# Tutorial Diagnosis of Substraction Errors

Ross Bryant

Department of Computer Science
University of Waterloo
Waterloo, Ontario
N2L 3G1

Research Report CS-86-09

# Tutorial Diagnosis of Subtraction Errors

*Ross Bryant*
*University of Waterloo*
*Department of Computer Science*
*Essay Requirement for MMath Degree in Computer Science*
*Supervisor: Marlene Jones*
*January 31, 1986*

*ABSTRACT*

The Buggy system (Brown and Burton) for diagnosing student errors in subtraction is reviewed. It is argued that more direct, precise and comprehensive diagnosis could be performed by an interactive automated tutor which is capable of monitoring detailed solutions input by the student on the screen. A design for such a tutor, including other tutorial functions, is proposed. The educational issues involved in the design are discussed, in particular the role of a system-maintained student model. The design issues are extended to a consideration of more complicated arithmetic skills and some non-mathematical domains.

## 1. Introduction

In this essay, methods of diagnosing children's errors in subtraction are considered. First the Buggy Model of Brown and Burton is reviewed. It is argued that (1) it is overly complicated and that (2) it is not able to diagnose factual or careless errors. This motivates a design proposal for a Subtraction Tutor which can diagnose errors interactively in the same way a human tutor would. Four modes of operation of this system are discussed, along with the purpose and use of a system-maintained student model. Related design issues such as locus of control and automatic bug generation are discussed. More complicated arithmetic domains such as integer division and operations with common fractions are analyzed to show the benefits of the tutorial approach as well as the special difficulties they present. The conclusions are:

[1] Tutoring is a better method for detailed diagnosis.

[2] Subtraction is an appropriately simple domain in which to develop many of the issues in tutorial strategy.

## 2. The Buggy Model

In this section, a description and evaluation of the Buggy Model of Burton and Brown [Brown and Burton 78] is presented. This model has been influential in CAI research and is worth examining in some detail.

### 2.1. Motivation

Brown and Burton originally proposed the "Buggy" model to account for student's errors in simple procedural skills. In this model, observed student errors are explained as symptoms of bugs, that is, deviant versions of the correct skills. Integer subtraction was chosen as the test domain because it is simple enough to model but complex enough to demonstrate the problems associated with diagnosis. Most errors, it is claimed, arise from bugged algorithms as opposed to carelessness, inattention, etc. To represent a skill such as subtraction, the goal of the skill is broken into subskills. When implemented, goals become calls to other subgoals. Bugs are then represented as variants to nodes in the network. Thus, a student's behaviour can be predicted by a network which is a combination of correct and incorrect subskills.

The belief is that if a student's incorrect knowledge can be diagnosed, it is much easier to prescribe precise remediation. It is also much easier to make the student understand what he is doing wrong if the faulty procedure is pointed out to him. As a simple example, some students when confronted with a problem in which a zero occurs in the minuend will use the buggy rule "0 - n = n" as an alternative to borrowing:

```
 500     312
- 65    -243
----    ----
 565     149
```

In this example [Burton and Brown 1982], the student borrows correctly when the minuend is non-zero, but uses the buggy rule when the minuend is zero or has become zero due to borrowing. Other common bugs include:

- 0 - n = 0
- adding instead of subtracting
- smaller from larger
- when borrowing from a column with a zero as the top digit, decrement the bottom digit of the column instead
- when borrowing from a column with a zero on top, forget about the decrement operation

## 2.2. The Buggy Diagnostic System

A naive version of the eventual system compares the answers predicted by all bugs with the answers produced by the student. The buggy algorithms which produce the same answers as the student are then selected as likely hypotheses. This set can be reduced by subsequent testing which discriminates between competing hypotheses. Originally the initial set of bugs was small, so an exhaustive search could be done. However, as the bugs of more and more students were added, the numbers swelled to 110 "simple" bugs and 20 common compound bugs. Testing for combinations of this basic repertoire became very expensive. Moreover, the system had to handle "noisy" data resulting from careless errors and inconsistent use of a bug. Other sources of noise are:

[1] Performance Lapses. Students can make mistakes from fatigue, boredom, etc. when following a correct as well as a buggy algorithm.

[2] Errors in primitive subskills. Math facts (eg. 13 - 7 = 6) are taken as primitive (undecomposed) skills. Burton [Burton 82] does not break these into 100 subskills corresponding to the 100 basic subtraction facts because they are not testable on the average diagnostic test, which includes around 60 distinct facts. More important, such a breakdown would swamp the system, and every error could be explained as a combination of factual errors and/or performance lapses.

[3] Hiding. One bug may "hide" another and prevent it from manifesting itself.

[4] Compound bugs. Bugs may occur in pairs. Some students have been observed with up to 4 bugs in a single skill. Even to search for pairs of bugs in 130 bugs is prohibitive. Also, pairs of bugs can hide simple bugs. (Burton does not mention the problem that two bugs may interact in such a way as to produce a correct answer, because if the bugs were already included in the hypothesis set, they would (correctly) predict the student's getting the correct answer.

The foregoing concerns motivated DEGUGGY, the first implementation of the system.

## 2.3. DEBUGGY

For this operational offline system, 130 buggy algorithms, each consisting of the correct algorithm with buggy variants selected from a fixed set of 110 primitive bugs and 20 common compound bugs, are each given the diagnostic test, and the results are compared with those of the student. An initial hypothesis set is constructed in which each bug explains at least one wrong answer. Heuristics are then used to determine which combinations of the initial bugs will be considered as additional candidates for the hypothesis set.

Burton does not include what he calls "single column evidence" when forming the hypothesis set. Using this scheme would involve adding a bug to the set if it explained a single incorrect column of the student's answer. For example [Burton 82], the bug "0 - n = 0" would explain the 10's column error in:

```
 303
-218
----
 105
```

In the first place, this would greatly expand the number of bugs to be considered. In the second we need to be sure that this "local" theory is consistent with the whole solution. For example, in the example shown, the student might not have had the bug "0 - n = 0": he might have borrowed in such a way (perhaps even the correct way) that he did not think that the 10's column involved a (0 - 1) subtraction. Using "0 - n = 0" we cannot explain the whole answer 105, since if no other bug were present the 10's column would have been (9 - 1) after borrowing. Thus, Brown uses global evidence to constrain the growth of the hypothesis set.

Before considering combinations of initial hypotheses, bugs are removed under the following circumstances:

[1] A bug produces the same error as another bug for which there happens to be other evidence. For example (700 - 5 = 705) can be explained by both "smaller from larger" and "adding instead of subtracting". If there were evidence for the first rule already, we would keep the first rule but remove the second.

[2] A bug which is a specialization of another is removed. For example, "smaller from larger" is more general than "0 - n = n".

[3] Some combinations of bugs must be rejected. For example, if the student has the bug "smaller from larger", no borrowing procedure will ever be used, so we would not compound one with, for example, "smaller from larger".

