COGNITIVE SCIENCE, 2, 155-192 (1978)

Diagnostic Models for Procedural Bugs in Basic Mathematical Skills*

JOHN SEELY BROWN AND RICHARD R. BURTON

Bolt Beranek and Newman

A new diagnostic modeling system for automatically synthesizing a deep-structure model of a student's misconceptions or bugs in his basic mathematical skills provides a mechanism for explaining *why* a student is making a mistake as opposed to simply identifying the mistake. This report is divided into four sections: The first provides examples of the problems that must be handled by a diagnostic model. It then introduces *procedural networks* as a general framework for representing the knowledge underlying a skill. The challenge in designing this representation is to find one that facilitates the discovery of misconceptions or bugs existing in a particular student's encoding of this knowledge. The second section discusses some of the pedagogical issues that have emerged from the use of diagnostic models within an instructional system. This discussion is framed in the context of a computer-based tutoring/gaming system developed to teach students and student teachers how to diagnose bugs strategically as well as how to provide a better understanding of the underlying structure of arithmetic skills. The third section describes our uses of an executable network as a tool for *automatically* diagnosing student behavior, for automatically generating "diagnostic" tests, and for judging the diagnostic quality of a given exam. Included in this section is a discussion of the success of this system in diagnosing 1300 school students from a data base of 20,000 test items. The last section discusses future research directions.

> If you can both listen to children and accept their answers not as things to just be judged right or wrong but as pieces of information which may reveal what the child is thinking you will have taken a giant step toward becoming a master teacher rather than merely a disseminator of information.
>
> J. A. EASLEY, JR. & R. E. ZWOYER (1975)

INTRODUCTION

One of the greatest talents of teachers is their ability to synthesize an accurate "picture," or model, of a student's misconceptions from the meager evidence

*This paper is a substantially expanded version of a paper written with Kathy M. Larkin entitled "Representing and Using Procedural Bugs for Educational Purposes," which appears in *Proceedings of the ACM*, National Conference, ACM 77, October 1977.

Development of the general framework of Diagnostic Models which underlies this research was supported, in part, by the Advanced Research Projects Agency, Air Force Human Resources Laboratory, Army Research Institute for Behavioral and Social Sciences, and Navy Personnel Research and Development Center under Contract No. MDA903-76-C-0108.

Requests for reprints should be addressed to Richard R. Burton, Bolt Beranek and Newman Inc., 50 Moulton Street, Cambridge, Mass. 02138.

inherent in his errors. A detailed model of a student's knowledge, including his misconceptions, is a prerequisite to successful remediation. The structure, use, and *inference* of such models for procedural skills in mathematics is the topic of this paper. In particular we shall describe some initial efforts in the development and use of a representational technique called "procedural networks" as the framework for constructing *diagnostic models*—i.e., models that capture a student's common misconceptions or faulty behavior as simple changes to (or mistakes in) a correct model of the underlying knowledge base. By being able to synthesize such deep-structure diagnostic models *automatically*, we can provide both a teacher and an instructional system with not only an identification of *what* mistakes a student is making, but also an explanation of *why* those mistakes are being made. Such a system also has profound implications for testing, since a student need no longer be evaluated solely on the number of errors appearing on his test, but rather on the fundamental misconceptions which he harbors.

This paper consists of four sections. The first provides examples of the problems that must be handled by a diagnostic model. It then introduces procedural networks as a general framework for representing the knowledge underlying a skill. The challenge here is to design a representation that facilitates the discovery of misconceptions or bugs existing in a particular student's encoding of this knowledge. The second section discusses some of the pedagogical issues that have emerged from our use of diagnostic models within an instructional system. This system is a computer-based tutoring game developed to teach both students and student teachers about the strategic diagnosis of bugs. The third section describes our uses of procedural network as a tool for automatically diagnosing student behavior, for automatically generating "diagnostic" tests, and for judging the diagnostic quality of a given exam. Included in this section is a discussion of the success of this system in diagnosing 1300 grade-school students from a data base of 20,000 test items. The last section discusses some future research directions.

1. DIAGNOSTIC MODELS OF BASIC SKILLS

The issues addressed in this paper arose from an investigation of the procedural skills necessary to solve high school algebra problems. These skills include not only the generally recognized rules of algebra, but also such normally implicit skills as the reading of formulas, the parsing of expressions, and the determination of which rules to apply next (Brown & Burton, 1975; Brown, Collins, & Harris, 1978; Matz, 1978). For this paper, however, we limit our discussion to examples encompassing arithmetic skills so that we can concentrate on the critical ideas of diagnosis without the need for a large number of algebraic rules. Limiting our examples to arithmetic also provides a compelling demonstration of the difficulty of the diagnostic task; it clearly demonstrates how much more difficult it is to diagnose what is wrong with a student's method of perform-

ing a task—to form a diagnostic model—than it is to perform the task itself. In particular, it is no great challenge to add or subtract two numbers, but, as we shall see, diagnosing misconceptions in these same skills can be quite subtle.

Let us consider diagnosing what is wrong with the arithmetic skills (procedures) of a couple of students. We shall start with a case study in which we examine five "snapshots" of a student's performance doing addition as might be seen on a homework assignment. Before proceeding, look at the following snapshots and try to discover the student's bug.

Sample of the student's work:

41	328	989	66	216
+9	+917	+52	+887	+13
50	1345	1141	1053	229

Once you have discovered the bug, try testing your hypothesis by "simulating" the buggy student so as to predict his results on the following two test problems.

446	201
+815	+399

The bug is really quite simple. In computer terms, the student, after determining the carry, forgets to reset the "carry register" to zero and hence the amount carried is accumulated across the columns. For example, in the student's second problem (328 + 917 equals 1345) he proceeds as follows: 8+7=15 so he writes 5 and carries 1; 2+1=3 plus the 1 carry is 4; lastly, 3+9=12 but that 1 carry from the first column is still there—it has not been reset—so adding it in to this column gives 13. If this is the bug, then the answers to the test problems will be 1361 and 700. This bug is not so absurd when one considers that a child might use his fingers to remember the carry and forget to bend back his fingers, or counters, after each carry is added.

A common assumption among teachers is that students do not follow procedures very well and that erratic behavior is the primary cause of a student's inability to perform each step correctly. Our experience has been that students are remarkably competent procedure followers, but that they often follow *the wrong procedures*. One case encountered last year is of special interest. The student proceeded through a good portion of the school year with his teacher thinking that he was exhibiting random behavior in his arithmetic performance. As far as the teacher was concerned there was no systematic explanation for his errors. Here is a sample of his work:

7	9	8	6	8	9	17	19	87
+8	+5	+3	+7	+8	+9	+8	+4	+93
15	14	11	13	16	18	25	23	11

365	679	923	27,493	797
+574	+794	+481	+1,509	+48,632
819	111	114	28,991	48,119

There is a clue to the nature of his bug in the number of ones in his answers. Every time the addition of a column involves a carry, a 1 mysteriously appears in that column; he is simply writing down the carry digit and forgetting about the units digit! One might be misled by 17+8 which normally involves a carry yet is added correctly. It would seem that he is able to do simple additions by a completely different procedure—possibly by counting up from the larger number on his fingers.

The manifestation of this student's simple bug carries over to other types of problems which involve addition as a subskill. What answer would he give for the following?

> A family has traveled 2975 miles on a tour of the U.S. They have 1828 miles to go. How many miles will they have traveled at the end of their tour?

He correctly solved the word problem to obtain the addition problem 2975 + 1875, to which he answered 3191. Since his work was done on a scratch sheet, the teacher only saw the answer which is, of course, wrong. As a result, the teacher assumed that he had trouble with word problems as well as with arithmetic.

When we studied this same student's work in other arithmetic procedures, we discovered a recurrence of the same bug. Here is a sample of his work in multiplication:

68	734	543	758	2764
×46	×37	×206	×296	×53
24	792	141	144	2731

Several bugs are manifested here, the most severe one being that his multiplication algorithm mimics the column behavior of his addition algorithm. But notice that the bug in his addition algorithm is also present in his multiplication procedure. The "determine-unit-and-carry" subprocedure bug shows up in both his multiplication and addition. For example, to do 68×46, in the first column he performs 8×6, gets 48 and then writes down the "carry," in this case 4, ignoring the units digit. Then, he multiplies 6×4 to get 2 for the second column. All along he has a complete and consistent proredure for doing arithmetic. His answers throughout all of his arithmetic work are far from random; in fact, they display near perfection with respect to *his* way of doing it.

A First Approximation to Representing Procedural Skills

For a computer system to be capable of diagnosing aberrant behavior such as the above, the procedural skill being taught must be represented in a form amenable to modeling *incorrect* as well as correct subprocedures of that skill. Furthermore, the representation must allow the intermixing of both the correct

and incorrect subskills, so that the model can capture those parts of a skill that are correct as well as those that are wrong. The breakdown of the skill into shared subskills can also account for the recurrence of similar errors in different skills. We introduce the term *diagnostic model* to mean a representation of a student's procedural knowledge or skill that depicts his internalization of a skill as a variant of a correct version of that skill.

In addition to satisfactory representational techniques, the diagnostic modeling task requires that the representation of a particular correct skill make explicit much of the tacit knowledge underlying the skill. In particular, the correct model must contain *all* of the knowledge that can possibly be misunderstood by the student or else some student misconceptions will be beyond the diagnostic modeling capabilities of the system. For example, if the model of addition does not include the transcription of the problem, the system would never be able to diagnose a student whose bug is to write 9s that he later misreads as 7s.

The technique we use to represent diagnostic models is a *procedural network*.[1] A procedural network model for a correct skill consists of a collection of procedures with annotations in which the control structure (i.e., calling relationships) between procedures are made explicit by appropriate links. Each procedure node has two main parts: a conceptual part representing the intent of the procedure, and an operational part consisting of methods for carrying out that intent. The methods (also called implementations) are programs that define how the results of other procedures are combined to satisfy the intent of a particular procedure.[2] Any procedure can have more than one implementation, thus providing a way to model *different methods* for performing the same skill. For most skills, the network representation takes the form of a lattice. Figure 1 presents a partial breakdown of a portion of the addition process into a procedural network. Conceptual procedure nodes are enclosed in ellipses. The top procedure in the lattice is addition.[3] Two of the possible algorithms for doing addition are presented as

[1]This term has been used by Earl Sacerdoti (1977) to describe an interesting modeling technique for a partially ordered sequence of annotated steps in a problem solving "plan" as well as for specifying control information. Our use of procedural nets coincides with his on this latter feature but differs from and is less developed than his with regard to "plans."

[2]The language we have used to define these programs is LISP. The particular programming language is unimportant from a theoretical standpoint because an implementation is nonintrospectable. The modeling aspects of the network must occur at the conceptual procedure level. For example, the implementation of the subtraction facts table look-up procedure in the computer is necessarily different from that in the student. However, the conceptual properties of the two implementations can be made to agree to an "appropriate" level of detail. Those aspects which are appropriate for our task—the invoking of other procedures, the values returned, the relevant side effects—are included in the network, while the implementation details that may differ are "swept under the rug" into the program. This distinction between conceptual and implementation details also allows skills to be modeled efficiently at different levels which may be appropriate for different tasks.

