Their program included both correct algebra productions and 'mal-rules' which led to errors.

A New Approach to Subtraction Diagnosis

Although the Young and O'Shea framework has limitations, it points the way to a more flexible approach to diagnosis. If we assume that *all* errors are due to rules with the correct actions but the wrong conditions, then diagnosis consists of determining the subject's conditions for each rule in the domain. This is the approach we will take to automating the diagnostic process in the subtraction domain. The 'correct actions' correspond to the operators in a problem space, while the 'subject's conditions' correspond to the heuristic conditions for applying those operators.

Identifying a Problem Space

As we have seen, the first step in our approach requires one to identify the problem space in which the subject is operating. This is a very difficult problem and little work in AI and cognitive science has addressed the issue. Rather than automating this aspect of the diagnostic process, we will use the standard subtraction space as defined by Ohlsson and Langley (1986).

The first step in defining a problem space involves specifying the *problem states* contained in that space. We will represent states in the subtraction domain as lists of relations between objects; this decision is consistent with previous analyses of student problems (VanLehn, 1982). The objects consist of digits, columns, and rows in the problem display. Relations between these objects include predicates like *above*, *left-of*, *in*, and so forth. A given problem state is represented as a set of such relations between objects. For example, the initial state for the two column subtraction problem 93 − 25 would be represented by the list of relations shown in Table 2.

Table 2
Initial problem state for 93 − 25.

`(in 9 column-2 row-1)`	`(above row-1 row-2)`
`(in 2 column-2 row-2)`	`(left-of column-2 column-1)`
`(in 3 column-1 row-1)`	`(processing column-1)`
`(in 5 column-1 row-2)`	`(focused-on column-1)`
`(result blank column-2)`	
`(result blank column-1)`	

In addition to such relational information, a problem state can also contain control information. For example, the statement `(processing column-1)` in the table signifies that the subject is attempting to find a result for `column-1`. Similarly, the `focused-on` predicate indicates which column is being considered for borrowing operations. Although not shown in the table, a state may also include information about the last operation performed. For example, after the **Find-Difference** operator has been applied, the problem description would contain the element `(just-did find-difference)`.

Table 3 presents one set of operators for multi-column subtraction. These operators can be used to generate new problem states, and this lets one systematically search the problem space, starting from the initial state. The first two operators - **Add-Ten** and **Decrement** - are responsible for borrowing. The second pair - **Find-Difference** and **Find-Top** - write the result for a particular column. The final three operators - **Shift-Column**, **Shift-Left**, and **Shift-Right** - influence the control symbols `processed` and `focused-on`.

These operators assume a certain set of primitive actions, including the ability to add ten to a digit, subtract any two numbers between 0 and 19, to shift attention between columns, and to write a result for a column. Such primitive skills define a given level of abstraction, but note that other choices are possible. For instance, we could have also broken the **Decrement** operator into three simpler operators, one for crossing out a digit, another for subtracting one from that digit, and a third for writing the decremented number in its place. This level of description would let us describe subjects' behaviors in more detail, but only at the cost of a much larger problem space.

Table 3
Operators for subtraction.

Add-Ten(number, row, column) Takes the number in a row and column and replaces it with that number plus ten.

Decrement(number, row, column) Takes the number in a row and column and replaces it with that number minus one.

Find-Difference(number1, number2, column) Takes the two numbers in the same column and writes the difference of the two as the result for that column.

Find-Top(column) Takes a number from the top row of column and writes that number as the result for that column.

Shift-Column(column) Takes the column which is both `focused-on` and being processed and shifts both to the column on its left.

Shift-Left(column) Takes the column which is `focused-on` and shifts the focus of attention to the column on its left.

Shift-Right(column) Takes the column which is `focused-on` and shifts the focus of attention to the column on its right.

Similarly, we could have replaced the **Decrement** and **Add-Ten** operators with a single **Borrow** operator, giving a more abstract problem space. This would produce fewer problem states, but we would be unable to model many observed subtraction errors. We believe that the operators in Table 3 constitute an optimal level of abstraction for the subtraction domain, but this assumption is open to empirical tests.

The final component of a problem space is the termination condition that specifies when search should halt. In the subtraction domain, this occurs when one has written results for all of the columns, and when these results agree with the subject's answer to the problem. Again, remember that the goal is not to find the correct answer, but to explain how the subject arrived at the observed answer.

Path Hypotheses

Given a problem space for subtraction and a subject's answer for a particular test problem, we must find some *path hypothesis* that explains the observed answer. Such a path hypothesis consists of the states traversed by the subject, along with the instantiated operators connecting those states.

For the problem-answer pair $93 - 25 = 68$, one explanatory path hypothesis that uses the operators from Table 3 would be:

```
Shift-Left(column-1)
Decrement(9, row-1, column-2)
Shift-Right(column-2)
Add-Ten(3, row-1, column-1)
Find-Difference(13, 5, column-1)
Shift-Column(column-1)
Find-Difference(8, 2, column-1)
```

For any problem-answer pair, there could be several path hypotheses, each one providing an explanation of how the subject could have obtained the observed answer.[5]

We can also use path-hypotheses to explain *incorrect* answers to subtraction problems. Ohlsson and Langley (1986) have shown how a number of different incorrect answers can be explained through paths in the standard subtraction space. For example, the 'smaller-from-larger' bug (in which the subject always subtracts the smaller number from the larger one in a given column) would give the answer 72 to the problem $93 - 25$. Within our problem space, one path hypothesis that explains this answer would be:

```
Find-Difference(5, 3, column-1)
Shift-Column(column-1)
Find-Difference(9, 2, column-2)
```

Note that this path is much shorter than the one needed to generate the correct answer. Empirically, buggy paths appear to be shorter than correct paths for the same problem. We will return to this fact when we consider methods for inferring path hypotheses.

Production System Models

Having generated plausible path hypotheses for a number of problem-answer pairs, we would like some production syste[illegible]del that will generate the inferred solution paths. The mapping from problem space ont[illegible]uction system is straightforward: the working memory

[5] In general, there may be m[illegible] than one 'correct' procedure or algorithm in that all produce the correct answer to all [illegible]btraction problems. For instance, the order in which one performs the **Add-Ten** and **Decrement** operators does not affect the final answer.

on each cycle corresponds to a particular problem state, the action sides of productions represent the operators for generating new states, and the condition sides of productions correspond to the legal and heuristic conditions on these operators.

In a traditional production system, multiple productions can match on any given cycle, and some *conflict resolution* scheme determines which rule to apply. However, one of our goals is to construct diagnostic models that can be easily understood, and we believe that reliance on conflict resolution methods obscures the semantics of such models. Instead, we will formulate production rules with mutually exclusive conditions; these will guarantee that only one rule will match on each cycle, and this in turn should lead to clearer models.

Table 4 presents a production system model of the standard subtraction algorithm. This model includes all of the operators from Table 3, and each rule contains both legal conditions (necessary to instantiate the action side) and heuristic conditions (necessary to ensure correct behavior). Heuristic conditions have been enclosed in braces to distinguish them from the legal conditions. (The reader should ignore conditions in brackets; we will return to these later.) This model generates the correct answers for all subtraction problems, including the borrowing problem 93 − 25 we saw earlier. In solving this problem, the model will produce the first path hypotheses we examined.