[4] Sometimes one member of a pair of bugs possessed by a student will not have independent evidence for its existence. For example, suppose the student has "0 - n = 0" compounded with "when borrowing from a column with 0 as the top digit, decrement the bottom digit of the column instead". We will never see evidence of the second since the result will always be 0.

The limit of compounding is 4 bug combinations. Now all the bugs in the hypothesis set are used on the test and compared with the student's results. Where they differ from the student test due to noise, "coercion" is used to see if the bug can be made to fit better. This involves making small extra bug assumptions such as allowing subtraction errors where the amount of error is 1 or 2. Finally the bugs are compared with one another according to how many right and wrong answers were predicted and how many results were predicted incorrectly. The highest scoring theories are compared and the best is chosen using simplicity as a criterion.

DEBUGGY was tested extensively on data for approximately 1200 children in Nicaragua [Brown and Burton 78]. It was observed that nearly 40% of the children tested exhibited consistently buggy behaviour.

## 2.4. IDEBUGGY

IDEBUGGY is an interactive online version of DEBUGGY. Since it is interactive, it is not limited to interpreting the results of a single, fixed diagnostic test. A small number of initial problems is analyzed first, and bug hypotheses are formed. New problems are generated for the student to solve. This new evidence then provides support for one subset of the hypothesis set over its complement. New hypotheses based on errors in current questions are added, and the process is repeated until a satisfactory diagnosis is obtained. In this interactive version, the questions produced by the question generator can be checked to see whether they will in fact distinguish bugs which are in the set of current hypotheses.

## 2.5. Subskill Lattice

The Debuggy model is used by Burton to provide an elegant definition of subskill which is not limited to components of the correct skill. The primitive bugs are applied to the set of all possible test problems, resulting in a partition of the set for each bug. This procedure sorts bugs into equivalence classes based on the partitions they produce. With each equivalence class is associated a subskill which can be defined as "any isolatable part of a skill that it is possible to mislearn". [Burton 82, p. 127] The subskills form a lattice partially ordered by the relationship "gets correct answer on all the same problems and more". The lattice so produced contains 58 subskills, a large number for such an elementary skill.† Many of the subskills are ones we would take for granted, for example, #25, "Borrow from columns that have a top digit one less than the bottom digit." We take these subskills for granted, according to Burton, because we do not normally think of them as decision points in the algorithm.

The definition of subskills is independent of the representation of the bugs because subskills are defined by a unique partition of the test questions, independently of how the partition is produced. (It is almost better to regard the subskill lattice as a behavioural classification than as a cognitive one.) The subskill lattice not only displays a fine-grained analysis of subskills, but also shows their relations, in particular, how one bug can hide another. That is, a subskill at a higher level which is not mastered will produce symptoms of all the lower level bugs to which it is connected in the lattice.

One of the main benefits of the lattice is that it can be used to analyze diagnostic tests. The set of bugs which generates the lattice can be used to partition the test. There is a homomorphism from the maximal lattice (where the test would include all possible questions) onto the test lattice. Subskills in the maximal lattice which map onto the same node into the test subskill lattice are therefore ones which are not distinguished by the test. Thus the test can be analyzed precisely to determine which skills it actually tests.

† See Appendices 1 and 2 for the lattice and definitions of subskills.

## 2.6. An Evaluation of the Buggy Model

Buggy is undeniably a very elegant model, and has a detailed implementation. Its complexity is disturbing, however, for at least two reasons.

### 2.6.1. How Complicated is Subtraction?

If as simple a skill as integer subtraction is as complex as its subskill lattice indicates, then more complicated skills must be overwhelmingly difficult to deal with in this manner. One might even be inclined to argue that there really **can't** be 58 subskills in subtraction. If they are not part of the repertoire of professional teachers of the subject, we may well wonder whether they might simply be cooked up. An examination of the origin of the list of bugs (which is what generates the subskills) [Brown and Burton 78] shows that bugs were added to the list every time it was necessary to explain the systematic errors of some individual student. Even the most common bugs (ones which teachers are likely to recognize), such as "0 - n = n", were only present in approximately 5% of the students tested. Some of the more exotic ones were present in only one student out of the 1300 or so who were tested.

It seems unreasonable to regard such unique behaviours as indicating a sub-skill. Perhaps these individual variations are better classified as variations in the data context in which a subskill is applied. For example, the subskill associated with borrowing involves knowing that you can borrow only if the top number is non-zero. If it is zero, you need to borrow from the next column etc. Indirectly, it involves knowing that the bottom number is irrelevant. A student may be thrown off by this extraneous information. That is, generally he knows what to ignore, but in some cases, the actual digits cause him to err. As an example, consider rule # 25: "borrow from columns that have a top digit one less than the bottom digit". A student who has this bug presumably borrows correctly except when this particular pattern is present. But rather than view borrowing correctly when the top is one less that the bottom as a separate skill, it seems more reasonable to say that the skill involved is knowing that when you borrow from a column, the bottom number is **always** irrelevant, regardless of what the number happens to be or what its relationship is to the top number. So in some contexts (i.e., combinations of top and bottom numbers), a student may be thrown off the correct application of a rule, but that is not a good reason for raising this definable situation to the status of a subskill. This practice reduces to absurdity if we include a rule for **every** combination of digits, for example, borrowing when the top is 3 and the bottom is 4, borrowing when the top is 3 and the bottom is 5, etc. This is not what Brown and Burton do, of course. Their bugs (and therefore potential subskills) are restricted to systematic errors which are actually observed to occur in students. But perhaps the statement of skills should not include reference to constants which are not explicitly part of the algorithm. Note that in the decomposition algorithm for subtraction, zero is a special case: it cannot be borrowed from. Consequently, we expect that the subskills for this algorithm will make reference to zero, but there is no need to make reference to other digits since they are all treated the same by the algorithm.†

To summarize, in Debuggy, complete diagnosis was strived for at the price of a very complicated system.

### 2.6.2. Can errors be handled more directly?

Even in the interactive version, IDEBUGGY, there is no **direct** way of handling errors due to carelessness or incorrect math facts.†† The approach is completely statistical and is based on the complete answer to a question.

We might compare this approach to that of a classroom teacher. The offline version, DEBUGGY, is analogous to a teacher's activity of marking tests out of class. If several errors are noticed, the teacher may look for a pattern. (Notice that the teacher usually has the benefit of seeing the student's scratch work, in particular, the indications of borrowing. DEBUGGY does not.) If a bug is obvious, the teacher may note it on the student's test. If it is not, he would not sit for hours doing a statistical analysis with 130 possible bugs. He would probably

† This is also the opinion expressed in [Young and O'Shea 82].