[3]This simplified representation demonstrates only those features of the procedural network particularly relevant to the diagnostic task. The actual breakdown into subprocedures may be different in a particular network and will be considerably more detailed.

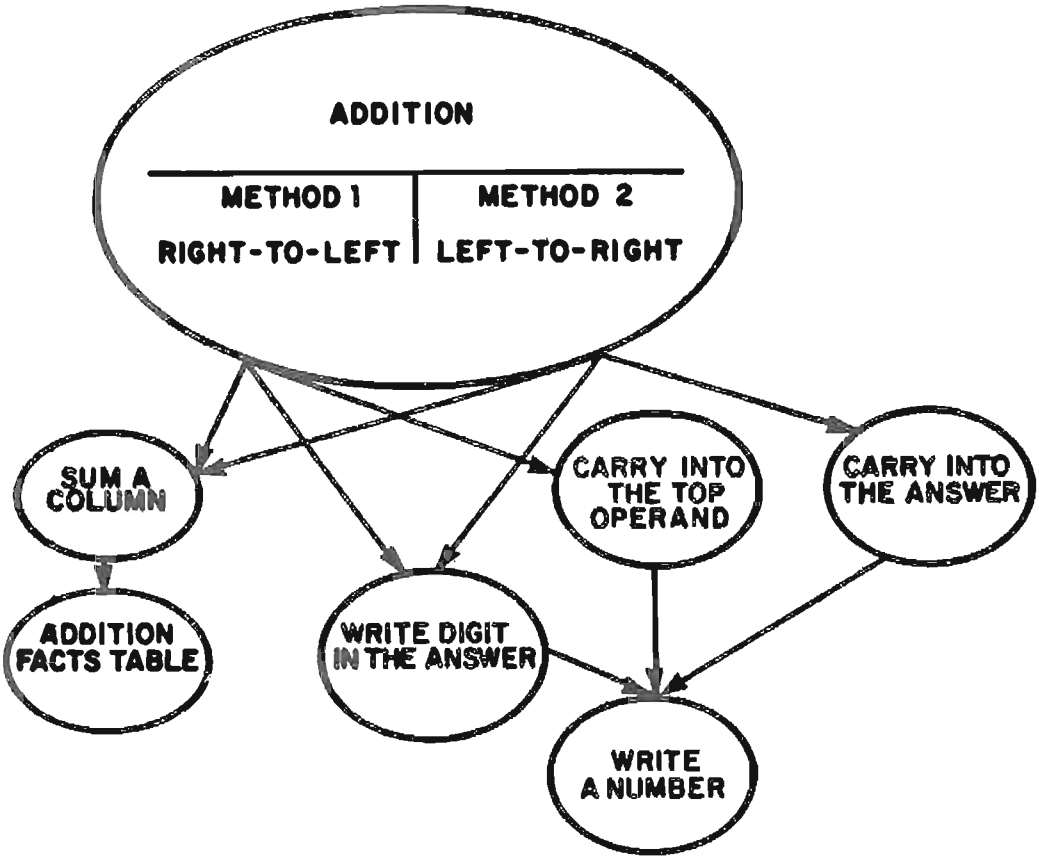

FIG. 1 A portion of the procedural network for addition.

alternative methods under the conceptual node for addition. In Method 1, the standard algorithm, the columns are added from right to left with any carries being written above and included in the column sum of the next column to the left. In Method 2, the columns are added from left to right with any carries being written below the answer in the next column to the left. If any carries occur in the problem, they must be added in a second addition. Notice that these two methods share the common procedures for calculating a column sum and writing a digit in the answer, but differ in the procedure they use when carrying is necessary. One structural aspect of the network is to make explicit any subprocedures that can be potentially shared by several higher level procedures.

The decomposition of a complex skill into all of its conceptual procedures terminates in some set of primitives that reflect assumed elements of an underlying computational ability. For addition, typical assumed primitives are recognizing a digit, writing a digit, and knowing the concepts of right, left, etc. The complete procedure network explicitly specifies all the subprocedures of a skill and can be evaluated or "executed," thereby simulating the skill for any given set of inputs. By itself, this network merely provides a computational machine

that performs the skill and is not of particular import. However, the possible misconceptions in this skill are represented in the network by incorrect implementations associated with subprocedures in its decomposition called "bugs".[4] Each buggy version contains incorrect actions taken in place of the correct ones.[5] An extension to the network evaluator enables the switching in of a *buggy* version of a procedure that allows the network to simulate the behavior of that buggy subskill. This feature provides a computational method for determining the external behavior of the underlying bugs.

Inferring a Diagnostic Model of the Student

The problem of diagnosing a *deep structure* failure in a student's knowledge of a procedural skill can now be accomplished, at least theoretically, in a straightforward manner. Suppose, as in the examples on page 157, we are provided with several *surface* manifestations of a deep structure misconception, or bug, in the student's addition procedure. To uncover those possible subprocedures which are at fault, we use the network to simulate the behavior of buggy subprocedures over the set of problems and note those which generate the same behavior as exhibited by the student. To catch a student's misconceptions that involve more than one faulty subprocedure, we must be able to simulate various combinations of bugs. A student may have a bug in his carrying procedure as well as believing that 8+7 is 17 (a bug in his addition facts table). To model his behavior, *both* buggy versions must be used *together*. A *deep-structure model* of the student's errors is a set of buggy subprocedures that, when invoked, replicates those errors. Each buggy version has associated information such as what the underlying causes of the bug may have been, as well as specific remediations, explanations, interactions, and examples of the bug—all of which may be used by a tutoring system to help correct the student's problem.[6]

Many technical questions are raised by the above brief overview of how to "infer" a diagnostic model. We have deferred a more detailed discussion of these questions until Section 3 in favor of a more general discussion of pedagogical ramifications of the procedural network model of procedural skills. We begin this discussion with a description of the procedural network for a simple skill—subtraction.

[4]The term "bug" is borrowed from computer science where it refers to a mistake in a computer program.

[5]A "buggy" implementation does, in general, call other "correct" subprocedures though they may be called at inappropriate times or their results used incorrectly.

[6]West (1971) has broken down the diagnostic teaching task into four steps: (i) distinguish between conceptual and careless errors; (ii) identify the exact nature of the conceptual error (bug); (iii) determine the conceptual basis (cause) of the bug; and (iv) perform the appropriate remediation. The system we describe has been directed toward problems (i) and (ii). The buggy implementation nodes in the network provide the proper places to attach information relevant to problems (iii) and (iv).

A Procedural Network for Subtraction

As an example of the surprising amount of procedural knowledge needed to perform a simple skill, let us consider a more complete network representation of the subtraction of two numbers.[7] Figure 2 shows the links of the procedural network for subtraction that indicate which subprocedures a procedure may use.

The topmost node (SUBTRACT) represents the subtraction of two n-digit numbers. It may use the procedures for setting up the problem (SETUP), transforming it if the bottom number is greater than the top (TRANSFORM), and sequencing through each column performing the column subtraction (COLUMN SEQUENCE). The implementation of the column subtraction procedure has to account for cases where borrowing is necessary (BORROW NEEDED) and may call upon many other subprocedures, including taking the borrow from the correct place (DO BORROW), scratching 0 and writing 9 if that place contains a zero (ZERO), and so on. An important subprocedure is the facts table look-up (FACTS TABLE) which allows any of the simple arithmetic facts to be wrong. The facts table subprocedure is called during the addition of 10 to a column digit

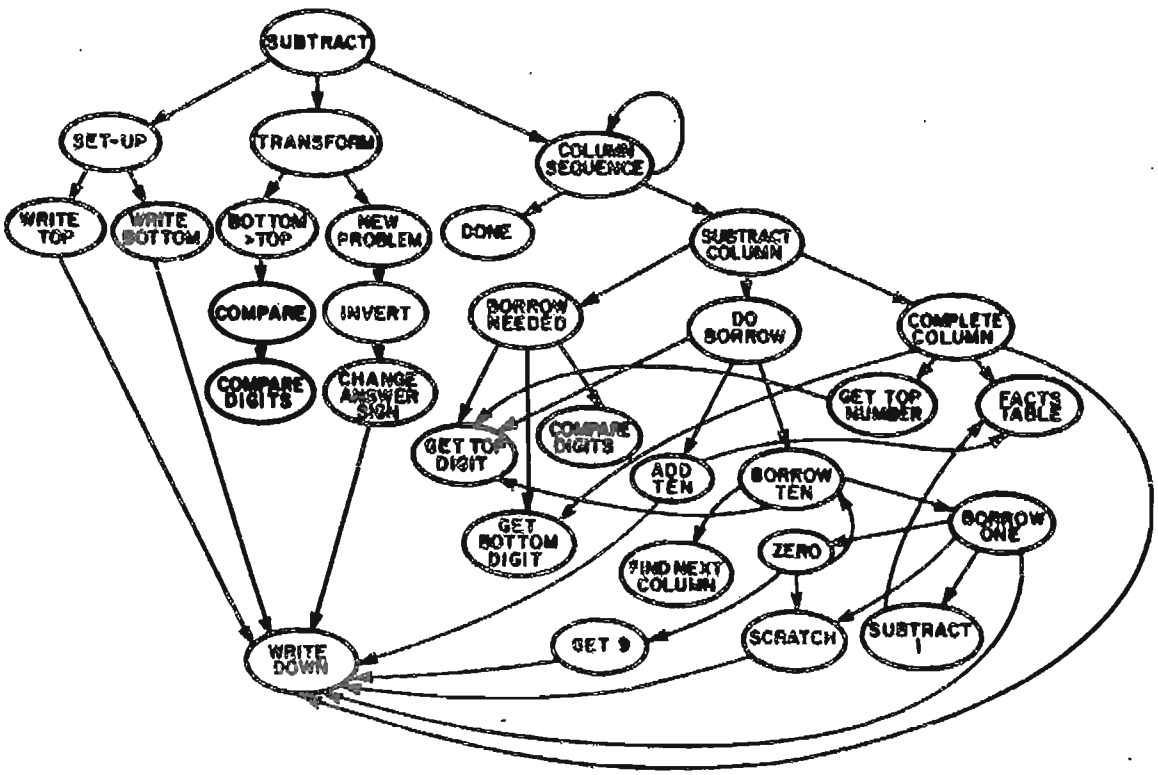

FIG. 2 A procedural network for subtraction.

[7]We have chosen just one of the several subtraction algorithms (the so-called "standard" algorithm), but the ideas presented here apply equally to others and can handle multiple methods as well.

143 −28 125	The student subtracts the smaller digit in each column from the larger digit regardless of which is on top.
143 −28 125	When the student needs to borrow, he adds 10 to the top digit of the current column without subtracting 1 from the next column to the left.
1300 −522 878	When borrowing from a column whose top digit is 0, the student writes 9 but does not continue borrowing from the column to the left of the 0.
140 −21 121	Whenever the top digit in a column is 0, the student writes the bottom digit in the answer; i.e., $0-N=N$.
140 −21 120	Whenever the top digit in a column is 0, the student writes 0 in the answer; i.e. $0-N=0$.
1300 −522 788	When borrowing from a column where the top digit is 0, the student borrows from the next column to the left correctly but writes 10 instead of 9 in this column.
321 −89 231	When borrowing into a column whose top digit is 1, the student gets 10 instead of 11.
662 −357 205	Once the student needs to borrow from a column, s/he continues to borrow from every column whether s/he needs to or not.
662 −357 115	The student always subtracts all borrows from the leftmost digit in the top number.