Note that two rules in the model - **Shift-Left-To-Borrow** and **Shift-Left-Across-Zero** - employ the **Shift-Left** operator in their action side. Each of these productions covers a different situation in which it is appropriate to shift the focus of attention to the left. This is equivalent to placing *disjunctive* conditions on the **Shift-Left** operator. In general, a diagnostic system may need to infer that a student uses an operator in such disjunctive situations.

But we are more interested in modeling incorrect subtraction behavior than the correct algorithm. Table 5 presents a production system model of the 'smaller-from-larger' bug described earlier. Some aspects of this incorrect strategy can be accounted for by missing operators; since borrowing never occurs, our model has no need for the **Decrement**, **Add-Ten**, **Shift-Right**, or either **Shift-Left** rules. The model includes **Shift-Column** and **Find-Top** in their correct forms, since students with the 'smaller-from-larger' bug still process all columns and behave correctly when no digit exists in the lower row.

However, the model's version of the **Find-Difference** rule lacks the *above* condition present in the correct model. This means it will subtract the smaller digit in a column from the larger, independent of their spatial relation. Given a non-borrowing problem like 54 − 31, this model will give the correct answer 23. But given a borrowing problem such as 93 − 25, the **Find-Difference** variant will subtract 3 from 5, giving the incorrect answer 72. In solving the problem in this manner, the system will generate the second path-hypothesis we saw above.

Table 4
Rules for correct subtraction strategy.

Add-Ten:
If *number-a* is in *column-a* and *row-a*,
{[and you are focused on *column-a*]},
{[and *row-a* is above *row-b*]},
{[and there is no result for *column-a*]},
{[and *number-a* is less than ten]},
{and you just did Shift-Right},
then replace *number-a* with *number-a* plus ten.

Decrement:
If *number-a* is in *column-a* and *row-a*,
{[and you are focused on *column-a*]},
{[and *number-a* is not zero]},
{[and *row-a* is above *row-b*]},
{[and you did not just Decrement]},
{[and you are not processing *column-a*]},
{[and *column-a* is to the left of *column-b*]},
{[and there is no result for *column-a*]},
{[and you have not added-ten to *column-b*]},
{and you did not just Shift-Right},
then replace *number-a* with *number-a* minus one.

Find-Difference:
If *number-a* is in *column-a* and *row-a*,
and *number-b* is in *column-a* and *row-b*,
{[and you are processing *column-a*]},
{[and you are focused on *column-a*]},
{[and there is no result for *column-a*]},
{and *row-a* is above *row-b*},
{and *number-a* is greater than or equal to *number-b*},
then write the difference between *number-a* and *number-b*
as the result for *column-a*.

Find-Top:
If *number-a* is in *column-a* and *row-a*,
and *row-a* is above *row-b*,
{[and you are processing *column-a*]},
{[and there is no result for *column-a*]},
{[and you are focused on *column-a*]},
{[and there is no number in *column-a* and *row-b*]},
then write *number-a* as the result for *column-a*.

Table 4. Rules for correct subtraction strategy (continued).

Shift-Column:
If you are processing *column-a*,
  and you are focused on *column-a*,
  and *column-b* is to the left of *column-a*,
  {[and there is a result for *column-a*]},
then shift your focus from *column-a* to *column-b*,
  and process *column-b*.

Shift-Left-to-Borrow:
If you are focused on *column-a*,
  and *column-b* is to the left of *column-a*,
  [and there is no result for *column-a*],
  [and you did not just Decrement],
  [and you did not just Add-Ten],
  {[and *number-a* is in *column-a* and *row-a*]},
  {[and *number-b* is in *column-a* and *row-b*]},
  {[and *row-a* is above *row-b*]},
  {[and you did not just Shift-Right]},
  {and you are processing *column-a*},
  {and *number-b* is greater than *number-a*},
then shift your focus from *column-a* to *column-b*.

Shift-Left-Across-Zero:
If you are focused on *column-a*,
  and *column-b* is to the left of *column-a*,
  [and *number-a* is in *column-a* and *row-a*],
  [and you did not just Decrement],
  [and you did not just Add-Ten],
  [and there is no result for *column-a*],
  {[and *row-a* is above *row-b*]},
  {[and you did not just Shift-Right]},
  {and you are not processing *column-a*},
  {and *number-a* is zero},
then shift your focus from *column-a* to *column-b*.

Shift-Right:
If you are focused on *column-a*,
  and *column-a* is to the left of *column-b*,
  [and you did not just Shift-Left],
  [and you are not processing *column-a*],
  {and you just did Decrement},
then shift your focus from *column-a* to *column-b*.

Table 5
Production rules for 'smaller-from-larger' bug.

```
Find-Difference:
  If number-a is in column-a and row-a,
    and number-b is in column-a and row-b,
    {[and you are processing column-a]},
    {[and you are focused on column-a]},
    {[and there is no result for column-a]},
    {and number-a is greater than or equal to number-b},
  then write the difference between number-a and number-b
    as the result for column-a.

Shift-Column:
  If you are processing column-a,
    and you are focused on column-a,
    and column-b is to the left of column-a,
    {[and there is a result for column-a]},
  then shift your focus from column-a to column-b,
    and process column-b.

Find-Top:
  If number-a is in column-a and row-a,
    and row-a is above row-b,
    {[and you are processing column-a]},
    {[and there is no result for column-a]},
    {[and you are focused on column-a]},
    {[and there is no number in column-a and row-b]},
  then write number-a as the result for column-a.
```

To review, our framework divides the task of cognitive diagnosis into three stages – identifying a problem space, generating a path hypothesis that explains the subject's behavior on each problem, and finding a production system model that explains these path hypotheses. We have chosen to focus on the domain of multi-column subtraction problems, both because of the empirical work that has been done in this area and because of the earlier diagnostic work. Now that we have seen some examples of problem spaces, path hypotheses, and production system models for this domain, let us examine a method for generating these hypotheses and models.

A System for Automated Cognitive Diagnosis

In order to test our approach to cognitive diagnosis, we have implemented ACM, an artificial intelligence system that constructs cognitive models of behavior. The user provides the system with an appropriate problem space and a set of problem-answer pairs. The program outputs a path hypothesis for each problem, along with a production system model that generates the inferred paths.

ACM's basic method rests on the notion of *problem reduction* (Nilsson, 1980), in which one decomposes a difficult problem into a number of independent, simpler problems. Once each of these subtasks have been solved, the results are recombined to form an answer to the original problem. We can apply this approach to cognitive diagnosis by realizing that the ultimate goal - generating a complete production system model - can be divided into a number of independent tasks - one for each operator in the problem space. In each case, we must determine the heuristic conditions on that operator that lets us predict when it will be invoked by the subject. Once we determine these conditions (which may be disjunctive), we simply combine the resulting production rules into our final model. The resulting set of rules can be used to simulate the subject's observed behavior and to predict his behavior on future problems.

In determining the conditions on an operator, ACM employs a method for *learning from examples.* A variety of such methods have been described in the machine learning literature (Mitchell, 1982; Michalski, 1983; Quinlan, 1983), but all rely on a division of the data into *positive* instances and *negative* instances. We will see that one can use path hypotheses to generate such a division, and that these instances can in turn be used to find the conditions on each operator. Thus, we will first consider ACM's approach to formulating path hypotheses and only then describe its condition-finding method. Finally, we will consider how these two components interact to produce a complete diagnostic system.