†† This problem is especially difficult to deal with on diagnostic tests used to determine what needs to be covered in a review of a skill. Often a student will "remember" a skill as he proceeds through the test. Consequently, the problem of handling inconsistent responses is compounded with the need to decide whether the student knows the material or not.

wait until the following day and tutor the student interactively. In such a session, the teacher can watch the student do problems and query him as to what he is doing at any given time. There are two main benefits to interactive tutoring as a means of obtaining a diagnosis:

[1] Since the process is "real time", every step the student takes can be observed in the order in which it occurs. Many operations which are not observed in the the final answer are made explicit, eg, the borrowing steps, the amount of time taken for each step, and so on.

[2] Since the process is interactive and the student is a person, the teacher can simply **ask** the student to describe the rules he is using or why he decided to do something. The teacher is not limited to doing scientific experiments with incomplete observations.

Usually this diagnostic function will be combined with various forms of remediation, for example, providing examples, working through examples, and assigning further questions. The point to note is that diagnosis in the tutoring context is not especially difficult, and that the reason it is not is that the details of the student's solution are available for inspection.

The interactive system IDEBUGGY, does not use tutorial diagnosis as a model, but instead continues the scientific experiment model, adding to it the tailoring of test questions to the data at hand.

In light of the above analysis, a design is presented for a tutoring system which could do more accurate and comprehensive diagnosis than Debuggy and do it in a more straightforward way. Such a system could also perform many of the other functions of a human tutor.

## 3. The Subtraction Tutor

In this section the design of a system called the Subtraction Tutor is presented. Its main features and modes of operation are described. This basic description and implications for student modelling are elaborated in the next section.

## 3.1. Screen I/O

The key feature of the Tutor is that the screen should simulate the scratch area of the student and that the Tutor should monitor the student's work. The problem can be set up on the screen as a conventional vertical subtraction, with the following features available.

[1] The student should be able to choose the location of his entries by moving the cursor, perhaps with a mouse. In this way, errors in location can be detected.

[2] Borrowings can be indicated by locating the cursor over the number to be decremented and using some form of command, for example, typing 'b' for borrow. Thus, borrowing from the wrong number can be detected.

[3] The resulting digit can be shown by moving the cursor above and entering the decremented value. This will show whether the decrement operation is erroneous or omitted.

[4] The new minuend (with carry added) can be entered above the old minuend, indicating correct location and addition.

[5] The cursor can be moved to the correct answer location and the answer digit entered. Thus location and factual errors can be detected.

This process can be repeated until the student signifies in some way that he is finished. As long as the system can keep track of all the screen I/O, it should be able to detect precisely what errors have been made. Consequently, it ought to be able to diagnose any error in an answer or in an intermediate step by considering a very small number of bugs, namely the set of bugs which explains errors in a particular step. Thus much of the complexity and sophistication of the full-blown Debuggy model can be eliminated, and replaced with what is essentially a programming problem, albeit a potentially tricky one.

## 3.2. Tutorial Modes

With this basic screen management capability, four useful modes of operation can be identified. Hopefully, these modes are a natural parallel to human tutor functions.

[1] DEMONSTRATION: In this mode the system calls upon its expert component to demonstrate how column subtraction is done. The expert can use a set of production rules such as those presented in [Young and O'Shea 82].† Questions can be from the system or from the student. The system can display the entire algorithm on the screen and highlight the current step or simply display the rule for the current step. The student can control the speed by indicating when he is ready for the next step.

[2] GUIDED PROBLEM SOLVING: In this mode, the student solves the problem guided by the expert. The expert can prompt at the most detailed level, for example, requiring the student to explicitly choose whether a borrow is necessary. The system is in control, and the student can be stopped as soon as a mistake is made. The system can even supply the algorithm step for the student if he needs help. If steps are provided, the student is in effect trying to **interpret** (in the computer science sense of the word) the correct algorithm. Errors can be corrected in a variety of ways, and hypotheses can be generated to account for errors in the same manner as Debuggy, except that errors would relate to individual steps, not whole answers. This mode is more complicated than DEMONSTRATION because of the increased I/O activity and the extra functions it makes possible.

[3] UNGUIDED PROBLEM SOLVING: The student works a problem of his own choice or a system-supplied one. The system does no prompting, but it may intervene if an error is detected. In computer science parlance, this is almost†† like putting a **trace** on the student's algorithm, possibly with **breakpoints** set corresponding to the entry of answer digits. This mode is necessary for initial diagnosis and for practice without prompts which prepares the student for independent problem solving..

† See Appendix 3 for a listing of these production rules. Young and O'Shea model incorrect student algorithms by combinations of deleting correct rules and adding incorrect rules. In this way they are able to account for all the common buggy variants of Brown and Burton with approximately 20 production rules, a considerable simplification.

†† The reservation is due to the problem that each unit of student output may not be the result of a single instruction. (Since we have no direct access to the student's "code", we cannot tell.) For this reason, it seems questionable to suppose that the tutor is **debugging** the student's algorithm, since this implies a comparison of the code with the output (or what would be worse for the analogy, change in internal registers.)

[4] EXPLANATION: Often, the most difficult part of tutoring (or teaching for that matter) is motivating an algorithm, that is, explaining to the student why the algorithm is the way it is and not the way the student may think it ought to be. In elementary school, subtraction is usually explained with physical objects. For example, poker chips of different colours can be used to represent place value notation. Borrowing can then be understood as "cashing in" chips of higher value in order to get chips of lower value so that subtraction (taking away) can be done. Simulating this† would require the graphics capability of representing suitable objects on the screen and performing the cashing operation. The nice feature of a system that could do this is that indefinitely many examples could be given, a particular example relating to the exact digits being used at the time. (Ordinarily, a student sees a few examples at most, and usually not when he is working on his own.)

## 3.3. Tutoring Strategy

The Tutor could be set to operate in one mode only, but if normal tutoring is to be simulated, it should be able to switch modes, either by student request or by its own tutorial strategy. If a student signs on and merely wants a demonstration of how to do a problem, then DEMONSTRATION mode alone is what is required, but if he needs major help, switching will probably be necessary. For example, the system may start off in UNGUIDED PROBLEM SOLVING mode in order to see what the student is doing wrong. If there is some difficulty in determining the student's bugs, it may switch to GUIDED PROBLEM SOLVING mode in order to "footprint"†† the student's ability to follow the algorithm. If he needs to see an example - if for instance he is not sure what the algorithm does in a certain case - a switch to DEMONSTRATION is called for. Perhaps he doesn't see why you need to perform some step. Then the system can switch to EXPLANATION. These kinds of switches can repeat under the

† This idea is also suggested in [Brown and Burton 78].

†† In this current educational metaphor, the "steps" a student goes through in the solution of a problem are compared to the footprints a person leaves as he goes towards his destination. Thus, to "footprint" a student is to see where he has been in the course of his solution. Compare this analogy to the literal stepping through of an algorithm in Papert's Logo when a student is advised to "play turtle" [Papert 80].

control of the system and possibly under student control as well, by offering a menu of over-riding choices to the student.