FIG. 3 Manifestations of some subtraction bugs.

(ADD 10), during the subtraction of 1 from a column digit in a borrowing operation (SUBTRACT 1), as well as during the subtraction of the bottom from the top digit in a column (COMPLETE COLUMN).

In principle, each of these subprocedures could have many buggy versions associated with it.[8] An example of a common bug is to calculate the column difference by subtracting the smaller digit from the larger regardless of which is on top. In another bug, the SETUP procedure left justifies the top and bottom numbers so that when the student is told to subtract 13 from 185, he gets 55. An

[8]In our current subtraction network, some subprocedures have only one buggy version, while others have as many as fifteen. The average is three or four.

interesting aspect of the left justification bug is that when the student is faced with seemingly impossible problems (185−75) he may be inclined to change the direction in which he subtracts, borrowing from right to left instead of from left to right, or to change his column difference procedure to larger minus smaller, thereby eliminating the need to borrow. Thus, there can exist relationships between bugs such that one bug suggests others. A major challenge in identifying the procedural breakdown or description of a skill is to have the network handle such ramifications and interactions of multiple bugs.

To provide a feeling for the range of "answers" that can come from simple underlying bugs, we have included in Fig. 3 "answers" to subtraction problems using some of the bugs in the procedural network for subtraction. Notice that a particular answer to a given problem can have more than one explanation since several distinct bugs can generate the same answer. A special case is that a student may harbor many misconceptions and still get the correct answer to a particular problem!

The Power of Simulating Bugs in the Network

Given a procedural network like the one in Fig. 2, it is not always obvious how bugs in any particular subprocedure or set of subprocedures will be manifested on the surface, that is, in the answer. Some of the complicating factors are that a single buggy subprocedure can be used by several higher-order procedures in computing an answer or that two bugs can have interactions with each other. These factors are further complicated by the fact that not all sample problems will manifest all of the possible symptoms. If asked to make predictions about the symptoms of a given bug, people often determine the symptoms by considering only the skills or subprocedures used in solving *one* particular sample problem. As a result, they often miss symptoms generated by other procedures that can, in principle, use the given buggy subprocedure but which, because of the characteristics of the particular problem, were not called upon. If a different sample problem had been chosen, it might have caused the particular faulty subprocedure to have been used for a different purpose, thereby generating different symptoms. These observations first led us to consider the value of simulation to systematically verify a conjectured bug.

2. PEDAGOGICAL ISSUES

BUGGY: An Instructional Game for Training Student Teachers (and Others)

As we saw in Section 1, it is often difficult to infer a student's bug from his answers. It was the need for teachers to appreciate this problem and to develop strategies for coping with the possible range of student bugs that led us to construct a game called BUGGY. BUGGY is a computerized game based on the diagnostic interactions of a tutor and a student. The computer plays the part of an

errant student whose malady must be identified by the diagnostician (a role played either by an individual or by a team). The diagnostician is presented with an arithmetic homework problem that the "student" has done incorrectly. The "student's" behavior is generated using the procedural network and shows the symptoms of an underlying bug in one of the arithmetic subprocedures. The diagnostician's job is to discover exactly what is the underlying misconception. He does this by providing strategic *test problems* for the "student" to solve. The problems given by the diagnostician are answered by the computer acting as the student using the bugged procedure.

The computer also acts as arbiter in the evaluation of the diagnostician's theory. When the diagnostician indicates that he has discovered the bug, he is asked to describe it. Then, to make certain that he really has found the bug, he is given a five-problem test during which he must answer the problems in the same errant way that the student would. In other words, he must simulate the "student's" bug. When the diagnostician's concept of the bug is sufficiently well formulated that he is able to predict the buggy answer on all five problems, he is considered to have diagnosed the student and he proceeds to a new bug.

We have found it beneficial to allow teachers to team up while playing the diagnostician's role in BUGGY. The interaction between team members encourages verbalization of the decision-making processes involved in the game.

The following is a protocol of a team of teachers using BUGGY. The dialogue is interspersed with commentary about the important issues. Those lines typed by the team are marked with a vertical line in the left margin.

Protocol of a Team Using BUGGY

WELCOME TO BUGGY.
I HAVE CHOSEN A BUG. HERE IS AN EXAMPLE OF THE BUG.

```
 17
+5
---
 13
```

The system presents an example in which the student does not get the right answer. At present the initial examples are stored in the network linked to the buggy implementation.

NOW YOU CAN GIVE ME PROBLEMS TO DETERMINE THE BUG.

```
| 18    43
| +6   +79
  --   ---
  15    23
```

The team generates problems that are answered by the "student". Notice that the team's test problems are similar to the initial example generated by BUGGY. This is usually a poor diagnostic strategy as it may leave other symptoms of a bug undiscovered.

| Got the bug!

The team forms an initial hypothesis and tells the system they have discovered the bug. The system now asks the team to describe the bug and then tests them on their understanding. Asking for a description forces the team to verbalize their sometimes fuzzy idea of what the bug is and

often makes them realize how ill-defined or vague their idea actually is. In the present system, the descriptions are saved but no attempt is made by the system to understand them.

PLEASE DESCRIBE THE BUG.

| Student adds the number to be carried into the last column worked in.

This cryptic description is typical of those we have observed. Describing bugs in English is surprisingly difficult. What the team is trying to say is that if a column adds to more than 9, the units digit and the carry digit are added together and the result is written as the answer. For example, in the first column of the example problem, when 7 and 5 are added, 1 (the carry) and 2 (the units digit) are added to get 3. The same with 8+6, 2+9, and 3+9. (This bug could easily arise if the student were using Method 2 from Fig. 1 and mistakenly wrote the carry under the same column.)

The hypothesis does account for the observed behavior. Unfortunately, it is not the student's bug. Notice the possibly humorous, probably disastrous results that would have transpired if the team had stopped at this point and tried to tutor the student on the basis of their initial belief!

HERE ARE SOME PROBLEMS TO TEST YOUR THEORY ABOUT THE BUG.
WHAT IS:

```
  21
+39
| 51
```

To ensure that the team has found the correct bug, the system produces several problems for the team to solve. At present, BUGGY has a heuristic problem generation program that gives a broad range of "different" problems. (For examples in which the bug is very localized, such as requiring a 0 in the top number, a special problem generator can be linked to the bug and will be used instead.) These problems are then filtered to try to get three problems which cause symptoms and two that give correct answers. This simple routine does surprisingly well at producing critical problems. So well, in fact, that a more "intelligent" routine (which determined the set of bugs consistent with the student's data and then selected the test problems which split this set) was discarded as unnecessarily time-consuming. However, this more intelligent technique—based on techniques discussed in Section 3—might be reverted to as our theory of procedural bugs becomes progressively more subtle.

The team's hypothesis is revealed as incorrect by the first of the five test problems. BUGGY presents its answer and then allows the team to enter more problems.

THAT'S NOT THE BUG I HAVE.
USING MY BUG:

```
  21
+39
  15
```

TRY GIVING ME SOME MORE EXAMPLES.

```
|   51      99      68
| +1707    +99      +9
    21      36      23
```

After failing the test, the team gives a greater range of problems, specifically those using bigger numbers. How is it that 51+1707 can add up to 21? By broadening the scope of example problems they are able to reevaluate their initial hypothesis, dismiss it, and develop another one that is complete and consistent with respect to the new errors.

| Got the Bug!

PLEASE DESCRIBE THE BUG YOU THINK I HAVE.

The student is adding all the digits together.

HERE ARE SOME PROBLEMS TO TEST YOUR THEORY ABOUT THE BUG.
WHAT IS:

33	1091	8	28	90
+99	+60	+34	+70	+6
24	17	15	17	15

This time the team's hypothesis passes the test and they are given the description of the bug stored in the procedural network.

VERY GOOD. MY DESCRIPTION OF THE BUG IS:
THE STUDENT ALWAYS SUMS UP ALL THE DIGITS WITH NO REGARD TO COLUMNS.

A variant of this game assigns a cost to each problem the user poses in the process of trying to discover the bug. At the same time, a much higher cost is assigned for claiming to have "discovered" the bug but then failing to pass criterion on the computer-generated test problems. A correctly structured cost algorithm encourages the user to think carefully about the diagnostic value of each problem he creates and not to jump to a conclusion too quickly or too slowly.

Pedagogical Issues for Teachers

One application of BUGGY and the "diagnostic model" view of procedural skills lies in the domain of teacher training. The realization that errors that appear "random" are often the surface manifestations of a systematic underlying bug is a major conceptual breakthrough for many student teachers. All too often, behavior that appears to be random has a simple, intelligent, and complete underlying explanation. By proper diagnosis, remediation for a student can be directed toward his specific weaknesses. The importance of simply admitting that there may exist underlying bugs cannot be overstressed. Without appreciation of this fact, a teacher must view failure on a particular problem as either carelessness or total algorithm failure. In the first case, the predicated remediation is giving more problems, while in the second, it is going over the entire algorithm.[9] When a student's bug (which may only manifest itself occasionally) is not recognized by the teacher, the teacher explains the errant behavior as carelessness, laziness, or worse, thereby often mistakenly lowering his opinions of the student's capabilities.

From the student's viewpoint, the situation is much worse. He is following what he believes to be *the* correct algorithm and, seemingly at random, gets marked *wrong*. This situation can be exacerbated by improper diagnosis. For example, Johnnie subtracts 284 from 437 and gets 253 as an answer. "Of

[9]In computer programming metaphors, this approach corresponds to the debugging activities of resubmitting the program because the computer must have made a mistake and of throwing the whole program away and starting over from scratch and writing a new program.

course", says the teacher "you forgot to subtract 1 from 4 in the hundreds place when you borrowed." Unfortunately Johnnie's algorithm is to subtract the smaller digit in each column from the larger. Johnnie does not have any idea what the teacher is talking about (he never "borrowed"!) and feels that he must be very stupid indeed not to understand. The teacher agrees with this assessment as none of his remediation has had any effect on Johnnie's performance.

BUGGY, in its present form, presents teachers with examples of buggy behavior and provides practice in diagnosing the underlying causes of errors. Using BUGGY, teachers gain experience in forming theories about the relationship between the symptoms of a bug and the underlying bug itself. This experience can be cultivated to make teachers aware that there are methods or strategies that they can use to diagnose bugs properly.

In fact, there are a number of *strategy bugs* that teachers may have in forming hypotheses about a student's misconceptions. That is, the task of diagnosing a student is a procedural skill and as such is susceptible to mislearning by the teachers. A common strategy bug is to jump too quickly to one hypothesis. Prematurely focusing on one hypothesis can cause a teacher to be unaware that there may be other competing hypotheses that are possibly more likely. A common psychological effect of this approach is that the teacher generates problems for the student that confirm his hypothesis! In some cases, teachers may believe their hypotheses so strongly that they will ignore contrary evidence or decide that it is merely random noise. One general diagnostic strategy that avoids this pitfall is the technique of differential diagnosis (Rubin, 1975) in which one always generates at least two hypotheses and then chooses test problems that separate them.