From Behavior to Path Hypotheses

Given a subtraction problem and a subject's answer to that problem, ACM must generate some path hypothesis to explain the observed answer. A path hypothesis consists of a sequence of operator instantiations that takes one from the initial problem state to the observed final state. But given only the legal conditions, many operators will match against most states. This leads to a combinatorial explosion of possible paths, and we need some means for managing the alternatives. Fortunately, AI provides a variety of methods for finding paths through combinatorial problem spaces, all revolving around the notion of *search.*

The basic action in these search methods consists of *expanding* a problem state by applying all operator instantiations that match against that state. One begins by expanding the initial state, selecting one of its successor states for expansion, generating still more states, and continuing this process until the termination condition is met or until no operators match. Alternative search algorithms differ radically in their methods for deciding which state to expand, but all generate a *search tree* of the alternative paths explored to date. The leaves of this tree represent those states which have not yet been expanded.

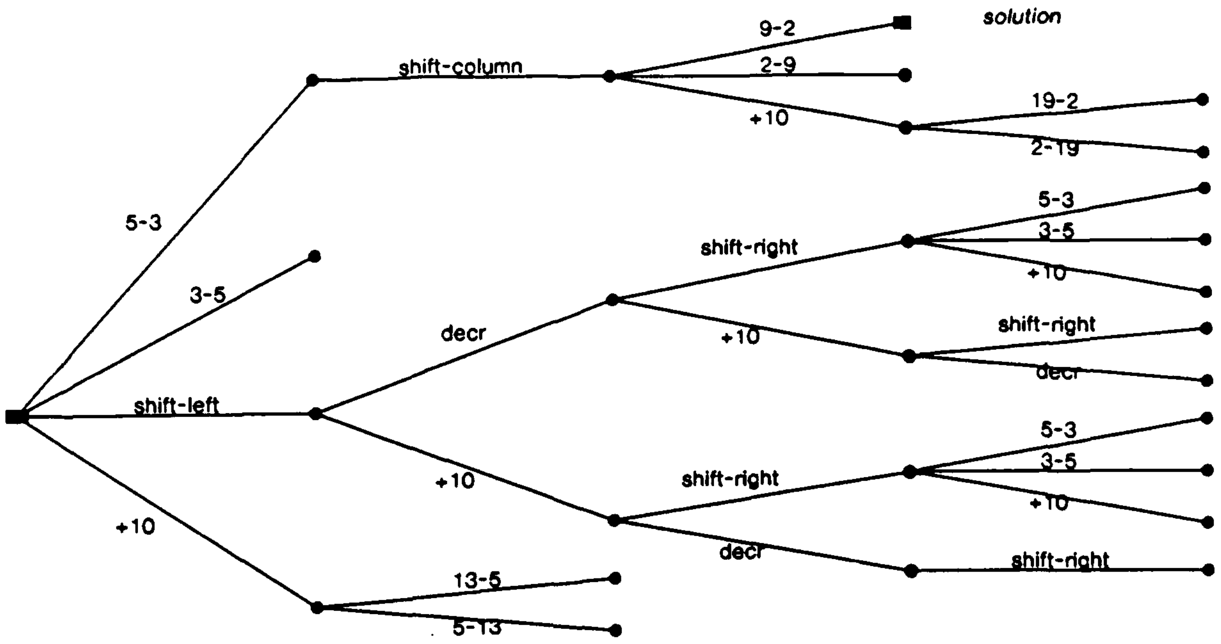

Figure 1. Search tree for solution to 93 - 25 = 72.

Figure 1 presents a search tree generated by ACM when solving the subtraction problem 93 − 25 = 72. The search tree shows all paths with five or fewer steps. For this problem, there was one solution path which led to a final state consistent with the subject's answer of 72. One issue that ACM must handle is the possibility that multiple path hypotheses will explain the subject's answer. In such cases, the diagnostic system must determine which path(s) provide the most plausible explanation of the observed behavior.

Different search methods arise when one alters the method of selecting which state to expand. If one selects the least recently generated state, then *breadth-first* search results. This method examines all states at the first level, then all states at the second level, and so forth until the solution is reached. Breadth-first search is guaranteed to find the shortest solution path, but its memory requirements are very high, since one must remember all states at the previous level. If one selects the most recently generated state, then *depth-first* search results. This method pursues a single path until the solution is found or until some depth-limit is reached. In the latter case, the algorithm backtracks to the next most recent state, follows it downward, and so forth. This method has very low memory requirements, but it is not guaranteed to find the shortest solution path.

Both breadth-first and depth-first search are *exhaustive* algorithms in that they systematically consider all possibilities. They place an organization on the combinatorial explosion, but they do not eliminate it. In contrast, *heuristic search* methods use knowledge of the domain to reduce the combinatorics inherent in the problem space. In some methods, this knowledge takes the form of a numeric evaluation function that determines which state should be expanded.[6] Another approach encodes heuristic knowledge in the conditions placed on operators, giving fewer operator instantiations and causing fewer states to be generated.

We have chosen to use this latter method in ACM. As we have seen, each operator has an associated set of legal conditions that specifies the constraints under which that operator can be applied. For example, in multi-column subtraction one cannot shift attention to the left when one is already in the left-most column. We can further constrain each operator by adding similar conditions that will keep them from generating path hypotheses that seem psychologically implausible. For instance, one might not allow the decrement operator to be applied to the same position twice in succession. The production rules in Table 4 show these search-constraining conditions in brackets. These should not be confused with the acquired heuristic conditions, which are enclosed only by braces.

To see the need for a heuristic search solution of this sort, consider solving a four column problem using the rules shown in Table 4 with only their legal conditions. Since we need one step to write the result for each column and one to shift columns, we cannot generate any solution (correct or otherwise) in fewer than seven steps. Given the generality of the conditions on each rule, we estimate an average of ten instantiations for each state in the problem space.

Carrying out an exhaustive search of this space to a depth of seven would require us to generate $10^8 - 1$ states. If we assume each node can be generated in 1/100th of a second, then this search would take over eleven days to perform. A five column subtraction problem would require two additional steps, increasing the depth to a minimum of nine. This would multiply the number of states generated by 100, requiring over three years to search.

In contrast, the additional conditions we have placed on the operators lead to an average branching factor of three. This lets us consider only $3^8 - 1$ states during a depth seven search, which can be generated in slightly over one minute.[7] But regardless of the number of states involved, we must still organize the search in some fashion. To this end, ACM carries out a depth-first search to a depth one step longer than the length of the *correct* solution path. However, the details of this search differ from the standard depth-first algorithm in two respects.

First, the system rejects all states in which the partial answer diverges from the observed answer. This not only eliminates the current state; it also eliminates all of its successors, reducing the combinatorial explosion. Second, ACM does not halt upon finding a single path hypothesis. Instead, it continues until it has considered all solutions with the specified

[6] Ohlsson and Langley (1986) have described DPF, a diagnostic system that uses numeric evaluation functions to direct the search for path hypotheses. The heuristics of DPF incorporated knowledge of the human information processing system, such as limited short-term memory.

[7] There is a price to be paid for this reduction in search; one may be unable to model certain errors. For example, VanLehn (1982) reports one buggy strategy in which the student decrements a number in the bottom row, which ACM cannot model using the current set of initial conditions.

length or less. In this way, the program finds all path hypotheses that explain the subject's answer on the current problem while requiring very little in the way of memory.