Clearly, what would be needed is a control structure with access to at least two main components, a Student Model and a Tutorial Strategy. The term "student model" refers to the system's representation of the student's knowledge, as well as other characteristics to be suggested later. (See the Introduction in [Sleeman and Brown 83] for a discussion of student models.) The term "tutorial strategy" refers to the system's strategy for dealing with a student's request for a particular service in light of the details of the student's responses to problems.

The above suggests that subtraction is a suitable domain for developing models of the tutoring process. Most current research into tutoring systems involves very complicated domains, for example:

- algebra: [Sleeman 82], [Bregar and Farley 83]
- electronic troubleshooting: [Brown, Burton and deKleer 82]
- programming: [Reiser, Anderson and Farrell 85]
- geometry theorem proving: [Anderson, Boyle and Yost 85]

Whereas subtraction can be dealt with by Young and O'Shea using only 20 production rules, the geometry theorem proving tutor of Anderson [Anderson, Boyle and Yost 85] requires approximately 700 rules. To be sure, subtraction does not involve strategy, but later in this essay it will be argued that strategy is introduced in integer division and reduction of common fractions, both of which are simpler than, for example, theorem proving. Subtraction is sufficient for introducing mode switching, problem choice and so on.

## 3.4. Extra Features

In addition to the tutorial strategy discussed, it would be useful to have an introductory session to explain the use of the system. The student should be made aware of cursor control, commands, and so on, as well as the modes of operation so that he can use the system effectively. Possibly some user information could be present on the screen throughout the whole session in a special reserved location. This is an important feature in that it helps to reduce error introduced by a student's lack of knowledge of how to use the system as opposed to his lack of knowledge of subtraction.[†]

## 4. The Use of Student Models in the Subtraction Tutor

In this section the features of the Subtraction Tutor which require the maintenance of a student model are identified. It is observed that useful systems could be built with no student model at all, although such models are necessary for advanced tutoring skills. The system described so far could be implemented in various combinations of modes. The following are some reasonable combinations.

### 4.1. DEMONSTRATION Mode Alone

This would be the most straightforward mode to implement. In DEMONSTRATION mode, the student could see examples of his own choosing worked out. He could use this to see how a particular problem is done or even to compare his answer to that of the expert, either to see what he did wrong or to see why what he did was right. On the other hand, he could use the system as a source of worked examples if he were just learning a particular skill.

An academically mature and motivated student could use the DEMONSTRATION mode as a programmed learning system. For example,we could imagine him stepping through the expert's proof but guessing ahead what the correct response would be. He could even try guessing what the correct algorithm step was going to be. (eg "the expert will have to see if it can borrow from the 100's column"). Using the system in this way is analogous to using a text book by answering questions and then checking the answers, but not "cheating" and

[†] This potential source of error was pointed out by Robin Cohen. The way to minimize this error is to ensure that the student has simple ways to undo incorrect input, for example by a "zap" command which can be used to cancel the latest input. By the same token, an "unzap" command can correct unwanted "zap" commands. Notice that by virtue of the fact that the screen represents the student's solution exactly as he would do it with pencil and paper, the system can be assured that if the student is happy with what he sees on the screen, then any errors must be in his understanding of the algorithm and not be due to a lack of understanding of the system. Compare this situation to one in which the subtraction details could not be represented graphically. Suppose, for example, that the student was forced to encode his steps in appropriate arrays. He might have to enter: "NewMinuend[3] <-- 2" in order to indicate that the value of the 100's column was now 2 as a result of a borrow. In such a case, errors such as reversing the digits might be common, and they would reflect errors in the student's input of information rather than of his understanding of subtraction. But with the graphical representation, these errors could not occur.

looking up the answer ahead of time. The upshot is that a system such as this would be useful in the classroom even though it "knows" nothing about the student, much less how to teach, and would exist purely as a **tool** to be **used** by a student. It would be particularly useful for students who need to see many examples, but who are too shy or intimidated to ask the teacher for extensive help. This is not to say that a more sophisticated tutor could not perform more sophisticated functions, but that lack of a managed student model does not make a CAI system ineffective.

## 4.2. GUIDED PROBLEM SOLVING Mode Alone

This mode would be useful for drilling the student in the use of the algorithm. If the algorithm is displayed, it gives the student practice at interpreting it. If it is not displayed, the student is forced to remember it and to execute it correctly. In this mode, the system is taking more control of the tutoring process, and preventing the student from merely looking at examples in a passive way: the system forces some kind of response from the student. The system would prompt for input from the student and compare this with its own answer. If the student is correct, it prompts for the next response (with the algorithm step shown possibly). If wrong there are two main alternatives discussed in the next two sections.

### 4.2.1. Drill-and-Practice Paradigm

If the system is designed to operate at a level analogous to typical drill-and-practice programs, it would indicate that an error had been made, allow the student a few more chances, then provide the answer if necessary, and continue. Note that even in this non-intelligent approach, the system ought to outperform conventional drill and practice programs because it at least shows which steps are incorrect (not just that the answer is incorrrect). It could even display the justification for the step. This ability to work through a problem in a human-like way is one of the prerequisites of a good tutor.

### 4.2.2. Intelligent Diagnosis

The system would display more intelligence if in addition it could diagnose student errors as well. This is a much simpler task than that facing DEBUGGY because the operation of subtraction has been broken into steps which are made explicit on the screen and therefore can be represented by the system. As an example, suppose the student believes he doesn't need to borrow in a given column. There are a limited number of possible bugs the student could have:

[1] He doesn't know the rule, "borrow if top less than bottom".

[2] He doesn't know which numbers are less than others. (inequality facts)

[3] He doesn't know that he is supposed to test to see if a borrow is necessary.

[4] He was careless, inattentive, etc.

In particular, the system does not need to worry that the current error concerning the need to borrow might be the result of a previous error because the system would already have detected this fact and presumably would have corrected it. For example, consider the erroneous solution:

```
 426
-127
 ----
 309
```

In this case the student apparently carried into the units column but forgot to decrement the 10's column. Consequently, the system, since it had already observed this, would have corrected the problem, and the student would be working with the correct digits in the 10's column, (1 - 2 = ?) Thus any errors, including careless ones, would be independent of previous ones. This allows us to use what Burton calls "single column evidence" without the danger of a previous error interfering.

Notice that the bugs can all be associated with the place in the algorithm where they apply, so if an error is noticed, some of them can be tested as possible hypotheses. For example, suppose a buggy rule is "borrow if top less than <u>or equal to</u> bottom". In this case, if top and bottom were the same, a student with this bug would decide to borrow instead of deciding not to borrow. If firing this buggy rule produces the same test decision as that of the student, then it could be added to the list of hypotheses. Unlike Debuggy, we could then simply ask the student which of the possible bugs, if any, he thought he had. In the context of Buggy, this is like bringing in the student as a consultant in his own diagnosis! But surely this is reasonable. After all, what is being diagnosed are the ideas in the student's mind, not, for example, the presence of bacteria in his tissues. Whereas a person is unlikely to know whether he has gonnococci in his bloodstream, he may well be aware of what his motivations are. Of course, he may not be aware as to why he didn't borrow when he should have, or he could even be mistaken. He may simply be confused about the whole process of subtraction. In any case, however, we need not take his introspective reports at face value, but merely add them to our evidence for given hypotheses. As a general educational point, it is valuable for students to be exposed to the detail of the diagnosis of their errors because it is important for them to ultimately be able to diagnose their own mistakes if they are to become fully independent learners. (Notice that medical patients are not supposed to become qualified practicioners as a result of their trips to the doctor.) If the system is unsure, it makes sense to ask the student.