Another common strategy bug is to lock onto only one type of symptom. For example, one student teacher was given the initial example (A), after which he proceeded to generate example problems (B) and (C):

A	B	C
19	23	81
+9	+6	+8
199	236	818

At this point, he concluded that the bug was "writes the bottom digit after the top number." But his hypothesis failed when he was given the first test problem:

$$\begin{array}{r} 8 \\ +12 \\ \hline \end{array}$$

to which he responded 812. The bug is that single digit operands are linked on to the end of the other operand, so that the correct buggy answer is 128. By presenting examples only with a shorter bottom digit, he had obtained what seemed to be confirming evidence of his hypothesis. A general rule which could

be employed to avoid this fixation is that whenever an example of incorrect behavior has an asymmetry (length of top and bottom numbers), then try an example with the asymmetry reversed. Using this rule, the teacher would also generate problems with larger top numbers before he reached a conclusion. BUGGY provides an environment in which teachers can experience the ramifications of not employing rules and strategies during diagnosis.[10]

Another important issue concerns the relationship between the language used to describe a student's errors and its effect on what a teacher should do to remediate it. Is the language able to convey to the student what he is doing wrong? Should we expect teachers to be able to use language as the tool for correcting the buggy algorithms of students? Or should we expect teachers only to be able to understand what the bug is and attempt remediation with the student with things like manipulative math tools? The following descriptions of hypotheses given by student teachers, taken from protocols of BUGGY, give a good idea of how difficult it is to express procedural ideas in English. The descriptions in parentheses are BUGGY's prestored explanations of the bugs.

"Random errors in carryover." (Carries only when the next column in the top number is blank.)

"If there are fewer digits on the top than on the bottom she adds columns diagonally." (When the top number has fewer digits than the bottom number, the numbers are left-justified and then added.)

"Does not like zero in the bottom." (Zero subtracted from any number is zero.)

"Child adds first two numbers correctly. Then when you need to carry in the second set of digits child adds numbers carried to bottom row, then adds third set of digits diagonally, finally carrying over extra digits." (The carry is written in the top number to the left of the column being carried from and is mistaken for another digit in the top number.)

"Sum and carry all columns correctly until get to last column. Then takes furthest left digit in both columns and adds with digit of last carried amount. This is the sum." (When there are an unequal number of digits in the two numbers, the columns that have a blank are filled with the leftmost digit of that number.)

Even when one knows what the bug is in terms of being able to mimic it, how is one going to explain it to the student having problems? Considering the above examples, it is clear that anyone asked to solve a set of problems using these explanations would, no doubt, have real trouble. One can imagine a student's frustration when the teacher offers an explanation of why he is getting problems marked wrong, and the explanation is as confused and unclear as these are.

For that matter, when the correct procedure is described for the first time, could it, too, be coming across so unclearly!

The problem of adequately describing bugs is further complicated by another

[10]One job for an "intelligent" tutor for BUGGY is to recognize and point out places where instances of general rules and strategies should be used. This possibility is discussed in Section 4.

surprising fact: Fundamentally different bugs can cause identical behavior! In other words, there can be several distinct ways of incorrectly performing a skill that *always* generate the same "answers". For example, here is a set of problems:

38	186	298	89
+46	+254	+169	+64
174	2330	2357	243

One possible bug which accounts for these results is: the columns are added without carries and the leftmost digit in the answer is the total number of carries required in the problem. In this case, the student views the carries as tallies to be counted and added to the left of the answer. But another equally plausible bug also exists; the student places the carry to the left of the next digit in the top number; then, when adding that column, instead of adding the carry to the digit, he mistakes it as a tens column of the top digit—so that when adding 298 and 169, in the second column he adds 19 to 6 instead of 10 to 6. This generates the same symptoms. So even when the teacher is able to describe clearly what he believes is the underlying bug, he may be addressing the wrong one. The student may actually have either one of these bugs.[11]

An Experiment Using BUGGY with Student Teachers

To determine BUGGY's impact on student teachers, we had a group play the game described in the beginning of this section. The goal of the experiment was to explore whether exposure to BUGGY significantly improves the student teacher's ability to detect regular patterns of errors in simple arithmetic problems.[12] The subjects were undergraduate education majors from Lesley College in Cambridge. Their exposure to BUGGY lasted approximately one and a half hours, during which time both addition and subtraction bugs were presented. The effects of their exposure to BUGGY were measured by comparing each subject's performance on the pre- and postexposure debugging test. The detailed analysis and discussion of the experiment is beyond the scope of this paper but has been described in a technical report (Brown et al., 1977). Briefly, though, the results of the experiment showed that exposure to BUGGY significantly improved their ability to detect regular patterns of errors.

We also investigated the qualitative issue of what the student teachers felt they gained from their exposure to BUGGY. To assess their impressions, we convened the entire group after they had finished using BUGGY. At that gathering, we first asked them to write their responses to two questions (discussed below), and then we taped a final group discussion in which we sought their reactions to using

[11]This possibility leads to an interesting question concerning how one can "prove" that two different bugs entail logically the same surface manifestations.

[12]We would like to thank Dr. Mark Spikell of Lesley College for his assistance in this endeavor.

BUGGY and their suggestions for its deployment with school-aged students. The following week, their professor, who also participated in the initial experiment, held a second group discussion and reported to us the consensus, which was consistent with what they had written.

Appendix 1 lists some of the written responses to the question "What do you think you learned from this experience?". All 24 responded that they came away with something valuable. Many stated that they now appreciated the "complex and logical thought process" that children often use when doing an arithmetic problem incorrectly. "It makes me aware of problems that children have and they sometimes think logically, not carelessly as sometimes teachers think they do." "I never realized the many different ways a child could devise his own system to do a problem." They also stated that they learned better procedures for discovering the underlying bug. "I learned that it is necessary to try many different types of examples to be sure that a child really understands. Different types of difficulties arise with different problems."

We also asked the students "What is your reaction to BUGGY?" Many felt that "BUGGY could be used to sharpen a teacher's awareness of different difficulties with addition and subtraction." They felt that it might be of use in grade school, high school, or with special needs students, or even as a "great experience in beginning to play with computers."

Pedagogical Issues More Specific to Middle School Students

We feel that all of the issues discussed above are as important for school-level students as they are for teachers. There is great value in introducing young students to procedural notions. The BUGGY system provides a well-controlled environment for such an introduction, as well as one that can be *easily* integrated into a *standard curriculum*. Also note that for a middle school student to play the diagnostician's role requires his studying the procedural skill per se (i.e., its structure) as opposed to merely performing it. This experience can be especially important as students begin algebra, which is their first exposure to procedure "schema." By presenting procedures as objects of study, BUGGY can thus be used to explore the powerful ideas of hypothesis formation, debugging, debugging strategies, and so on. Of course, such a use requires more than just the BUGGY game.

Another reason for having students develop a language for talking about procedures, processes, bugs, and so forth is that such a language enables them to talk about, and think about, the underlying causes of their own errors. This facility is important in its own right, but it also gives a student the motivation and the apparatus for stepping back and critiquing his own thinking, as well as saying something interesting and useful about his errors. The difficulty in getting a student to test the plausibility of his own answer (such as by estimating it) may be due to the lack of any appropriate introspective skills that the student could use once he knew his answer was wrong.

An important ancillary, nonmathematical benefit of a student's involvement with BUGGY is exposure to the idea of role reversal.[13] To communicate effectively with others, children must learn not only language, but also the use of "social speech"—speech that takes into account the knowledge and perspective of another person (Krauss & Glucksberg, 1977). Piaget uses the term "childhood egocentrism" to describe the child's inability to detach himself from his own point of view and to take into consideration another's perspective. Although Krauss and Glucksberg agree that egocentrism plays a large part in very young children's speech, they believe that even in older children the ability to role-play breaks down when they are faced with a demanding cognitive task. We believe that taking on the viewpoint of the errant student by analyzing another's mistakes can provide valuable practice in role-playing in a demanding situation and can be beneficial to the development of "social speech."

Some Results on Using BUGGY with Seventh and Eighth Graders

To explore the effect of the BUGGY game on seventh and eighth graders and to discover what type of additional instructional material and activities must back up the use of this computer-based system, we placed a terminal running BUGGY in a classroom. The teacher provided a short introduction to the game and the notion of bugs and then the students[14] were free to use the system during the term. During this experience, we noticed the following phenomena. When the students first started trying to discover the underlying bugs in BUGGY, their most common reaction, upon seeing the mistakes of the simulated buggy student, was to exclaim "Wow, how dumb and stupid this kid must be." However, after a week or so of exposure to BUGGY, the students' reactions changed from believing that the simulated student was dumb to one of appreciating that there was, in fact, a systematic explanation for what this student was doing. They began to see that he (it) had fundamental misconceptions as opposed to being just stupid. This result is particularly exciting, since it paves the way for students to see their own faulty behavior not as being a sign of their stupidity, but as a source of data from which they can understand their own errors.

Unfortunately, we have as yet no data concerning the transferability of this awareness, and whether or not it will lead to students being more capable and willing to look over their own work for errors.

[13]This idea is attributable to Tim Barclay, at the Cambridge Friends School, who has been experimenting with various uses of BUGGY with sixth through eighth graders.

[14]All of the students participating in this activity had already mastered the procedural skills being "bugged." In fact, we have severe reservations about the advisability of younger students using this particular game especially if they have not mastered the correct versions of the skills. However, there are extensions to BUGGY (discussed in Section 4) that make it appropriate for school children in the process of learning the given procedural skills.

3. AN AUTOMATED DIAGNOSTIC MODELING SYSTEM

In this section, we describe a diagnostic system that is based on our arithmetic procedural network, recently completed by Burton. The notion of modeling procedural skills as *executable* networks and then expressing all potential misconceptions as buggy versions of subprocedures in the network provides a technique for efficiently determining the consequences or symptoms of a given collection of bugs on a set of problems. It therefore has the potential of diagnosing and explaining all of the procedurally incorrect answers for any given problem. For example, as we indicated in Section 1, given a procedural network for addition and an addition problem (like 35+782), all the buggy subprocedures as well as combinations of buggy subprocedures, can be inserted and then executed in the addition procedure one by one so that all the possible buggy answers to the problem are generated. Those answers can then be compared to a student's work to determine possible explanations of the student's particular misconceptions. In this way, one might use the procedural network to diagnose a student's basic misconceptions over an entire homework assignment or an arithmetic test.

There are, however, several complications with this simple paradigm for diagnosing a student. One is that the student who has developed a *novel* singleton bug (as opposed to one that arose out of a combination of "primitive" bugs) will not be diagnosed. Another is that students *do* make "random" mistakes (presumably as many while following an *incorrect* procedure as a correct one) that could erroneously lead to the exclusion of his bug or the inclusion of another bug that happened to coincide with his "randomness." Finally, blindly considering all possible combinations of bugs can lead to a combinatorial explosion of possibilities. In what follows, we shall discuss some solutions to these problems.