Our depth limit requires some justification. We noted earlier that very few errorful subtraction strategies require more steps than the correct algorithm. One theory of bug origins suggests than errors result from memory limitations during the learning process, and such an explanation would also account for the simple nature of most bugs. But whatever the reason, the vast majority of buggy solution paths are shorter than the correct solution path, and this means we can use the latter's length as a heuristic cutoff in our search.

From Path Hypotheses to Rules

Recall that our ultimate goal is a production system model of the student's behavior on a set of problems. We have already mentioned that ACM takes a problem reduction approach to constructing such models, determining the conditions on each operator independently and combining the resulting rules into the final production system. In finding the conditions for an operator, the system employs a method for learning from examples that requires its input be divided into positive instances and negative instances.

The ACM system constructs these sets using a method that Sleeman, Langley, and Mitchell (1982) have called *learning from solution paths.* Given a solution path to some problem, one labels all instantiations of an operator that lie along the path as positive instances of that operator. Similarly, one labels all instantiations that lead one step off the path as negative instances of the operator.[8] We have already seen how ACM infers path hypotheses for each problem-answer pair, and the system uses these paths to classify each operator instantiation.

Given this information, the program looks for conditions on each operator that cover (match against) the positive instances, but which fail to cover the negative instances. The conditions on each operator let it propose steps that lie along the inferred path hypothesis, but avoid paths that lead off the path. Thus, the resulting rules (when combined) let one replicate the subject's behavior – whether correct or errorful – without the need for search.

ACM uses a particular condition-finding method that is based on Michalski's (1983) A^q algorithm.[9] We can summarize the basic method in three steps. Let *Pos* be the set of positive instances for an operator, *Neg* the set of negative instances, D the empty set, and C the initial conditions on the operator:

1. Randomly select an example p from *Pos*.
2. Determine a set of heuristic conditions H which (when added to C) covers the positive instance p, but which covers none of the instances in *Neg*. Let $C' = C \cup H$.

[8] One ignores states that lie two or more steps off the path, since the system would never have reached that point given the right conditions on its operators.

[9] An earlier version of ACM (Langley & Ohlsson, 1984) used a method for constructing decision trees similar to that described by Quinlan (1983).

3. Remove all instances from *Pos* that are covered by C' and add the conditions C' to the set D. If $Pos = \emptyset$, then halt; otherwise go to step 1.

At the end of this process, D contains a set of condition sets which, taken as a disjunct, covers all of positive instances of the operator and none of the negative instances. ACM then proceeds to simplify each set in D by removing component conditions that do not significantly aid in distinguishing positive from negative instances. The resulting simplified rules may cover some negative instances or fail to cover some positive ones; these cases are interpreted as noisy data.

This outline has left many details unspecified, the most important being the method for generating a new set of conditions from a single positive instance. In order to explain the technique, we must first review ACM's representation of instances, its representation of conditions, and the relation between them. In the subtraction domain, each state is described as a set of relations between numbers, rows, and columns like those shown in Table 2. Thus, the working memory against which rules are matched would contain elements like `(in 3 column-1 row-1)`, `(in 5 column-1 row-2)`, and `(above row-1 row-2)`.

Within the production system framework, the conditions of rules have forms analogous to elements in working memory, but in which some constant terms have been replaced by pattern matching variables. Thus, one set of conditions that would match the elements given above is:

```
((in number-a column row-a)
 (in number-b column row-b)
 (above row-a row-b))
```

where pattern matching variables are shown in italics. This condition set will match a state description in which two numbers (*number-a* and *number-b*) occur in the same column, with *number-a* above *number-b*. Variables can match against any symbol, but they must bind consistently in different conditions.

The positive and negative instances each have two components. The first consists of the working memory elements present in memory when that operator was applied; this represents the overall state description. The second component consists of the variable bindings for the legal conditions that allowed the operator to match. Together with the legal conditions on the operator, this information lets ACM generate conditions for discriminating between positive and negative instances.

This generation process can be viewed in terms of heuristic search. The initial state in this search consists of the operator's legal conditions, while the final state consists of conditions that cover some of the positive instances but none of the negative instances. At each step in the search, ACM transforms all working memory elements in the instance into potential conditions. This involves replacing all constant symbols with the variables to which they were bound in the instantiation. For example, if *row-a* were bound to `row-1` and *row-b* to `row-2`, the element `(above row-1 row-2)` would be transformed into the condition `(above` *row-a row-b*`)`. Those constants not already bound to some variable are replaced with a new variable, except for predicates like `above`, `left-of`, and `in`.

The system also considers two types of conditions that are not based on elements present in the instantiation. User-defined predicates such as **greater**, **equal**, and **zero** lead to conditions relating all numeric variables. Thus, if *number-a* and *number-b* were already mentioned in the existing conditions, ACM would consider conditions such as (**greater** *number-a number-b*) and (**zero** *number-a*). The program also considers negated conditions that must *not* match for an operator to apply, basing them on elements that were present in negative instances of the operator.

The problem is that ACM can generate very many conditions in these ways, and somehow it must select among them. Since its goal is to find a set of conditions that discriminate between positive and negative instances of the operator, the system employs the χ^2 statistic to measure the ability of each competitor in distinguishing between the two classes. Thus, conditions which match many of the positive instances and few of the negative instances will be preferred to those which fail to discriminate.

At each point in its search, ACM selects that condition with the highest score on the χ^2 metric, selecting one at random in case of ties. This condition is added to the set and the element on which it was based is removed from the instantiation so it will not be regenerated. As more conditions are added, fewer and fewer negative instances will be covered, until ultimately all have been eliminated. At this point, the conditions are combined with the operator to form a production rule that is added to the diagnostic model. In summary, ACM employs a form of hill-climbing through the space of conditions, using the χ^2 measure to direct its search.

The system employs the same statistic during the simplification process. After ACM finds a set of conditio: s that covers some positive instances but none of the negatives, it considers the effect of dropping each condition in turn. If the elimination of a condition significantly reduces a production's χ^2 value, then that condition is retained; otherwise it is dropped from condition side of the rule. In this way, the system eliminates extra conditions that were introduced 'accidentally' during the search process.

Combining the Methods

In an earlier version of ACM (Ohlsson & Langley, 1984), the path finding component and the rule finding component worked independently of each other. The system first generated path hypotheses for all of the problems and then used these paths to formulate a production system model that explained them. There were two serious flaws in this approach. First, the amount of search required for twenty subtraction problems of even moderate complexity was prohibitive. Second, the path-finder often generated multiple path hypotheses for each problem, and ACM had no means for determining which explanations were consistent with each other.

The current version of the system takes a new approach that overcomes these limitations. The basic idea involves interleaving the generation of path hypotheses with the formulation of production system models. ACM begins by finding one or more solution paths that explain the subject's answer to the first problem. Based on each path, the system constructs a production system model that predicts that path. However, because these models are based on small amounts of data, they are likely to be nondeterministic. Thus, they will be able to reproduce the path hypotheses, but they will generate other paths as well.

However, the total amount of search is greatly reduced when ACM encounters the next problem-answer pair. This time it uses the partial rules generated during the first round, and each tentative model leads to a second set of path hypotheses. These determine new positive and negative instances, and these in turn let the system refine its models by adding discriminating conditions. The process is repeated on successive problems, gradually giving more detailed models of the student's strategy. The production rules slowly become more specific, and this reduces the amount of search required at each step. If the subject's behavior is sufficiently regular, the model eventually becomes algorithmic and search is eliminated.