The hope would be that by observations of common mistakes, the system develops a model of the student† which it can use at any time to make good guesses as to what the problem is, and to decide what area to teach next. The advantages of this capability are twofold:

[1] It may be easier for the student to correct an error if it is pointed out to him. It is not desirable to exagerate the usefulness of this. The main goal is to produce students who can subtract. If it turned out that straight drill in GUIDED PROBLEM SOLVING mode eliminated bugs automatically, there would be little use for a student model as a teaching aid.

/(dg Diagnosis could be done with the Theorist system [Poole 84] [Jones and Poole 85] [Poole, Aleliunas and Goebel 85]. The only reservation regarding its use is that it may be a more powerful tool than is needed. Theorist reasons by finding hypotheses (possibly defaults) which "explain" observations, in the sense that the observations are logical consequences of the hypotheses. It also checks that the hypotheses are mutually consistent and also consistent with the observations. We may not want to check for consistency, however, if we want to allow for inconsistent student behaviour. The choice of an hypotheses choosing mechanism depends therefore, on the details of the implementation.

[2] In the case where errors **persist,** a model which identified these errors would be extremely useful, either as forming the basis of a report to the teacher or for use by the system in determining tutorial strategy. Consider the following types of problems:

(a) If the student makes factual errors in subtraction, the system could switch to a program which provides drill-and-practice in math facts. Note that the program need not be computerized, but could consist of problems in a text book, math lab, or other paper based system. Moreover, statistics could be kept on the types of errors so that the drill and practice could be done efficiently. For example, the student may have difficulties with some numbers more than others, or only when the minuend is in the teens, and so on. Notice that this fine-grained analysis of factual bugs is beyond the capability of DEBUGGY because it only dealt with a single test and because the added computation was too great.

(b) The student may simply be careless. This can be detected by offering a student the chance to correct the results of a step. If the step is a yes/no one, then we cannot tell much, but if it is a subtraction step where there are 10 choices, a pattern of incorrect first answers and correct second answers would be evidence for carelessness. It is difficult for a computer to deal with carelessness, but it is extremely useful for a teacher to know that a student is not having a more fundamental difficulty.

(c) A very useful source of error for a system to detect would be learning disability type of errors such as number transposition. One way to detect this is by having an explicit transcription step wherein the student must use the cursor to set up the vertical subtraction problem after being given the problem in horizontal format. Errors in transcription could be recorded, indicating which numbers were typically transposed. In addition, buggy rules could be added to appropriate algorithm steps to account for mistakes due to transposition.

A report to the teacher concerning persistent errors would be extremely useful, particularly in the case of a learning disability. Even if the classroom teacher is unable to deal with such a disability, it could be referred to specialists. This approach has the added advantage of saving time by eliminating other hypotheses, such as carelessness, dullness etc.

### 4.2.3. Where does detailed diagnosis belong?

The diagnosis being suggested here is at least as detailed as that provided by DEBUGGY and probably moreso. If a tutoring system is used at all, and if in the course of its operation it needs an indepth diagnosis, then it seems that there is less need for diagnosis to be done in detail by the sort of standardized test and DEBUGGY analysis done by Brown and Burton. In this context, a standardized test could be analyzed to determine problem areas in enough detail that the online Tutor could be used efficiently. In a situation where resources are limited, (i.e., virtually everywhere), it would be inefficient to have the Tutor find out what sorts of problems a student had. If a diagnostic test (which can be computer scored) can be used to determine that the student only has trouble with problems involving zeros on the top, then the Tutor can immediately deal with that type of question and find out why.†

If the Tutor being described were available, then DEBUGGY would appear to be not the way to get detailed analysis. DEBUGGY is justified as being a good way to get detailed diagnosis if there is a restriction on the detail of the data available. IDEBUGGY, the interactive version of DEBUGGY, does not exploit the extra data possible in an interactive system but only refines the test questioning. It is interesting to note that IDEBUGGY was chiefly used with student teachers to give them practice (and the very idea of) debugging student algorithms. (The system simulated buggy algorithms and the student teachers, by choosing good test problems for the system to answer, had to find the bug.) This is analogous to the educational use of MYCIN in the GUIDON system [Clancey 82] wherein the art of diagnosis is taught to medical students.

† High level diagnosis based on such tests may itself be done be a computer diagnosis system such as the one described in [Colbourn (Jones) and McLeod 83], [Colbourn (Jones) and McLeod 84] and [Colbourn (Jones) 83].

## 4.3. UNGUIDED PROBLEM SOLVING

In this mode, the system does not issue prompts, but merely monitors the student's solution. Since the student might not make every step explicit, the problem of diagnosing errors becomes more difficult. For example, if the student does not explicitly state that he is going to borrow, then what we interpret as borrowing errors may well have been errors in the test to determine whether borrowing was necessary. To alleviate this possible compounding of bug hypotheses, the system might insist that the borrowing steps be made explicit. This is not too unreasonable, since a human tutor would probably insist on the same thing if the student were having difficulties. One thing to notice is that the problem of determining what the student is up to at any given time is alleviated by the fact that the screen position of an entry indicates whether the student is trying to borrow, show the new minuend, enter the answer, etc. So all the system needs to do is run through its algorithm until it finds an entry in the same region (eg., answer) and then compare. Also, if there is too much confusion, it has the option of switching to GUIDED PROBLEM SOLVING mode and stepping through the problem from the point where there is some uncertainty.

The two main concerns in this mode are (1) what to do in case of a detected error and (2) when to switch to another mode. These two decisions are based on information in the Student Model, the Tutorial Strategy, and the current state of the tutorial. Some examples will show this interaction.

Suppose the system believes that a student has mastered a skill but discovers an error in that skill. In this case it may be best to let the student continue in hopes that he will discover his own error, possibly by checking his work via a method such as adding. This would be particularly beneficial in the case of a student who is known to be careless. It is the sort of strategy we would want to be following if the student had already been through the more elementary modes and was in the process of becoming able to solve problems independently. On the other hand, if the error is due to a bug which the student is thought to have, or if there is no evidence one way or another, it may be better to interrupt, give him a chance to correct, or even go to GUIDED mode or DEMONSTRATION mode. That is, sometimes it is beneficial for the student to struggle, but other times it is a waste of time and only contributes to confusion. But suppose the student had just signed on with the general request to get help with a range of subtraction problems. The system may use UNGUIDED mode in order to get a rough estimate of the bugs the student has. If this were the case, it may note error types but not even bother to correct or tutor the student until it had enough detail to decide what its tutorial strategy should be, in particular, what skills and examples it would deal with first. (For example, it would not be a good idea to try and correct errors with zeros in borrowing if the student does not even understand borrowing without zeros.)