A Deep-Structure Data Analysis Tool

As a first step in developing techniques for automatically diagnosing a student's errors, we sought a large data base of student answers on some arithmetic test. We started out this way for two reasons. First, an analysis of the student errors not already explained by the existing network can suggest extensions to the network that may have been overlooked. Obviously, if the BUGGY system is going to be useful as a diagnostic tool, its arsenal of bugs must be very extensive! Second, once a "complete" network of bugs has been constructed, analyzing a large data base can provide some evidence of how many student errors are procedural rather than "careless" errors. Such an analysis also indicates how consistently students apply buggy procedures. Do they use the buggy procedure every time it is appropriate? When they get a correct answer does that contraindicate their use of a buggy procedure, or is it just that the buggy procedure for this problem *does* produce the correct answer? This analysis also reveals which procedural errors occur most often, and in what combinations. Answers to these

questions not only influence the design of the diagnostic system (as we will see later) but also have an impact on how that system could be used.

We were fortunate to be able to obtain a large collection of pertinent data[15] already in computer readable form. This data stemmed from an achievement test administered in Nicaragua to fourth, fifth, and sixth graders. There were 10 different test versions, each consisting of 30 problems combining both simple and complex addition and subtraction problems. One version of the test is given in Fig. 4. The makeup of each test followed a complex procedure discussed by Friend (1976).

1	8	7	99	43	353	213	633	521	81
+2	−3	+9	−79	+41	−342	21	−221	502	−17
123	4769	9	257	597	6523	156	103	8	7315
13	−0	91	−161	+75	−1280	873	−64	54	−6536
610		+6				+311		+9	
+12									
505	1039	77	705	917	10038	864	10060	579	7001
743	−44	18	−9	639	−4319	9	−98	96	−94
12		+47		+5		4		833	
+35						+3		+43	

FIG. 4 A sample test.

Admittedly, these data have the limitation that the particular results derived from them are not necessarily generalizable to American schools. Although the procedures taught for addition and subtraction are similar, the environmental and cultural experiences of the students are quite different. Nevertheless, this data base provides a convenient starting point for this research as well as a general idea of the percentage of students making procedural as opposed to "random" errors. In addition, the methods we devised to analyze these data would apply equally well to data collected under other circumstances.

The Process of Organizing the Data

The data base available for this study was large—19,500 problems performed by 1300 students. We limited ourselves to consideration of just the subtraction problems on the test because the addition problems included some in which three

[15]The data used in this study were made available by the Institute for Mathematical Studies in the Social Sciences at Stanford University. The data were collected in Nicaragua as part of the Nicaragua Radio Mathematics Project supported by Contract A10/CM-ta-C-73-40 from U.S. Agency of International Development. We are grateful to Barbara Searle for allowing us access to the data. See Searle, Friend, and Suppes (1976) for a description of the NRM project.

TABLE 1
An Initial Bug Comparison Table[a]

8	99	353	633	81	4769	257	6523	103	7315	1039	705	10038	10060	7001
3	79	342	221	17	0	161	1280	64	6536	44	9	4319	98	94
5	20	11	412	64	4769	96	5243	39	779	995	696	5719	9962	6907
Student answers														
—	—	—	—	98	—	418	—	169	738	1095	706	14319	10078	7097
FORGET/BORROW/OVER/BLANKS														
*	*	*	*	!	*	!	*	139	!	***	***	15719	10062	7007
STOPS/BORROW/AT/ZERO														
*	*	*	*	!	*	!	*	49	!	***	***	6719	10062	7017
DIFF/0−*N*=*N*														
*	*	*	*	!	*	!	*	!	839	!	!	***	9978	!
ADD/INSTEADOF/SUB														
11	178	695	854	***	*	***	7803	167	13851	1083	714	14357	10158	7095

[a] This table presents how well a student's answers are explained by different bugs; "***" and "*" indicate places of agreement. (See text for discussion.)

or more numbers were to be added—a condition that can produce errors not modeled in the present addition network.

As a first step, we extended the BUGGY system to print out "bug comparison" tables for students as shown in Table 1. These tables summarize how well the student's behavior can be explained or predicted by a simple bug in the network. The format of a table is as follows: The problems with the correct answers appear at the top of each table. The student's answers appear on the next line using the convention that "—" indicates a correct student answer. Each of the remaining lines provides the name of a bug and the answers produced by the assumption that the student had this and only this bug. For each of these lines a "***" means that the bug predicted the student's incorrect answer. A "*" means that the bug in that row would give the correct answer and also that the student got the correct answer. Thus "*" and "***" indicate places of agreement between the student's behavior and the simulated behavior of the bug. An "!" means that the bug would give the correct answer, but that the student gave an incorrect one.[16] A number which appears in a bug row is the answer that the bug would give when it is different from both the student's answer and the correct answer.[17]

[16] In both the "*" and the "!" cases, the bug has not manifested itself in the answer as an error. For example, if a bug is 0−*N*=0, it would not show itself in a problem unless there is a 0 in the problem's top number (or 0 is generated during solution by borrowing from a column with a 1).

[17] An additional case arises when the student does not answer a problem. Although none of our example tables will include this case, it is marked with a "#". "#" is also used in a bug row to indicate that the bug could not do the problem. We saw very little evidence of students not doing a problem possibly because the students were given as much time as they wanted to complete the tests.

Initially, any bug that explained any of the student's behavior (i.e., generated at least one "***") was included in the table. However, these tables proved to be too large to conveniently read so a routine was added to delete any bug if there was another bug that accounted for the same set of answers as well as some others.

Analysis of a Bug Comparison Table

When we are determining whether or not a student has a particular bug, "*" and "***" are confirming evidence that the student has the bug while both "!" and numbers in the bug row are disconfirming evidence. We refer to the results in a bug comparison table as "evidence" because there may be several possible explanations for any particular answer to one problem. The student may have made a careless error while following his bugged procedure, therefore leading to a number in that bug row instead of a "***". He may have an unmodeled combination of bugs, only one of which manifested itself and resulted in an "!" in the row of the other bug. Or he may have been following a totally different procedure, or no procedure at all, that just happened to give him the same answer as a bug leading to a "***" in a bug row. The final decision on whether or not the student is using a buggy subprocedure must be made by considering all of the evidence from the test. But how should *conflicting evidence* be weighted and summed?

Let us consider an analysis of the bug comparison table for a particular student, Table 1. Both FORGET/BORROW/OVER/BLANKS and STOPS/BORROW/AT/ZERO produce the same answer in problems 11 (1039−44) and 12 (705−9)—the errant student answers. But neither has particularly good agreement across the rest of the table. Which, if either, misconception was the student operating under? Our inclination is to believe that neither bug satisfactorily explains the behavior, but how does one decide in general? To answer these questions, we analyzed, by hand, several hundred students' tables like the one in Table 1. During these formative analyses, the tables were examined to ferret out students whose behavior was not captured by any existing bugs. The work of these students was closely scrutinized for any underlying computational pattern. If a pattern could be discerned, the incorrect subprocedure was defined and this new "bug" was added to the network. During this formulation period our list of bugs grew from 18 to 60.

Multiple Bugs and Their (Nonobvious) Interactions

Most of the 60 subtraction bugs discovered during this period were primitive, in the sense that each redefined only one subprocedure in the subtraction network. Was it possible that some of the students' behavior was due to multiple bugs that we had failed to notice? To explore this possibility, we programmed the BUGGY diagnostic system to try all *pairs* of bugs. That is, buggy definitions of

TABLE 2
Multiple Bug Comparison Table

8	99	353	633	81	4769	257	6523	103	7315	1039	705	10038	10060	7001
3	79	342	221	17	0	161	1280	64	6536	44	9	4319	98	94
5	20	11	412	64	4769	96	5243	39	779	995	696	5719	9962	6907
Student answers														
—	—	—	—	98	—	418	—	169	738	1095	706	14319	10078	7097
DIFF/0−*N*=*N* and STOPS/BORROW/AT/ZERO														
*	*	*	*	!	*	!	*	***	839	***	***	***	***	***
ADD/INSTEADOF/SUB														
11	178	695	854	***	*	***	7803	167	13851	1083	714	14357	10158	7095

two subprocedures were systematically inserted and then executed.[18] This process turned up 270 bug combinations whose symptoms were different from any of the primitive bugs and from each other in one test of 15 problems.

In order to illustrate the diagnostic power of this generative technique, consider the example of the student whose work is shown in Table 1. From Table 1, no discernible pattern is evident. The student's work does however admit to a beautifully simple characterization which is the composite of two primitive bugs. Table 2 shows the new bug comparison table that was generated for this student from comparisons using multiple bugs. Notice that the comparison line which resulted from the combination of the two bugs is substantially different from either of the single bug lines or even from a linear combination of the two comparisons. This is due to the nonobvious interactions of the two bugs, particularly where the intermediate products of one of the bugs enables or disables the other bug. For example, in problem 9 (103–64), the "0−*N*=N" bug alone will not manifest itself because the borrow from the first column will have changed the 0 in the second column to 9. However, the "stops-borrow-at-zero" bug[19] has the side effect of *not* changing the 0 to 9, and hence enables the "0−*N*=0" bug. In general, the interactions between bugs can be arbitrarily complex. This can make a teacher's diagnostic task very difficult.

[18]During this process, bugs which were alternative definitions of the same conceptual procedure, of course, could not be paired. There were also cases where one bug would preclude another. For example, SMALLER/FROM/LARGER precludes any of the bugs in the borrowing procedures as borrowing is never required. In these cases, a bug can prevent other portions of the network from ever being executed and hence "switching in" bugs in the unused portion would be useless as long as the higher bug remained in effect. Rather than an extensive analysis of the potential interactions of bugs, we opted for the simpler solution of comparing the bugs via their symptoms over a fixed set of problems.

[19]In the "stops-borrow-at-zero" bug, the student does not know how to borrow from a column which has a 0 in the minuend so he does not do anything to it. He does however add 10 to the column he is processing.

Judging Diagnostic Credibility

Using the multiple bug comparison tables generated for about one hundred students, we identified which of the students, in our judgment, were making procedural, as opposed to careless or random errors. During this hand-done classification process, we articulated and refined our intuitive use of the evidence from a student's entire test in order to make that decision. Eventually, our understanding and description of the process became precise enough to be computerized so that it could be run on all 1300 students.

Our hand-done study suggested six intuitive groupings of students:

1. Those students who got all the problems correct.
2. Those students who erred on any number of problems but whose errors were explained by one bug or one bug pair.
3. Those students who clearly exhibited the presence of a bug but who also exhibited some behavior that was not explained by the bug.
4. Those students who missed only one or two (of fifteen problems) and in a way not consistent with any bug.
5. Those students who exhibited some buggy behavior but not consistently.
6. Those students whose behavior appeared random relative to the known bugs.