One interesting complication can arise within this framework. In some cases, the rules generated from problems 1 through n cannot generate the answer observed for problem $n+1$. The natural interpretation of this situation is that the subject has exhibited a *bug migration* (VanLehn, 1982). That is, he has shifted from one errorful strategy to a different algorithm. The appropriate response is to retain the existing production rules as the model for behavior on the first n problems, but to construct an entirely new model for the remaining problems. This is the approach that ACM takes when its cannot find an adequate path hypothesis.

To illustrate how the combined method operates, let us work through an example diagnosis of a subject with the smaller-from-larger bug. We will use the productions in Table 4 (with only the legal and bracketed conditions) as the initial rules, and we will use the subtraction problems: 647 − 45 = 602, 885 − 297 = 612, and 83 − 44 = 41. The correct answer to the first of these problems requires five steps; when ACM searches to a depth of six, it finds only one path hypothesis to explain the observed answer 602:

```
Find-Difference(7, 5, column1)
Shift-Column(column1, column2)
Find-Difference(4, 4, column2)
Shift-Column(column2, column3)
Find-Top(column3)
```

For the sake of simplicity, we will focus on the **Find-Difference** operator. The inferred path has a single negative instance for this operator: `Find-Difference(5, 7, column1)`; and three positive instances: `Find-Difference(7, 5, column1)` and `Find-Difference(4, 4, column2)`; the last of these occurs twice, once for 4 in both positions.

Based on these instances, ACM finds a set of conditions which covers all the positive instances but not the negative example. However, the lack of data leads the system to reject all of these conditions during the simplification process, so search continues unrestrained on the second problem: 885 − 297 = 612. Again the correct solution takes five steps, so the program searches to a depth of six. Only one solution path accounts for the answer, but this time it differs from the correct solution path:

```
Find-Difference(7, 5, column1)
Shift-Column(column1, column2)
Find-Difference(9, 8, column2)
Shift-Column(column2, column3)
Find-Difference(8, 2, column3)
```

Three positive examples are generated from this path hypothesis: `Find-Difference(7, 5, column1)`, `Find-Difference(9, 8, column2)`, and `Find-Difference(8, 2, column3)`. In addition, three negative examples are produced: `Find-Difference(5, 7, column1)`, `Find-Difference(8, 9, column2)` and `Find-Difference(2, 8, column3)`.

ACM now has seven positive instances of **Find-Difference**, along with three negative instances. This is enough to allow the most discriminating condition to be retained during the simplification process. This condition is (`greater` *number1 number2*), which matches six of the seven positive instances and none of the negative instances. Another likely condition is (`above` *number1 number2*), but this only matches three of the seven positive instances and none of the negative instances, causing it to be dropped during simplification.

With its new **Find-Difference** rule, ACM attempts to solve the next problem: $83-44 = 41$. Because of the additional condition on the new rule, the system generates only two instantiations of **Find-Difference**, both of which lie on the final path hypothesis:

```
Find-Difference(4, 3, column1)
Shift-Column(column1, column2)
Find-Difference(8, 4, column2)
```

This solution path provides two additional positive examples: `Find-Difference(4, 3, column1)` and `Find-Difference(8, 4, column2)`. Provided the subject is consistent on future subtraction problems, the program will only consider instances of the operator that lie along the inferred solution path. In this case, ACM has arrived at a nearly complete model of the subject's behavior after the first two problems. Before generating the complete 'smaller-from-larger' model shown in Table 5, it also requires some problems that use the **Find-Top** operator, but little search occurs on the intervening problems.

This example was simplified in that ACM never found more than one path hypothesis that accounted for the subject's answer on a problem. This in turn effectively eliminated search through the space of alternative production system models. In general, the system may discover multiple solution paths, and when this occurs it constructs a separate production system model based on each path (together with previous paths). ACM then examines each model's ability to explain the current path and all previous paths. This involves running the model on earlier problems to see how many positive and negative instances it generates for each operator. The χ^2 measure is also used in this evaluation process, with the highest-scoring model being retained, along with the path hypothesis on which it was based.

Advantages of the Approach

The approach we have taken to automating cognitive diagnosis has a number of benefits. For instance, the method's reliance on the χ^2 measure lets it tolerate noise. The learned conditions on an operator need not cover all of its positive instances, nor must they mismatch all negative instances. As long as a condition accounts for significant regularities in the data, it is retained in the resulting production system model.

The method can also formulate models in which an operator has disjunctive conditions. The χ^2 statistic prefers condition sets that cover more of the positive instances, and thus prefers conjunctive rules when these cover the data. But when two or more sets of conditions are necessary, ACM has no difficulty in generating such disjuncts. These are represented as separate rules in the final production system.

In addition, the system detects bug migrations when the model produced from the first n problems cannot solve the $n + 1$st problem. Earlier versions of ACM (Ohlsson & Langley, 1984) did not have this ability. Schlimmer and Granger (1986) have identified a closely related issue in learning from examples, which they term *concept drift*. This task involves distinguishing between noisy data and changing environments, and they argue that this is a very difficult problem. We will not claim to have a complete solution to such ambiguities, but we have made a start.

In fact, the above three problems remain basic research issues in the field of machine learning. Few AI learning systems can acquire disjunctive rules and even fewer can learn from noisy data. Very little work has examined the problem of concepts which change over time, which corresponds to our bug migration. Although our main goal is to automate the process of cognitive diagnosis, ACM can also be viewed as responding to the joint issues of disjuncts, noise, and concept change.

A final advantage of the approach is that it measures the quality of the final production system model. After all problems have been solved, ACM computes the χ^2 score for the final model in terms of its ability to predict the positive and negative instances of each operator. The greater the number of steps along the inferred path hypotheses that are predicted by the model, the higher its descriptive power. Although Newell and Simon (1972) proposed methods for evaluating their hand-crafted models, their approach was not linked to standard statistical measures.

We have implemented ACM in Interlisp-D on Xerox Lisp Machine and we have successfully applied it to some common subtraction bugs. Recall that Table 1 lists the eleven most frequent subtraction bugs reported by VanLehn (1982). For each bug, we provided the system with twenty test problems and the answers that would result from an idealized application of that strategy. Given these data, ACM generated plausible production system models to account for all eleven buggy strategies (including the correct strategy).

For example, one of the bugs listed in table 1 is 'borrow-from-zero' in which the subject needing to borrow from zero changes the zero to a nine instead of continuing left to borrow. The model ACM formed for the buggy procedure is identical to the model it formed for the correct procedure except for the **Shift-Left** and **Add-Ten** operators. In the correct procedure, there are two rules for shifting left; **Shift-Left-To-Borrow** and **Shift-Left-Across-Zero**. The model that ACM formed for the 'borrow-from-zero' bug includes the version of the **Shift-Left-To-Borrow** rule it had formed for the correct procedure, but excludes the **Shift-Left-Across-Zero** rule entirely. This second rule is not needed to model the 'borrow-from-zero' bug since when the subject is confronted with borrowing from zero, he simply changes the zero to a nine and has no need to continue shifting left. For the **Add-Ten** operator, ACM forms the version of the rule used for the correct procedure, as well as the following (paraphrased) rule: *if you are focused on a column in which the top number is zero, and you have just shifted left, then change the zero to a ten.* Whenever this rule is used, the conditions for the **Decrement** rule are subsequently met and the ten is then decremented to a nine and the processing continues as in the correct procedure. Together these rules form a production system that correctly models the buggy procedure 'borrow-from-zero'.