To summarize, the issues in the UNGUIDED PROBLEM SOLVING mode get very complicated, primarily because of the number of options the system has at any point and because of the large amount of knowledge the system needs in order to choose among them. The problem of implementing a system with these capabilities is much more formidable than implementing one which functions in either the DEMONSTRATION mode or the GUIDED PROBLEM SOLVING mode. What is clearly needed is an analysis of the tutorial strategies required for teaching subtraction skills as well as a student model formalism which is adequate for representing these skills and their deviations. But given that diagnosis is not computationally demanding in the Subtraction Tutor, the complexity of the higher functions, ie., strategy, is proportionately less of a problem.

## 4.4. Student Control vs System Control

Another general CAI issue concerns the degree of student control the system is prepared to allow. Laurillard [Laurillard 83] uses this distinction to classify CAI systems into four types on a spectrum from complete computer control to student control:

[1] Drill-and-Practice : the student's answers determine the level of difficulty of the next question.

[2] Tutorial :the computer attempts to correct errors the student may have.

[3] Simulation : the computer simulates some phenomenon not easily observable.

[4] Modelling : the student is allowed to change the rules which describe the operation of a system being studied.

Generally, the claim is that student controlled systems are more successful; however, part of the difference may be explained by the differences between the goals of the systems mentioned. For example, it may be the case that computers are simply better at doing simulations (which incidentally involve student control) than drill-and-practice. Possibly simulations are simply more interesting than tutorials. (Notice that computers used to simulate physical phenomena are not being used to simulate an activity of a teacher the way CAI tutorial programs are.) In any case, the importance and desirability of student control is acknowledged, but its importance within each of the four types of CAI system should be stressed.

Locus of control is also important because it captures differences in design philosophy. In the computer controlled design we see the computer as emulating a teaching style in which the student is somewhat of a second class citizen. Teaching strategies and such are often kept transparent to the student, and his preferences may not be actively solicited. (This is analogous to some medical styles, in which the expert does not open up his methods to inspection by the patient.) Such a CAI system is a tool used by the **teacher** to help him **teach** the student. On the other hand, a machine which is under total student control is a tool used by the **student** to help himself **learn.** The ideal is a compromise which has its parallel in a tutoring relation in which the tutor is there to help the student learn, but at the same time has expertise that the student lacks. In some cases, therefore, we must expect that the tutor does know better.

Student control (at least partial) of a tutorial session is helpful in that it often makes up for tutoring errors. For example, the system may think that a student has mastered some skill based on correct responses, but the student may not feel comfortable (maybe he was just lucky on some answers). In this case, the system should be amenable to requests for additional examples. It can even solicit requests with messages like: "Would you like a simpler example?", "Would you like me to finish the example for you?", "Would you like me to explain why this is done?", etc. In the opposite case, sometimes a student will try to progress too fast. In this case, it may be necessary to hold him back until he has achieved a certain level of performance. Messages can be used such as, "You seem to be having difficulty with this kind of problem - let's go back to a simpler example." On the other hand, some students find too much external control to be frustrating and even demeaning.†

Similarly, the system may allow the student to ask for hints in either the GUIDED or UNGUIDED modes. There will be a range of hints available from none at all up to the answer. It would be appropriate for a hint to be understandable within the student's current knowledge. For example, if the student is thought to know the test which determines whether he can borrow from a column, then a hint as to whether he can borrow might be "is the top zero?" (The value as a hint assumes that if he knows the rule, the answer to the hint will tell him the answer to his question. If he doesn't know the rule, the answer to the question tells him nothing and only encourages guessing, which is the last thing we want him to do.) If he does not know the rule, it would be better to provide the rule itself: "you can borrow only if the top is not zero." At some point in the tutoring process, it is expected that the student should not require hints. If a pattern of hint requests occurs past this point, then there may be some more serious problem.

## 5. Student Model Formalisms

Given that a student model is necessary for some functions of the Subtraction Tutor, we can raise the issue as to what form it should take. Clearly, some notion of a bug is necessary if the system is to form hypotheses as to the sources of a student's errors. In this section alternatives proposed in the recent CAI literature are examined. The main suggestion is that the choice of a formalism should be guided by what the system needs to know about the student in order to perform the functions for which it is designed.

† I recall using a CAI package which insisted on moving me back to square one every time I made an error. I would have liked the option to carry on if I thought I understood the problem.

## 5.1. Debuggy

The Buggy model in full-blown version is just too complicated for an interactive system. Recall that the full list of bugs includes ones which are extremely rare. Consequently, a set of only the most common bugs would be adequate.

## 5.2. Overlays and Genetic Graphs:

Goldstein [Goldstein 82] experimented with student models in the course of developing a computerized coach which helped students learn to play a simple maze exploration game called "Wumpus" [Yob 75]. Wumpus required logical and probabilistic reasoning. Once the rules of the game have been learned (i.e., the legal moves), students can increase their skill in the game and eventually learn how to win (slay the Wumpus). Various WUSOR systems were developed to coach the student, enabling him to learn the strategy of the game faster than he would have on his own. WUSOR-I [Stanfield, Carr and Goldstein 76] was based on a set of rules used by an expert playing the game. The student model here consists of an "overlay", or subset of the expert's repertoire of rules.

In WUSOR-II [Carr 77], the rules were divided into 5 levels of difficulty which reflected the general order in which students should learn them. This avoided attempting to teach students the subtle aspects of the game before they had mastered the basics. WUSOR-III was intended to be the final version, and was to incorporate a "genetic graph", the nodes of which correspond to rules. The arcs, or edges, of the graph represent evolutionary relationships between the rules, that is, the learning relation they have to one another. For example, rules can be analogous to one another. They can also be refinements, corrections (specializations), or generalizations of one another. The motivating idea is that students do not (should not) learn rules in a random, haphazard way but as natural extensions of the rules they have already mastered. The genetic graph therefore, can represent these relations. The student model, which is an overlay (that is, a subgraph) of this graph can contain not only the rules the student has mastered but also a representation of the order in which he learned them. This allows the coach to determine a type of learning preference of the student and thereby coach more effectively. For example, if the student exhibits a pattern of learning by analogy then the coach should pick the next rule to be taught from those nodes which are linked by an analogy relationship to a rule already mastered. This structure allows for a teaching strategy which is not based on scripts (in which material is presented in an invariant order).

The complete student model is an overlay of the genetic graph with buggy variants attached to the appropriate nodes. The order of aquisition of the rules is encoded into the student's graph. The genetic graph developed for WUSOR-III contained approximately 100 nodes (rules) and 300 links.