Tables 3, 4, and 5 show representative students from Groups 2, 3, and 5. The intuitive justification for these groupings stemmed from the possible tutorial approaches a teacher might take to remediate a student. The classes of students and the possible very general tutorial approaches we saw are:

i. those students who are correct or very nearly correct and probably just need more practice if anything (Groups 1 and 4);
ii. those students who are exhibiting consistently incorrect behavior and, therefore, whose remediation may profitably be viewed as a process of "debugging" the student's present algorithm (Groups 2 and 3); and

TABLE 3
Example of a Student Whose Behavior is Well Explained by One Bug[a]

8	99	353	633	81	4769	257	6523	103	7315	1039	705	10038	10060	7001
3	79	342	221	17	0	161	1280	64	6536	44	9	4319	98	94
5	20	11	412	64	4769	96	5243	39	779	995	696	5719	9962	6907
Student answers														
—	—	—	—	—	—	—	—	139	—	1995	76	15719	10962	7007
BORROW/FROM/ZERO														
*	*	*	*	*	*	*	*	***	*	***	796	***	***	***
Best guess: BORROW/FROM/ZERO														
GROUP=2														

[a] The bug, BORROW/FROM/ZERO, is that when borrowing from a column in which the top number is 0, the student writes 9 but does not continue borrowing from the next column to the left.

TABLE 4
Example of a Student Who Exhibits a Consistent Bug but Also Has Other Problems

8	99	353	633	81	4769	257	6523	103	7315	1039	705	10038	10060	7001
3	79	342	221	17	0	161	1280	64	6536	44	9	4319	98	94
5	20	11	412	64	4769	96	5243	39	779	995	696	5719	9962	6907
Student answers														
—	—	—	—	74	—	—	—	9	1209	95	704	10019	70	6007
DIFF/0−*N*=0 and MOVE/OVER/ZERO/BORROW														
*	*	*	*	!	*	*	*	***	809	***	606	***	***	***
BORROW/ACROSS/SMALLER/ADDING/TEN/EXCEPT/ZERO and DIFF/0−*N*=0														
*	*	*	*	!	*	*	*	***	889	***	606	***	***	***
SUB/UNITS/SPECIAL and SMALLER/FROM/LARGER&0−*N*=0														
*	*	*	*	!	*	116	5363	!	***	1015	!	***	10042	!
BORROW/NO/DECREMENT and QUIT/WHEN/BOTTOM/BLANK														
*	*	*	*	***	9	196	5343	49	1889	***	6	6729	72	17
BORROW/NO/DECREMENT														
*	*	*	*	***	*	196	5343	149	1889	1095	706	16729	10072	7017
SMALLER/FROM/LARGER														
*	*	*	*	76	*	116	5363	161	1221	1015	***	14321	10038	7093
Best guess: DIFF/0−*N*=0 and MOVE/OVER/ZERO/BORROW														
GROUP=3														

iii. those students for whom a thorough reteaching of the entire algorithm appears to be essential (Groups 5 and 6).

We are definitely *not* saying that for all students in Groups 2 and 3 the only or even the best pedagogy is to focus on the student's buggy procedure. Instead, we are trying to identify those students who are consistently making the same mistake and for whom debugging of their procedures *may* be useful.

TABLE 5
Example of a Student Who Exhibits Some Buggy Behavior but not Consistently

8	99	353	633	81	4769	257	6523	103	7315	1039	705	10038	10060	7001
3	79	342	221	17	0	161	1280	64	6536	44	9	431	98	94
5	20	11	412	64	4769	96	5243	39	779	995	696	5719	9962	6907
Student answers														
!	!	!	!	70	!	98	!	109	839	1095	706	1431	10700	7001
DIFF/0−*N*=*N* and STOPS/BORROW/AT/ZERO														
*	*	*	*	!	*	!	*	169	***	***	***	***	10078	7097
DIFF/0−*N*=*N* and FORGET/BORROW/OVER/BLANKS														
*	*	*	*	!	*	!	*	139	***	***	***	***	1078	7007
DIFF/0−*N*=0&1−*N*=1 and STOPS/BORROW/AT/ZERO														
*	*	*	*	71	*	!	*	***	809	***	***	10019	10070	***
ZERO/INSTEADOF/BORROW														
*	*	*	*	***	*	106	5303	100	1000	1005	700	10020	10000	7000
GROUP=5														

Classification Algorithm

The algorithm that we eventually converged on for assigning a student to one of the six major categories defined in the previous section is rather involved, so some readers may prefer to skim the following discussion. If the student erred on at least one problem, each bug was rated according to how well it accounted for this behavior. This rating results in a group number for each bug.[20] The *rating* of each bug depends on the number of answers falling into each of five groups:

i. those student answers for which the bug predicts the student's incorrect answer (the number of "***"s appearing in the bug's line referred to as $N***$);
ii. those student answers for which the bug predicts the student's correct answer (the number of "*"s—$N*$);
iii. those student answers for which the bug predicts the correct answer but which the student answered incorrectly (the number of "!"s—$N!$);
iv. those student answers for which the bug predicts an incorrect answer but which the student answered correctly (Nr);
v. those student answers for which the bug predicts an incorrect answer different from the student's incorrect answer (Nw).

This analysis gives a symptom vector of five numbers ($N***$, $N*$, $N!$, Nr, Nw).

From the symptom vector, a group number corresponding to the six major categories of student behavior given in the previous section is calculated for *each bug* using the following procedure:[21]

A bug indicates Group 2 student behavior if it agrees with all of the student's answers ($N! + Nr + Nw = 0$) or if it agrees more than 75% of the time on problems in which it predicts a wrong answer [$N*** \geq 3 \times (N! + Nr + Nw)$].

A bug indicates Group 3 behavior if it explains two or more student errors and predicts more correct than incorrect answers ($N*** > Nr + Nw + N!/2$). In this formula, those problems in which the bug did not exhibit a symptom are weighted by half—the intuition is that, on these problems, the student may be exhibiting other bugs as well.

A bug is also rated as indicating Group 3 behavior if it is a primitive bug (not multiple) that predicts more than half of the student's errors, and predicts erroneous behavior more times than it fails to do so ($N*** > N! + Nw$ and $N*** > Nr$).

A bug indicates Group 5 behavior if it predicts at least two incorrect answers ($N*** \geq 2$).

Otherwise a bug is rated Group 6.

[20] In the actual implementation, the bugs are ordered by the number of symptoms accounted for and are rated from the most promising bug until the group number increases.

[21] As we have said, the classification scheme was based primarily on empirical studies. There are intuitive justifications (rationalizations) for each of the decisions; however, in the final analysis, this algorithm was used because it classified students in close accordance with our hand-done analysis.

TABLE 6
Totals and Percentages of Student Classifications[a]

	Group													
	1		2		3		4		5		6		Totals	
Grade	No.	(%)	No.	(%)	No.	(%)	No.	(%)	No.	(%)	No.	(%)	No.	(%)
4th	10	2.4	101	20.5	93	18.9	31	6.6	197	39.5	72	14.7	504	100.0
5th	10	3.0	86	22.0	73	18.7	38	10.0	132	33.5	60	15.5	399	100.0
6th	17	4.5	88	21.3	64	15.6	47	11.6	131	31.5	75	18.2	422	100.0
Totals	37	3.2	275	21.2	230	17.8	116	9.2	460	35.2	207	16.1		

[a] Groups 2 and 3 are consistently following an incorrect procedure (see text for further explanation of groups).

The student is assigned the group number of the lowest rated bug. If the lowest rated bug is not Group 2 or 3 and the student has missed only one or two problems, he is put in Group 4. If the student is put in Group 2 or 3, the bug with the lowest group rating (and which accounts for the most symptoms in cases of ties) is chosen as the most likely hypothetical student bug. Examples of this result can be seen in the last line of Tables 3 and 4.

Diagnostic Results for the Nicaraguan Data Base

The above classification procedure was used to analyze the set of test responses for 1325 fourth, fifth, and sixth graders.[22] A summary of the diagnostic classification by grade is given in Table 6. As can be seen, nearly 40% of the students exhibited consistently buggy behavior. This figure agrees with a similar result reported by Cox (1975). The similarity across grade level may be due to the fact that addition and subtraction are not presented again after the 4th grade.

Table 7 gives the frequency of the 14 most common bugs. Most of the difficulties arise while borrowing, especially when a zero is involved. The most common bug was "when borrowing from a column in which the top digit is 0, change the 0 to a 9 but do not continue borrowing from the next column to the left"; it occurred alone or together with other bugs 153 times in the 1325 student tests.

What Does a Test Score Mean?

One of the ramifications of this fully automatic diagnostic technique concerns its ability to score tests based on what a student knows or does not know as

[22] An implementation note: The entire data analysis program including the BUGGY subtraction models is written in INTERLISP. The analysis of all 1325 students against the 330 bugs required on the order of 90 minutes of CPU time on a PDP-KL10.

TABLE 7
Bug Frequency Table

The 14 most frequently occurring bugs in a group of 1325 students

57 students used: BORROW/FROM/ZERO (103−45=158)
When borrowing from a column whose top digit is 0, the student writes 9, but does not continue borrowing from the column to the left of the 0.

54 students used: SMALLER/FROM/LARGER (253−118=145)
The student subtracts the smaller digit in a column from the larger digit regardless of which one is on top.

50 students used: BORROW/FROM/ZERO and LEFT/TEN/OK (803−508=395)
The student changes 0 to 9 without further borrowing unless the 0 is part of a 10 in the left part of the top number.

34 students used: DIFF/0−*N*=*N* and MOVE/OVER/ZERO/BORROW
Whenever the top digit in a column is 0, the student writes the bottom digit in the answer; i.e., 0−*N*=*N*. When the student needs to borrow from a column whose top digit is 0, he skips that column and borrows from the next one.

14 students used: DIFF/0−*N*=*N* and STOPS/BORROW/AT/ZERO
Whenever the top digit in a column is 0, the student writes the bottom digit in the answer; i.e., 0−*N*=*N*. The student borrows from zero incorrectly. He does not subtract 1 from the 0 although he adds 10 correctly to the top digit of the current column.

13 students used: SMALLER/FROM/LARGER and 0−*N*=0 (203−98=205)
The student subtracts the smaller digit in each column from the larger digit regardless of which one is on top. The exception is that when the top digit is 0, a 0 is written as the answer for that column; i.e., 0−*N*=0.

12 students used: DIFF/0−*N*=0 and MOVE/OVER/ZERO/BORROW
Whenever the top digit in a column is 0, the student writes 0 in the answer; i.e., 0−*N*=0. When the student needs to borrow from a column whose top digit is 0, he skips that column and borrows from the next one.

11 students used: BORROW/FROM/ZERO and DIFF/*N*−0=0
When borrowing from a column whose top digit is 0, the student writes 9, but does not continue borrowing from the column to the left of the 0. Whenever the bottom digit in a column is 0, the student writes 0 in the answer; i.e., *N*−0=0.

10 students used: DIFF/0−*N*=0 and *N*−0=0 (302−192=290)
The student writes 0 in the answer when either the top or the bottom digit is 0.

10 students used: BORROW/FROM/ZERO and DIFF/0−*N*=*N*
When borrowing from a column whose top digit is 0, the student writes 9, but does not continue borrowing from the column to the left of the 0. Whenever the top digit in a column is 0, the student writes the bottom digit in the answer; i.e., 0−*N*=*N*.

10 students used: MOVE/OVER/ZERO/BORROW (304−75=139)
When the student needs to borrow from a column whose top digit is 0, he skips that column and borrows from the next one.

10 students used: DIFF/*N*−0=0 (403−208=105)
Whenever the bottom digit in a column is 0, the student writes 0 in the answer; i.e., *N*−0=0.