We have also tested the system on idealized subtraction data containing bug migrations. For the same twenty problems, we provided ACM with the correct answers to the first ten problems, but with incorrect answers to the last ten problems that would be produced by the 'smaller-from-larger' strategy. The system detected when the bug migration occurred and produced two separate production system models: one for the first ten correct answers, and another for the last ten buggy answers. We have not yet tested ACM on actual subject data in the subtraction domain. However, earlier work suggests that such data will contain both noise and migrations, and such tests are one of our highest research priorities.

Rules and Principles in Cognitive Diagnosis

All existing work on automated cognitive diagnosis, including the approach we have described in the preceding pages, builds on some procedural analysis of the skill being diagnosed and results in some description of the subject's procedure or strategy. Before closing, we should consider an alternative framework for diagnosis that may be more relevant in the long run, at least for educational purposes.

Bugs and Misconceptions

As the field of cognitive psychology has progressed, our understanding of errorful behavior has become more sophisticated. Researchers began with the simple distinction between correct and incorrect performance as measured by tests. This simplistic view has been superceded by the distinction between forgetting errors (due to memory limitations) and procedural errors (due to faulty strategies). Even more recently, researchers have started to distinguish different types of procedural errors. Brown and Burton's (1978) work on DEBUGGY was instrumental in clarifying the difference between systematic errors (bugs) and those caused by carelessness.

In this section, we would like to propose a further refinement within the set of systematic errors, namely between *bugs* and *misconceptions.* A bug is a syntactic entity, involving some fault in a procedure which can be corrected by a set of editing operations on the code of the procedure, such as adding or replacing conditions. In other words, a bug is inherently procedural in nature. A misconception, on the other hand, involves a person's beliefs about the world. These are more declarative in nature, though they must interact with problem solving processes to impact behavior. Misconceptions imply faulty *understanding* rather than faulty *performance.*

Procedural bugs can be modeled by clearly defined programming languages, and thus lend themselves to formal analysis. Unfortunately, we have no formalisms analogous to programming languages that would make possible a formal treatment of misconceptions. In order to proceed, we will follow the lead of several other researchers interested in understanding and assume that understanding resides in *principles*, i.e., propositions of general validity within a task domain. This view of understanding has virtually no basis, except that principles are commonly used within well-understood domains such as mathematics and natural science.[10] However, we currently have no viable alternative. Given this model of understanding, it seems natural to model misunderstanding as involving the *violation* of some principle or, equivalently, the use of an incorrect principle.

In a task domain like arithmetic, one would expect some relationship between understanding and performance, between misconceptions and bugs, between principles and heuristics. Resnick (1982) has explicitly argued that subtraction bugs can be seen as corresponding to violations of specific subtraction principles. Indeed, it seems natural to hypothesize procedural bugs as being *caused* by misconceptions. In other words, the process of acquiring procedures results in buggy algorithms due to a lack of constraints (or due to wrong constraints), and these follow from a lack of understanding.[11]

From a pedagogical point of view, misconceptions are more central than bugs. Not only will faulty understanding cause bugs, but in a hierarchically organized subject matter like mathematics, faulty understanding at one level is likely to create difficulties at the next level of learning. Indeed, one could argue that the schools' main goal should be to communicate understanding, and that faulty understanding indicates failure even in the presence of correct performance. Even if one rejects this stance, it seems clear that remedial teaching should be directed towards misconceptions rather than bugs whenever possible. This in turn raises the problem of *diagnosing misconceptions:* given a set of observed answers to problems in some domain, and given a set of principles which encode understanding of that domain, identify a set of misconceptions (violated principles) which explain the observed answers.

In considering this problem, we should naturally attempt to build upon existing results from cognitive diagnosis. If possible, we should adapt earlier techniques to the new goal of finding misconceptions in place of buggy heuristics. Below we consider two approaches to diagnosing misconceptions, both of which build on the notion of path hypotheses and the problem space hypothesis. Thus, these methods share features with the ACM system described earlier, even though they make no attempt to construct process models to describe the inferred paths.

Principles as Evaluation Functions

We assume a set **P** of principles $P_1, P_2, \ldots, P_n$ for a task domain **T**. We further assume a set of problems $T_1, T_2, \ldots, T_m$, to which some person has produced a set of answers $A_1, A_2, \ldots, A_m$; the person can be viewed as a function **a** which maps problems to answers,

[10] See Rissland (1978) for a view of mathematical knowledge and understanding which adds considerable richness to the notion of a 'principle'.

[11] Resnick and Omanson (in press) have reported experimental evidence against this hypothesis, but we do not feel their results are sufficient to justify abandoning the view at this point.

$A_i = \mathbf{a}(T_i)$. As before, we will incorporate the notion of a problem space and path hypotheses, so A_i is (strictly speaking) a *path*. We assume that an answer is generated by some procedure, strategy, or collection of heuristics which traverses the states in some problem space, thus generating a path from the initial state in the space to the goal state.

The central idea is that principles are related to paths, rather than to answers or procedures. We will argue that *solution paths are best viewed as obeying or violating the principles of the task domain* in which behavior occurs. As an example, consider the general arithmetic principle of *compensation* as proposed by Resnick (1982). This principle states that when a number is re-decomposed into additive components, its value remains constant only when a quantity that has been subtracted from one component is also added to another component. Suppose that we observe the solution of a subtraction problem in which there are decrements that are not 'paid back', or in which increments are not preceded by any borrowing operation. Any solution path in the standard subtraction space that has unequal numbers of applications of the **Add-Ten** and **Decrement** operations will thus violate the principle of compensation.

This framework suggests that *all* principles of a domain can be viewed as *path constraints* (Carbonell, 1986), and this suggests our first method for diagnosing misconceptions. The basic idea is *to use the number of constraint violations as the evaluation function for heuristic search through the problem space.* In other words, we start with some problem space (such as the standard subtraction space) and with a set of principles, and we formulate the principles in terms of constraints on paths through that space. We then carry out a best-first search through the space, always selecting for expansion that node with the lowest score. We measure this score by the number of principle violations occurring along the path to that node, rejecting paths which do not lead to the same digits in the same columns as the observed answer.[12] This procedure will find the path through the problem space which accounts for the observed answer using the assumption of minimal number of misconceptions.

This method for diagnosing misconceptions in terms of principle violations has a number of interesting features:

- If we use the A* search algorithm, the method will discover the best possible diagnosis; this follows from properties of the A* algorithm.
- The diagnosis contains not only the number of principle violations, but also a list of *which* misconceptions are implied by the path. From this we can hypothesize that the subject fails to understand these principles.

[12] The A* algorithm for best-first search also requires some estimate of the cost of the remaining path. One simple cost estimate for the subtraction domain is the number of columns that still have no result. Since each column takes at least one operator application to generate a result, this measure is guaranteed to underestimate almost any cost function, as the A* algorithm requires to guarantee optimal results. Another issue is the monotonicity of the cost function. If we formulate the path constraints so that each violation is associated with a single step, then monotonicity is satisfied.