## 5.3. Genetic Graph of Subtraction Skills

Wasson [Wasson 85] combines the genetic graph formalism of Goldstein with the subskill analysis of Brown and Burton, and produces a genetic graph of subtraction skills. The student model consists of an overlay of the genetic graph with buggy variants to be added to the nodes of this graph. Thus, it is claimed, the advantages of the genetic graph formalism are made available. I believe that the genetic graph formalism is unnecessary in the case of subtraction and that the complexity of it is not warranted.† The reason it is unnecessary is that subtraction is a much simpler domain than the game of Wumpus (even though Wumpus is much simpler than most other adventure games).

In Wumpus, a set of **game rules** must be learned **before** the game can even be played. The game itself is non-deterministic (as all real games are), that is, the player is free to choose any legal move at any time. The "rules" in the genetic graph represent **strategies** a player may or may not employ. But whether a player is a novice or an expert, he plays the **same game** according to the same regulations. In the case of subtraction, however, a deterministic algorithm is applied to a given problem. There really isn't any strategy involved. (This is not true of all procedural skills, however, as will be seen in a later section.) In a reasonable educational system, the rules of subtraction (steps in the algorithm) are presented and learned in a generally fixed order as in WUSOR-II. In

† Note that the claim in [Wasson 85] is not that the a genetic graph is the most appropriate way to model student's knowledge of subtraction, but that it can be done in this way. The idea is to test the genetic graph formalism in a simple domain and then extend it to more complicated domains.

particular, students are not exposed to types of problems which require rules they have not been exposed to. For example, if borrowing has not been taught, a student does not get questions like:

```
  33
-  7
 ---
   ?
```

Similarly, if borrowing from zero has not been taught, there will be no problems presented which require this skill. If indeed subtraction were learned like Wumpus, a beginning student would be exposed to all types of subtraction right from the beginning, and would experience large amounts of failure and frustration as he was gradually filled in on the rules needed to cope with more complicated examples. Thus, although it is possible to teach some subtraction skills in various orders, there is no necessity to do so, and probably no advantage anyway. The conventional approach, which is essentially script-based seems adequate. (Conversely, one could probably teach the game of Wumpus in a way analogous to subtraction, that is, by devising cave designs which are simple enough to allow the student to win at first by using only the most rudimentary strategies, and then gradually increasing the complexity while introducing new rules. Notice that this is possible with maze-type games because the terrain is not constant, as it is in chess, for example.)

Consequently, the simplest way to model student knowledge for subtraction is probably an overlay of an ordered set of subskills along with buggy variants. In fact, a genetic graph representation using just component and pre/post links would constitute such an odered set of subskills.

Given that the genetic graph formalism is powerful enough to represent knowledge in domains more complicated than subtraction, there is some reason to advocate its general use on the grounds of uniformity of representation. If, however, a practical system is being built, and especiallly if the system must give real-time response, it may not be possible to live with the overhead that a complex structure requires. Moreover, in terms of obtaining a clear theoretical understanding of the tutoring and diagnostic processes over various domains, it would seem that trying to achieve the minimal representation of student knowledge which is required in any case would display the similarities and differences in domains more clearly than systematically using the maximal one.

This is not to say that research into the application of generalized student model formalisms is not useful. The researcher concerned with building practical systems needs a variety of formalisms to choose from, particularly when attempting to design greater capabilities into a system. Also, there is an advantage to finding a single student model formalism because it would simplify the development of CAI systems in different domains. In some sense, it is what we would like to find feasible.

## 5.4. Automatic Generation of Bugs

A further issue raised by Wasson is the automatic generation of the buggy variants of skills. [Wasson 85] The idea is to have a procedure which, given a correct algorithm†, will automatically generate buggy variants according to generic errors:

[1] Omit or add steps to the algorithm.

[2] Permute the order of the steps.

[3] Permute the order of the arguments in a step.

[4] Test conditions incorrectly.

[5] Substitute one argument for another.

These general bugs do in fact account for many of the bugs discovered by Burton and Brown. (There is some doubt as to whether all actual bugs can be generated this way. In [Burton 82, p. 177] the claim is made that some observed bugs "have no vestiges in the correct skill". In this case, a bug catalogue would

† [Jones, Poole, Wasson 85] contains a recursive algorithm for subtraction written in Prolog which is offerred as a candidate for buggy modification. However, it does not simulate the conventional algorithm in that it assumes that it will be able to borrow and then adds the carry of 10 without actually testing or borrowing from the next column. In the recursive call to the next column, the top digit is decremented if a previous borrow did occur. If further borrowing is needed, the process repeats recursively. In the special case of zero, in which an extra borrow is always required, the value is reduced to -1 when decrementation occurs, and then increased to 9 (correctly) when the carry is added. Thus, there is no special treatment of zero in the algorithm. This is in fact more efficient than the conventional algorithm, but an algorithm for a buggy-style diagnostic application should reflect the actual practice, including (1) a test for the special case of zero and (2) explicit borrow before the subtraction is done, otherwise buggy variants will not be the ones possessed by students.

be needed in order to identify all errors. On the other hand, such bugs do not occur with sufficient frequency to warrant their prediction.) In any case, it is not clear what the advantages would be to doing this. After all, for CAI type domains, there are professionals with a wealth of experience which needs only to be tapped, so why not extract this expertise and code it directly into a given system.† (Although work in expert systems has shown that extraction of expertise is not easy and often is inaccurate, in the arithmetic domains such as subtraction, expert knowledge is relatively accessible. The ease of direct coding of buggy variants will depend on the domain and experts involved.) Moreover, the above strategy is potentially a computational nightmare. In addition, it complicates the procedure whereby the set of bug hypotheses is tested, for there will be many fictitious bugs generated which must be distinguished from real, though possibly not guilty, bugs. If experience is appealed to in order to limit the set of bugs automatically produced to real ones, then experience might as well be used to generate them in the first place.

The motivation for automatic bug generation can be seen as in part a validation of the psychological view that errors are due to generic variations in algorithms, but more importantly as a desire to dispense with what is normally thought of as domain specific information. If automatic bug generation were possible, we would be able to design diagnostic components which only require an expert for the correct algorithm. Thus a large part of the process of encoding domain knowlede would be automated.