10 students used: DIFF/0−*N*=*N* (140−21=121)
Whenever the top digit in a column is 0, the student writes the bottom digit in the answer; i.e., 0−*N*=*N*.

9 students used: DIFF/0−*N*=*N* and LEFT/TEN/OK (908−395=693)
When there is a 0 on top, the student writes the bottom digit in the answer. The exception is when the 0 is part of 10 in the left columns of the top number.

opposed to scoring it based *solely* on the number of right and wrong answers on his test. Even with the advances in criterion referenced testing, it remains true that a test is simply scored by what problems a student gets right or wrong. Because of the embedded nature of most procedural skills, a student can get many problems wrong simply by having one fundamental underlying bug in a primitive subprocedure that he is using to solve different problems on the test. In such a situation, the score that a student gets can bear little relationship to the misconceptions that he actually harbors! A student can receive a low score either because he has many misconceptions, each one of which is more or less the top of the procedural network of skills he is using, or he may possess a few, or even just one, misconception that is deep down inside the internal workings of the procedural network and which is constantly being used to compute intermediate results used by higher-up subprocedures.

Current techniques for correcting tests do not offer easy and reliable methods of separating these two situations. The diagnostic modeling technique discussed above can take the answers that a student gives on a test and, through its modeling system, show not only which questions were answered incorrectly but *why* they were incorrectly answered. It is interesting to note that 107 of the 1325 students tested had a bug in their borrow-from-zero subprocedure and missed 6 out of the 15 problems on the test because of this one underlying bug. The characterization given by BUGGY is a much fairer evaluation than scoring these students 60% correct.

A Methodological Tool for Judging the Diagnostic Quality of a Test

The procedural network apparatus also provides a methodological tool for judging the quality or diagnostic capabilities of a given test. This allows one to talk about how well the test can discover and delineate common misconceptions. Given a test, each bug in the network can be used to answer all of the problems. The resulting "buggy" responses are then used to partition the bugs: two bugs are put in the same partition if they produce the same answers to all of the test problems. "Diagnosticity" of the test can now be defined in terms of the size of the resultant partitions. A diagnostically perfect test has every bug in a partition by itself. Those bugs in the same partition are undifferentiated by the test. Any bugs in the same partition as the correct answers are not tested for at all. Taking the test in Fig. 4, this system discovered that a student can have either of two bugs (BORROW/ACROSS/SAME or $N-N=1$/AFTER/BORROW) and still get 100% correct answers.

Such a system could provide professional test designers with a formal tool for establishing the diagnostic quality of a proposed test. However, our belief is that professional test designers have good intuitions about diagnostic tests. This belief was confirmed by running two standardized national tests through the 330 subtraction bugs. One of the tests left only one bug unexposed in 17 problems while the other left 4 bugs unexposed in 10 problems. The unexposed bugs were rare

for they were not even found in the Nicaraguan data. There is a difference, however, between exposing a bug and diagnosing it. *Exposing* a bug amounts to having at least one problem on the test in which the bug is manifested; *diagnosing* a bug requires having test problems which differentiate between it and every other bug.

Using the Artificial Intelligence paradigm of generate-and-test, it would be straightforward to use BUGGY as a diagnostic test generator. The problem generator must produce "interestingly" different problems. This generation can be done using important features of problems such as the number of times borrowing is necessary, or whether or not a zero appears in the top number. Sets of generated problems can then be filtered using the procedural network to identify bugs which are not diagnosed. From the bugs left undiagnosed, features can be retrieved which direct the generation of alternative problems to be added to the test. In this way, a highly diagnostic test can be developed. Furthermore, since the answers that would be generated by using the bugs are known, the test could be a multiple choice test and still maintain its total diagnostic property! Similarly, a real-time adaptive testing system could be created based on these tools.

4. FUTURE RESEARCH

This paper has presented some of the problems that must be faced in diagnosing failures in procedural skills, and has described some ideas about the formulation and implementation of diagnostic modeling techniques that address these problems. It has also presented some novel uses of diagnostic models as a gaming/instructional device, as a deep-structure test grader, and as a tool to judge the diagnostic quality of a test or a set of problems. The central idea underlying this research is the use of a *procedural network* as a means of representing diagnostic models. The critical properties of this representation scheme are its abilities to represent an appropriately structured breakdown of a skill into subskills, to make explicit the control structures underlying a collection of skills, and to make the knowledge encoded this way directly executable. Such a representation enables a particular subskill to be easily modified and then simulated or executed so that the ramifications of the modification can be quickly ascertained. The structure of the network becomes important not only because it allows efficient modification but also because the representation of the modification can be used to contain explanatory or remedial material.[23] In addition, the structure allows certain types of control structure failures to be directly represented in the network and hence articulated—if necessary.

[23]Contrast this with the (admittedly strawman) technique of randomly switching instructions in a machine language program which carries out a skill. Even if a student's behavior could be duplicated, the resulting "model" would be worthless as an explanatory device or as an aid to remediation.

Relationship of Diagnostic Models to Other Kinds of Structural Models

We now turn to a brief look at the past and current work on structural models of students and how they relate to diagnostic models based on procedural networks. Most of the past and current research on this subject has been focused on the intuitively appealing notion that if one has an explicit, well formulated model of the knowledge base of an expert for a given set of skills or problem domain, then one can model a particular student's knowledge as a simplification of the rules comprising the expert's procedures (Barr, 1974; Brown, 1974; Brown & Burton, 1975; Burton & Brown, 1977; Carbonell & Collins, 1973; Carr & Goldstein, 1977; Collins, Warnock, & Passafiume, 1975). Recently, Goldstein has expanded this concept in his Computer Coach research and has coined the term "overlay model" for capturing how a student's manifested knowledge of skills relates to an expert's knowledge base (Goldstein, 1977).

The work reported in this paper differs in that the basic modeling technique is based on viewing a structural model of the student not as a simplification of the expert's rules but rather as a set of *semantically meaningful deviations* from an expert's knowledge base.[24] Each subskill of the expert is explicitly encoded, along with a set of potential misconceptions of that subskill. The task of inferring a diagnostic model then becomes one of discovering which set of variations or deviations best explains the surface behavior of the student. This view is in concert with, although more structured than, the approach taken by Self (1974) in which he models the student as a set of modified procedures taken from a procedural expert problem-solver.

Another closely related approach to modeling a student's knowledge base uses a production rule encoding scheme (Smith & Sleeman, 1977; Young, 1977). However, procedural networks differ both theoretically and computationally from these efforts in that they are designed to make explicit the representation of the control-structure knowledge underlying a macro skill so that it can be efficiently diagnosed and explicitly tutored.

In the remainder of this section, we present our view of the more promising directions for research relating to diagnostic models.

Extensions to the Gaming Environment

In the second section, we described the BUGGY game which was designed to introduce the notion of "buggy" behavior and provide practice in diagnosing it. Although this activity was initially designed for training student teachers in diagnosing and articulating procedural bugs, it has also been used as an activity

[24]Because these deviations are based on both the student's intended goals and the underlying teleology of the subskills, we have no automatic way to generate them (as opposed to what could be done if the deviations were based on the surface syntax of the rules). However, ongoing work by the authors, as well as Miller and Goldstein (1976) and Rich and Schrobe (1976), is directed toward helping to overcome this limitation.

to get kids to introspect on their already known procedures as well as to encounter the concept of bugs and debugging strategies in an easily grasped context.

A limitation of the current gaming environment is that most of what the players learn while using the game they learn or discover on their own. At the moment, BUGGY does no explicit tutoring; it simply provides an environment that challenges their theories and encourages them to articulate their thoughts.[25] The rest of the learning experience occurs either through the sociology of team learning or from what a person abstracts on his own. The next step in realizing the educational potential of the BUGGY game is to implement an intelligent tutor which can recognize and point out weaknesses (or interesting facets) in a student's debugging strategies.[26] Our experiences indicate that such a tutor would be very helpful for middle school and remedial students who often get caught in unproductive ruts. The tutor could also help focus the student's attention on the structure of the arithmetic procedures themselves. It is worth noting that *some* of the tools for constructing an intelligent tutor for the BUGGY game already exist in the form of the test validation techniques described in the previous section. Nevertheless, these techniques do not provide the right kind of information for explaining to a student *why* the problem he just generated had little or no diagnostic value. We are currently exploring the kind of reasoning required to answer "why."

An intelligent tutor designed specifically to help teachers will profit from a theory of what makes an underlying bug easy or difficult to diagnose. Simple conjectures concerning the depth of the bug from the surface do not seem to work, but more sophisticated measures might. It is hard to see how to predict the degree of difficulty in diagnosing a particular bug without a precise information-processing or cognitive theory of how people actually formulate conjectures about the underlying bug or cause of an error. For example, one such theory is that people walk through their own algorithm, looking for places where a part of the incorrect answer is different from their own and then try to imagine local modifications to their algorithm that could account for the error. Under this theory, one would expect bugs that involve major modification of the procedure, such as changing the direction in which columns are processed, to be difficult to diagnose. Similarly one would expect difficulty if a student's algorithm differs from the diagnostician's. Given an adequate theory, the difficult situations can be watched for and corrected through appropriate tutorial comments during diagnostic training.

[25]As a historical footnote, BUGGY was originally developed to explore the psychological validity of the procedural network model for complex procedural skills. During that investigation we realized the pedagogical potential of even this simple version of BUGGY as an instructional medium.

[26]For examples of the types of tutoring, see Brown and Burton (1975, 1977), Carr and Goldstein (1977), and Goldstein (1974).

Another extension to the BUGGY environment to encourage further exploration of the ideas of hypothesis formation and debugging is the development of a specialized programming language for writing simple arithmetic procedures. Actually having students write procedures provides immediate focus on debugging strategies—a topic usually left until the end in most secondary school programming courses. In this limited environment, it should be possible to construct an intelligent programming assistant as well as a computer-based tutorial helper that can aid a student when he gets stuck. Providing students with a language in which they can write their own procedures also allows use of a game developed in the SOPHIE environment (Brown, Rubinstein, & Burton, 1976), where one student writes a procedure introducing a bug and another student tries to discover it by presenting test problems.

Extensions to the Diagnostic System

Concerning the use of procedural networks as a tool for diagnosing real students, we reiterate that the capabilities of the present system are solely diagnostic: no tutoring is attempted. The issue of what tutorial strategy to use, even when it is known exactly what a student is doing wrong, is still an open question.

One possible strategy is that the "expert" portion of the procedural network could be made *articulate* in the sense of being able to explain and justify the subprocedures it uses. Since the system would know the student's bug, special problems can be chosen and special care can be taken while presenting the steps which illustrate the student's bug. This feature could also be used to allow students to pose their own problems to the system and obtain a running account of the relevant procedures as the "expert" solves the problem. A useful notion for the articulate expert may be to have additional explanation or justification of each symbolic procedure in the network expressed in terms of a "physical" procedure using manipulative tools such as Dienes' blocks (Dienes & Golding, 1970). In this way, the execution of each *symbolic* procedure could cause its analogous *physical* procedure whose execution could be displayed on a graphics device, thereby letting the student see the numeric or abstract computation unfold in conjunction with a physical model of the computation. This approach directly attacks the problem of getting procedures to take on "meaning" for a student: the acquisition of meaning is, we believe, accomplished by recognizing mappings or relations between the new procedures and existing procedures or experiences.