- The method bypasses the need for a procedure to generate the correct path. Neither the correct path nor the correct procedure play any role in this technique.
- The method diagnoses the misconceptions *without* diagnosing the buggy procedure. The method matches *paths* against principles, thus avoiding the need to infer the subject's strategy.

This diagnostic method is closely related to our earlier work on the Diagnostic Path-Finder (Ohlsson & Langley, 1986). In the DPF system, the best-first search was also guided by principles. However, DPF employed general psychological principles to evaluate the plausibility of any one path hypothesis. In effect, our new method applies the same computational machinery, but with a very different interpretation. The principles now represent the content of the domain, rather than general theoretical principles based on our knowledge of the human information processing system. The psychological theory embedded in DPF has been replaced with the single assumption that the path which violates the least number of domain principles is the most plausible one.[13]

Like all best-first search methods, the new method is only guaranteed to find the optimal path if each node in the state space is expanded completely. This means that all its (one-step) descendents must be explicitly generated (and evaluated). In spaces with large branching factors, this means that even a very selective search (in the sense that only a few non-profitable paths are explored) will nevertheless generate a very large number of states. This follows from the fact that evaluation functions provide selectivity only *after* a node has been expanded and its successors generated.

In general, this problem can be overcome by introducing selectivity during the process of *generating* new nodes. In the current context, this means using domain principles to constrain the branches leading out from a search node. In other words, we would like to find a way to move the selective power of the principles from the node-selection stage to the step-generation stage. Below we outline an alternative method for diagnosing misconceptions that incorporates this idea.

Principles as Condition Generators

The use of principles as path constraints arose from the idea that misconceptions can be diagnosed separately from the buggy procedure which generated them. However, this approach ignores the relation between principles and problem solving rules. We will see that one can also use principles to *generate* such rules, and that one can use these rules to propose steps through the problem space which violate some specific principle. Instead of first generating all logically possible states and then using the principles to evaluate them, we can use rules based on the principles to selectively generate steps which violate those principles. This method involves three successive stages: constructing rules for correct

[13] Note that this scheme could also be used as a module in the current ACM system to find solution paths. This diagnostic method does not *forbid* production system models, it simply does not *rely* on them.

performance, generating rules that produce specific misconceptions, and using these rules to perform cognitive diagnosis.

First, let us consider the relation between principles and correct performance. The total set of principles for a domain like subtraction should dictate exactly which steps to take during problem solving. This suggests a technique for systematically generating rules for correct performance in a domain. For each operator in the problem space, create a rule which applies that operator. For each principle, add to the rule *those conditions which ensure that the application of the operator does not violate the principle.* If we do this for each principle, then the operator will only be applied when its action obeys all the principles. If we carry out this scheme for each operator, the resulting set of productions will perform in accordance with the given principles, producing correct behavior.

The importance of such problem solving rules is *not* that they constitute a plausible model of human performance. Humans do not acquire procedures using the method just described, and we doubt the resulting production system would be a good cognitive model. By the same reasoning, one should not attempt to teach such rules to students. The importance of such rules is that we can use them as a step generator with known properties. Recall that each condition in these rules is motivated by some particular domain principle. From this it follows that *removing a condition from a given rule produces a rule which violates the corresponding principle.* In other words, since we know which condition corresponds to which principle, we can generate variants on the correct rules which will generate steps that violate specific principles.

For instance, imagine a correct rule which includes a condition that the first argument of the **Find-Difference** operator be *above* the second argument in the problem display. This condition is motivated by the principle which says that the purpose of subtraction is to subtract the subtrahend from the minuend, rather than vice versa. Removing this condition leads to a rule which generates steps violating this principle; the surface symptom of this is the 'smaller-from-larger' bug we described earlier.

In short, we start with the set of rules for correct performance. For each rule, we create one variant for each principle by deleting those conditions which ensure that the rule's actions obey that principle. The resulting set of rules is a step-generator which will *not* explore every path in the problem space, but *only those paths which can be interpreted in terms of violations of the stated principles.* The functional or operational branching factor in this space is independent of the logical branching factor. Instead, it is dependent upon the number of principles and the different ways in which they can be violated.[14]

Since our goal is diagnosis, we must search the reduced problem space in order to find a path explaining the observed answer, using the new rules as step generators. What kind of search is appropriate? Again, best-first search seems the right paradigm, with the number of rule violations as the evaluation function. However, expansion of a node no longer involves the exhaustive generation of all logically possible descendents, but only those generated by

[14] Recall that in order to reduce ACM's search for path hypotheses to a reasonable size, we were forced to add conditions to our basic set of rules. The above method suggests a more 'principled' response to this problem.

one or more of the rules. Also, there is no need to actually compare principles with the paths because we know which rule violates which principle. Therefore, one can determine which principles have been violated by inspecting the path itself.

Moving the principles into the step-generator allows us to consider only that portion of the total problem space which involves misconceptions. Other (uninterpretable) faulty paths will not be visited during search. Moreover, the method contains a criterion of consistency with respect to misconception: we can analyze a path in order to see whether a principle which was violated in one state was violated in every possible state. The approach also tells us when a diagnosis is ambiguous with respect to misconceptions. If a problem solving step was generated by two different rules, then either of two misunderstood principles could be responsible for the problem. Thus, we can determine all interpretations of a student's errors. If the search through the space terminates without finding a path that explains the observed answer, then we know that the answer cannot be explained in terms of the given principles.

Summary

Cognitive simulation provides a framework for describing the complexities of human cognition, but this methodology (Newell & Simon, 1972) requires major effort in both the data-collection stage and the model-building stage. Initial attempts to automate the process of cognitive diagnosis have proved quite successful in limited domains such as subtraction (Brown & Burton, 1978) and algebra (Sleeman & Smith, 1981). However, these approaches relied on hand-crafted 'bug libraries', which in turn required extensive analysis of subjects' errors in the domain.

In this paper, we described an alterative approach to automated cognitive diagnosis that avoids the need for bug libraries. The method borrows two assumptions from other work in cognitive simulation – the problem space hypothesis and the production system hypothesis. One inputs some problem space for the domain by specifying the state representation and the operators for generating new states. One also provides a set of test problems and a subject's answers to those problems. Given this information, the method automatically infers path hypotheses that explain each answer, and then induces a production system model that explains the inferred paths.

This paradigm requires significantly less domain analysis than the bug library approach, and the method should be applicable to many procedural domains. We have implemented the technique as a running AI system called ACM, and we have tested this system on idealized buggy behavior in the subtraction domain. In addition, the particular methods it uses to infer paths, to induce rules, and to formulate production system models promise to handle three major issues that arise in cognitive diagnosis – disjunctive rules, noisy behavior, and bug migration. Our preliminary tests of ACM along these dimensions have been encouraging.

However, we need more serious tests of the system's capabilities, and this is a major direction for future work. First we must determine whether the program can model the majority of the observed subtraction bugs given idealized data. The next step is to provide ACM with actual subtraction data; this will further test the system's ability to handle noise and migrations. Finally, we hope to use the same system to model behavior in two additional domains - arithmetic with fractions and simple algebra. This will mean providing ACM with new problem spaces, but we hope that few modifications of the system itself will be necessary.