While we consider the articulation of the expert to have great promise as a corrective tutorial strategy, it is by no means the only possible such strategy. It is possible that with certain bugs (and certain students), a clear description of what is going wrong may be sufficient to allow the student to correct his problem. Or it may be possible to formulate a series of problems (possibly in conjunction with physical models) that enables the student to discover his own error. Or it may be best to abandon his algorithm (rather than trying to debug it) and start over with a different, simpler algorithm to build the student's confidence.

Generalization to Strategic Knowledge

Caution should be exercised in generalizing the procedural network model to other procedural skills. In particular, the aspects of knowledge discussed here are almost totally algorithmic in nature containing little heuristic or strategic knowledge in selecting or guiding the execution of the primitives (procedures). Many mathematical skills involve an interplay between strategic and algorithmic knowledge. For example, when adding two fractions, one does not necessarily compute a common multiple by multiplying together the two denominators. Instead, an examination is made of the relationships between the two denominators such as identity, obvious divisibility of one by the other, relative primeness, and so on. On the basis of such relationships specialized procedures are chosen for adding the given fractions. The rules underlying these decisions can have their own bugs and, therefore, these rules must be modeled within some representation scheme. Although procedural network schemes can represent such decision rules, we believe that other schemes, based perhaps more on annotated production rules (Goldstein & Grimson, 1977), deserve serious consideration. We are currently investigating a hybrid approach for modeling fractions (and their bugs) that involves merging annotated production rules with procedural networks.

Psychological Validity

An important area for exploration concerns the psychological validity of the skill decomposition and buggy variants in the network. That is, how well do the data structures and procedure calls in the network correspond to the structures and skills that we expect people to learn? From the network designer's point of view, the psychological validity can be improved or denigrated by choosing one structural decomposition instead of another. Determining a psychologically "correct" functional breakdown of a skill into its subskills is critical to the behavior of gaming and diagnostic systems using it. If the breakdown of the skill is not correct, common bugs may be difficult to model while those suggested by the model may be judged by people to be "unrealistic". For people playing BUGGY based on a nonvalid network, the relationship between its bugs and the behavior they observe in people will often be obscure. Measuring the "correctness" of a particular network is a problematic issue as there are no clear tests of validity. However, issues such as the ease or "naturalness" of inclusion of newly discovered bugs and the appearance of combinations of bugs within a breakdown can be investigated.

Finally, we have left open the entire issue of a semantic theory of how procedures are *understood* (and learned) by a person and why bugs arise in the first place. The need for a theory of how procedures are learned correctly or incorrectly is important for at least two reasons. First, an interesting theoretical framework that accounts for the entire collection of empirically arrived at bugs will undoubtedly provide insight into how to correct the teaching procedure that

produced the bugs in the first place. Second, such a theory would be the next step in a semantically based generative theory of student modeling. As we have stated earlier, bugs have to be hand-coded into the network now. One can envision generatively producing bugs by a set of syntactic transformations (additions, deletions, transpositions, etc.) based on some appropriate representation language. While some bugs can be naturally accounted for as "syntactic" bugs, others, such as inappropriate analogy from other operations or incorrect generalization from examples, are best explained outside of the representation itself and hence require a "semantic" theory. One of the by-products of the diagnosis of the student data mentioned in the previous section has been a thorough and precise catalogue of bugs arising in one particular skill, subtraction. This network can now be used to suggest and evaluate theories about the origin of bugs.

APPENDIX 1: STUDENT RESPONSES TO BUGGY

In an experiment described in Section 2, a group of student teachers were exposed to the BUGGY game. This appendix provides a list of responses to the question, "What do you think you learned from this experience?"

I see from this system that you learn from your mistakes. In a certain operation there are so many mistakes that you can make. When you learn what the mistakes are you learn to do the operation correctly.

That children's errors can be a way of diagnosing the way the child learns material. Also it raises questions about the way a child is tested both standardized and informally.

A student's errors and/or misunderstanding of a concept may have not been due to carelessness but rather involved a complex and logical thought process.

I learned that it is necessary to try many different types of examples to be sure that a child really understands. Different types of difficulties arise with different problems.

Although it's hard to tell from these pre- and posttests, in the middle is learned a great deal about the complexity of student's errors. I know that young students can get these preconceived notions about how to do things and it's very hard to find a pattern to their errors though it exists and I believe that BUGGY convinced me of [it].

That if you study the errors long enough you can *eventually* come up with a reasonable solution as to why the [error] is occurring.

Through looking carefully at children's math errors it is sometimes possible to discover a pattern to them. This pattern will tell you an area or a concept the child does not understand.

I learned that there could be more to a child's mistakes other than carelessness. Working with children with special needs I have encountered many such problems, yet never stopped to analyze what could be a systematic problem—for this I thank you.

Children do have problems and they are very difficult to spot especially when a number of different operations are used to come to an answer. I've learned to be more aware of how these children reach these "answers" and to help them to correct them; first by knowing how they arrived at the answer.

Although many arithmetic errors may be careless, there may also be a pattern that the kid is locked into. If you pick up on a pattern you can test the child to see if he/she conforms to it and work on it from there.

I found that I have looked closer at the problems, looking for a relationship between the set after working with BUGGY.

How to perceive problems, that don't look too consistent, a little easier.

I learned and was exposed to the many different types of problems children might have. I never realized the many different ways a child could devise to create his own system to do a problem. I am now aware of problems that could arise and I'm sure this will help me [in] my future career as a teacher.

How to more effectively detect "problems" students have with place value.

I have learned several new possible errors students may make in computation. I have also learned somewhat how to diagnose these errors, i.e., what to look for, and how specific errors can be.

I learned about diagnosing math difficulties. It makes me aware of problems that children have, and they sometimes think logically, not carelessly as sometimes teachers think they do.

That there are many problems that you can diagnose about a child by looking at his homework.

If a child has repeatedly made [the] same mistakes, it is more easily identified if the teacher has an opportunity to try and make [the] same mistakes. This method can be solved at least quicker than . . .

Tuned in to picking up malfunctions in simple addition and subtraction which seemed to be realistic problems.

ACKNOWLEDGMENTS

We are especially indebted to Kathy M. Larkin for her extensive assistance on this project and in particular for her contribution in refining the BUGGY activity and in discovering many new arithmetic "bugs."

REFERENCES

Barr, A. A rationale and description of the BASIC instructional program. Psychology and Education Series, Stanford University, Technical Report 228, April 1974.

Brown, J. S. Structural models of a student's knowledge and inferential processes. BBN Proposal No. P74-CSC-10, Bolt Beranek and Newman, Cambridge, Massachusetts, April 1974.

Brown, J. S., & Burton, R. R. Systematic understanding synthesis, analysis, and contingent knowl-

edge in specialized understanding systems. In D. Bobrow & A. Collins (Eds.), *Representation and understanding: Studies in cognitive science*. New York: Academic Press, 1975.

Brown, J. S., & Burton, R. R. A paradigmatic example of an artificially intelligent instructional system. Presented at the First International Conference on Applied General Systems Research: Recent Developments and Trends, Binghamton, New York, August 1977.

Brown, J. S., Collins, A., & Harris, G. Artificial intelligence and learning strategies. To appear in H. F. O'Neil (Ed.), *Learning strategies*. New York: Academic Press, 1978, in press.

Brown, J. S., Rubinstein, R., & Burton, R. R. Reactive learning environment for computer assisted electronics instruction. BBN Report No. 3314, A. I. Report No. 1, Bolt Beranek and Newman, Cambridge, Massachusetts, October 1976.

Brown, J. S., Burton, R. R., Hausmann, C., Goldstein, I., Huggins, B., & Miller, M. Aspects of a theory for automated student modelling. BBN Report No. 3549, ICAI Report No. 4. Bolt Beranek and Newman, Cambridge, Massachusetts, May 1977.

Burton, R. R., & Brown, J. S. Semantic grammar: A technique for constructing natural language interfaces to instructional systems. BBN Report No. 3587, ICAI Report No. 5, Bolt Beranek and Newman, Cambridge, Massachusetts, May 1977.

Burton, R. R., & Brown, J. S. A tutoring and student modeling paradigm for gaming environments. In *Proceedings of the Symposium on Computer Science and Education*, Anaheim, California, February 1976.

Carbonell, J., & Collins, A. Natural semantics in artificial intelligence. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, 1973.

Carr, B., & Goldstein, I. Overlays: A theory of modeling for computer aided instruction. Massachusetts Institute of Technology, AI Memo 406, February 1977.

Collins, A., Warnock, E., & Passafiume, J. Analysis and synthesis of tutorial dialogues. In G. H. Bower (Ed.), *The psychology of learning and motivation*, Vol. 9. New York: Academic Press, 1975.

Cox, L. S. Systematic errors in the four vertical algorithms in normal and handicapped populations. *Journal for Research in Mathematics Education*, 1975, **6**, 202–220.

Dienes, Z. P., & Golding, E. W. *Learning logic, logical games*. New York: Herder & Herder, 1970.

Easley, J. A., Jr., & Zwoyer, R. E. Teaching by listening - toward a new day in math classes. *Contemporary Education*, 1975, **47**, 19–25.

Friend, J. Description of items to be used in addition/subtraction tests. Internal Memo, Institute for Mathematical Studies in the Social Sciences, Stanford University, 1976.

Goldstein, I. Understanding simple picture programs. Massachusetts Institute of Technology, AI Laboratory, Technical Report 294, September 1974.

Goldstein, I. The computer as coach: An athletic paradigm for intellectual education. Massachusetts Institute of Technology, AI Memo 389, February 1977.

Goldstein, I., & Grimson, E. Annotated production systems. A model for skill acquisition. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, August 1977.

Krauss, R. M., & Glucksberg, S. Social and nonsocial speech. *Scientific American*, 1977, **236**, 100–105.

Matz, M. Some issues on building an Articulate Expert for high school algebra. BBN Report, ICAI 10 (in preparation), 1978.

Miller, M., & Goldstein, I. Overview of a linguistic theory of design. Massachusetts Institute of Technology, AI Memo 383, December 1976.

Rich, C., & Schrobe, H. E. Initial report on a LISP programmer's apprentice. Massachusetts Institute of Technology, AI-TR-354, December 1976.

Rubin, A. Hypothesis formation and evaluation in medical diagnosis. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, AI-TR-316, January 1975.

Sacerdoti, E. *A structure for plans and behavior*. The Artificial Intelligence Series. New York: Elsevier North-Holland, 1977.

Searle, B., Friend, J., & Suppes, P. *The radio mathematics project: Nicaragua 1974–1975*. Institute for Mathematical Studies in the Social Sciences, Stanford University, 1976.

Self, J. A. Student models in computer-aided instruction. *International Journal of Man-Machine Studies*, 1974, **6**, 261–276.

Smith, M. J., & Sleeman, D. H. APRIL: A flexible production rule interpreter. SIGART Newsletter No. 63, June 1977.

West, T. Diagnosing pupil errors: looking for patterns. *The Arithmetic Teacher*, 1971, 64.

Young, R. M. Mixtures of strategies in structurally adaptive production systems: Examples from seriation and subtraction. SIGART Newsletter No. 63, June 1977.