An entirely different research path would explore the role of principles in cognitive diagnosis. Rather than explaining subject's errors in terms of an algorithmic production system model, one can summarize behavior in terms of misconceptions about the domain. In this framework, one still searches for path hypotheses that explain the subject's answers, but one prefers those paths which violate fewer principles of the domain. The resulting paths identify which principles the subject has misunderstood, and this has more direct implications for remedial instruction than the errorful rules induced by ACM. We plan to follow this approach in parallel with our extensions of the rule-based framework.

We have no illusions about the difficulty of fully automating the task of cognitive diagnosis. This would force us to replicate the entire range of scientific reasoning in cognitive psychology, and we have no such pretensions. However, we believe that major components of the diagnostic process - the formulation of path hypotheses and the generation of production system models - *can* be automated using existing techniques from artificial intelligence and machine learning. We envision the day when psychologists interact with such a partially automated system to construct detailed models of a subject's behavior, and when intelligent teaching systems invoke similar modules to construct accurate student models. We will not predict the date when diagnostic systems become robust enough for these purposes, but we hope our efforts with ACM will speed its arrival.

References

Anderson, J. A., & Hinton, G. E. (1981). Models of information processing in the brain. In G. E. Hinton & J. A. Anderson (Eds.). *Parallel models of associative memory.* Hillsdale, NJ: Erlbaum.

Anderson, J. R. (1976). *Language, memory, and thought.* Hillsdale, NJ: Erlbaum.

Anderson, J. R. (1983). *The architecture of cognition.* Cambridge, MA: Harvard University Press.

Anderson, J. R. (1984). Cognitive principles in the design of computer tutors. *Proceedings of the Sixth Conference of the Cognitive Science Society*, 2-9.

Anderson, J. R., & Bower, G. H. (1973). *Human associative memory.* Washington, DC: Winston.

Anzai, Y., & Simon, H. A. (1979). The theory of learning by doing. *Psychological Review*, *86*, 124-140.

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

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

Burton, R. (1982). Diagnosing bugs in a simple procedural skill. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 157–183). London: Academic Press.

Carbonell, J. G. (1986). Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (Vol. 2, pp. 371–392). Los Altos, CA: Morgan Kaufmann.

Ericsson, K. A., & Simon, H. A. (1984). *Protocol analysis: Verbal reports as data.* Cambridge, MA: MIT Press.

Hagert, G. (1982). On procedural learning and its relation to memory and attention. *Proceedings of the European Conference on Artificial Intelligence*, 261–266.

Hayes, J. R., & Simon, H. A. (1974). Understanding written problem instructions. In L. W. Gregg (Ed.), *Knowledge and cognition.* Potomac, MD: Erlbaum.

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

Langley, P., & Ohlsson, S. (1984). Automated cognitive modeling. *Proceedings of the National Conference on Artificial Intelligence*, 193–197.

Langley, P., Ohlsson, S., & Sage, S. (1984). *A machine learning approach to student modeling* (Tech. Rep. No. CMU-RI-TR-84-7). Pittsburgh, PA: Carnegie-Mellon University, Robotics Institute.

Lewis, C. (1986). Composition of productions. In D. Klahr, P. Langley & R. Neches (Eds.), *Production system models of learning and development.* Cambridge, MA: MIT Press.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (pp. 83–134). Palo Alto, CA: Tioga Press.

Mitchell, T. M. (1982). Generalization as search. *Artificial Intelligence, 18*, 203–226.

Mitchell, T. M., Utgoff, P.E., & Banerji, R. B. (1983). Learning by experimentation: Acquiring and refining problem -solving heuristics. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (pp. 163–190). Palo Alto, CA: Tioga Press.

Neches, R. (1981). A computational formalism for heuristic procedure modification. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 283–288.

Newell, A. (1967). *Studies in problem solving: Subject 3 on the crypt-arithmetic task Donald + Gerald = Robert* (Technical Report). Pittsburgh, PA: Carnegie Institute of Technology, Center for the Study of Information Processing.

Newell, A. (1972). A theoretical exploration of mechanisms for coding the stimulus. In A. W. Melton & E. Martin (Eds.), *Coding processes in human memory.* Washington, Winston.

Newell, A. (1973). Production systems: Models of control structures. In W. G. Chase (Ed.), *Visual information processing.* New York: Academic Press.

Newell, A. (1980). Reasoning, problem solving, and decision processes: The problem space hypothesis. In R. Nickerson (Ed.), *Attention and performance VIII.* Hillsdale, NJ: Erlbaum.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice–Hall.

Newell, A., Shaw, J., & Simon, H. A. (1960). Report on a general problem-solving program for a computer. *Proceedings of the International Conference on Information Processing,* UNESCO, Paris.

Nilsson, N. J. (1980). *Principles of artificial intelligence.* Palo Alto, CA: Tioga Press.

Ohlsson, S. (1980). *Competence and strategy in reasoning with common spatial concepts: A study of problem solving in a semantically rich domain.* PhD thesis, Department of Psychology, University of Stockholm.

Ohlsson, S. (1983). A constrained mechanism for procedural learning. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* 426–428.

Ohlsson, S., & Langley, P. (1984). Towards automatic discovery of simulation models. *Proceedings of the European Conference on Artificial Intelligence.*

Ohlsson, S., & Langley, P. (1986). Psychological evaluation of path hypotheses in cognitive diagnosis. In H. Mandl & A. Lesgold (Eds.), *Learning issues for intelligent tutoring systems.* New York: Springer.

Pearl, J. (1984). *Heuristics.* Reading, MA: Addison–Wesley.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application to chess end games. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (pp. 463–482). Palo Alto, CA: Tioga Press.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1,* 81–106.

Resnick, L. B. (1982). Syntax and semantics in learning to subtract. In T. P. Carpenter, J. M. Moser & T. A. Romberg (Eds.), *Addition and subtraction: A cognitive perspective.* Hillsdale, NJ: Erlbaum.

Resnick, L. B., & Omanson, S. F. (In press). Learning to understand arithmetic. In R. Glaser (Ed.), *Advances in instructional psychology* (Vol. 3). Hillsdale, NJ: Erlbaum.

Rissland, E. L. (1978). Understanding understanding mathematics. *Cognitive Science, 2*(4), 361–383.

Rosenbloom, P. (1983). *The chunking model of goal hierarchies: A model of practice and stimulus-response compatibility.* Doctoral dissertation, Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Schlimmer, J. C., & Granger, R. H. (1986). Incremental learning from noisy data. *Machine Learning, 1,* 317–354.

Sleeman, D. (1983). Inferring student models for intelligent computer-aided instruction. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach* (pp. 483-510). Palo Alto, CA: Tioga Press.

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

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

Smedslund, J. (1969). Psychological diagnostics. *Psychological Bulletin*, *71*, 237-248.

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

VanLehn, K. (1983). Human procedural skill acquisition: Theory, model and psychological validation. *Proceedings of the National Conference on Artificial Intelligence*, 420-423.

Waterman, D. A., & Newell, A. (1972). *Preliminary results with a system for automatic protocol analysis* (CIP Report No. 211). Pittsburgh, PA: Carnegie-Mellon University, Department of Psychology.

Williams, M. D., & Hollan, J. D. (1981). The process of retrieval from very long-term memory. *Cognitive Science*, *5*, 87-119.

Young, R. M. (1976). *Seriation by children: An artificial intelligence analysis of a piagetian task.* Basel, Switzerland: Birkhauser.

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