## 5.5. Repair Theory

The work of Brown and Van Lehn [Brown and Van Lehn 80] provides a useful approach to this problem. They provide a "generative" theory of bug formation based on the idea that a student with a bug is typically following an incomplete algorithm. He uses it until he encounters a case in which the incompleteness prevents further progress. The student then tries to "repair" the algorithm with a buggy variant which enables him to solve the problem. The test for the

† In case the catalogue of bugs were very large, a genetic variant approach would save space but at the expense of an increase in time complexity, which is not appropriate in a tutorial context, though it may be in say a Debuggy type of context. In addition, the genetic variant approach may save time in initial system development.

adequacy of a repair theory is that it be able to generate those and only those bugs which students are observed to have. Thus there is a control on the hypotheses which are generated. This set will correspond to the actual bugs observed if the theory reflects the psychological processes of the typical student. (Even if the theory does not correspond to student processes, it might pick out the same bugs by luck.) Automatic generation, by contrast, has no hope of producing a controlled output, since it is not guided by any theory which might implicitly do the controlling. The size of the hypothesis set at a given time could be controlled by dynamically generating bugs relevant to the current step in the algorithm. Thus in the context of stepping through a solution (as in the Subtraction Tutor but not Debuggy), the explosive growth of bugs which must be considered could be controlled. However, within the set of current possible bugs, there would be no guarantee that any bug is a "real" one, that is, one which students are actually found to possess.

Repair Theory has an intuitive appeal in that it captures the behaviour of those students whose strategy is to blunder through a problem at all costs, but in at least a reasonable looking way. It is not a theory to incorporate in a practical system, since all a practical system needs to know are the actual bugs. But a tutoring system may well address the issue of trying to identify students who have a tendency to freestyle their way through a procedure. It is certainly a worthwhile educational goal to produce students who are aware of their own mental struggles to the extent that they can distinguish knowing an algorithm from inadvertently faking it.

## 5.6. Summary of Student Model Formalisms

A simple way to model student knowledge would be to use the production rule analysis in [Young and O'Shea 82]. The advantage is that there are a small number of these rules which can be added, deleted, and refined independently of one another. For purposes of teaching or tutoring, the production rules can be ordered into levels, as the strategies for Wumpus are in Wusor-II. A student model then consists of a subset of the correct production rules and the incorrect ones, that is, an overlay. The goal of a tutorial session, then, is to produce a model (corresponding to a student) which consists of exactly the set of correct rules.

## 6. Other Domains

In this section other domains involving simple procedural skills are considered in order to assess which areas can make use of the principles of the Subtraction Tutor.

### 6.1. Integer Arithmetic

#### 6.1.1. Addition, Multiplication, Division

An Addition Tutor would be much simpler that an Subtraction Tutor because carrying is a much simpler operation than borrowing. (There is no special case analogous to borrowing from zero.) On the other hand, if more than two addends are allowed, the difficulty of "footprinting" the partial sums would have to be dealt with. Possibly, a separate routine could debug these errors by having the student enter the partial sums as he calculates them. (Requiring the student to make explicit steps which are not normally made explicit is an extremely important issue in the design of tutors such as the Subtraction Tutor. I will return to this shortly.)

Multicolumn multiplication involves both multiplication and addition, and so it would be more complicated due to the increased number of errors which are possible. However, as long as operations can be represented in a grain size equal to the grain size of students' normal solutions, all typical errors should be easily identified. In particular, we want to have access to the partial products and their screen locations. We also would want to save the carry information in situations where errors are not corrected as they are made. The screen representation of multiplication is required for easy diagnosis of location errors of partial products, which is a main source of errors especially when there are zeros in the multiplier.

Division is the most complicated integer operation. In the first place, the physical layout of the solution is the most demanding in terms of correct column placement of the successive dividends. Secondly, it involves a degree of indeterminism in the selection of the divisors. In simple problems, a student needs to know his "goes-into's" for example "6 goes into 52, 8 times with something left over". In questions with larger divisors, a strategy is needed to estimate the quotient, for example, rounding the divisor and current dividend and using a primitive "goes-into" rule. At the base level of operation, a Division Tutor ought to be able to monitor a student's trial and error approach to determining the correct quotient digit. It should allow the student to make unwise choices which will be found out to be incorrect, but it should be able to correct mistakes such as trying a quotient digit when a smaller one has already proven to be too large. At a higher level the Tutor should be able to simulate a round-off type of strategy. Consequently, the system will need to know which strategy (trial-and-error or estimation) is appropriate for a given student, and when it is time to introduce the better strategy†. Hopefully, the estimation strategy is motivated by the student's perception that it offers a faster way to accomplish something he already understands. It may be that the system could focus on the development of the quotient discovery skill by exempting the student from the multiplication and subtraction steps that are needed to determine whether the trial quotient digit is too big or too small.

### 6.1.2. Making Steps Explicit

It has been suggested several times that a Tutor may find it convenient or necessary to require the student to make steps explicitly on the screen instead of "in his head". The advantage is clearly that it potentially eliminates some of the confusion concerning sources of error. The difficulty is that the process of doing this may be confusing to the student since he in effect is being taught a new "written" algorithm which is to replace the conventional one, but only for the duration of the tutorial. Consequently, we have to balance the student overhead for facilitating communications with the usefulness of the extra information. The process of making steps explicit is itself a new source of error. For example, what the system sees as an error in a step which has been newly made explicit may be due to the student's unfamiliarity with the I/O protocols.

† Also, shortcut methods for dividing (and multiplying) by 10, 100, 1000 etc. should be introduced.

Therefore, the idea must be conjectured with some care.

The examples mentioned so far have been the decision points in subtraction and the running totals in addition of more than two numbers. By and large, the pencil and paper algorithms which are learned in Canadian schools are well suited for tutoring in the method which has been suggested, because they are relatively complete. By contrast, some pencil-and-paper algorithms taught in some European schools would be very difficult to debug with a Tutor. For example, division is often taught with the suppression of the partial products. The student estimates the quotient and writes it down. Then the quotient digit is multiplied by the divisor digits, the result **mentally** subtracted from the current dividend, and only the difference actually written down. Students asked about this method claim that the justification is that it sharpens their skills because they are forced to remember more. Whatever the reason, it means that a European Division Tutor would have to use far more buggy rules to explain mistakes in the partial products, and have a corresponding greater degree of uncertainty in its diagnoses. The temptation would be to "reteach" the explicit algorithm to a student having difficulty with the abbreviated one in order to facilitate diagnosis. However, learning an expanded paper representation in this case may really confuse the student and, as mentioned previously, introduce a new source of error. As an option, the partial product could be entered in a separate screen location as part of a full multiplication procedure with the current quotient digit and the divisor. Also the subtraction step could be represented separately. These sorts of options would probably have to be tested in order to discover which in fact works better with students who have difficulty with division.

The intricacies of this issue are being stressed because it is believed to be very important in the design of tutors such as the Subtraction Tutor. Every time a distinct part of a skill can be represented on the screen instead of only in the student's head, the system has a chance for direct confirmation (or rejection) of its hypotheses concerning the student's knowledge. And every time this is done the computational aspects of the diagnostic function become simpler and more reliable. As a useful byproduct, the system itself becomes easier to understand because its operations become closer to our intuitive idea of what it means to tutor a student, and farther away from the statistically sophisticated control structure of DEBUGGY.

### 6.1.3. Vertical Integration of Tutors

Multiplication and division illustrate the need for vertical integration of tutors. That is, a Division Tutor, for example, in addition to knowing the skills and bugs associated with the division operation itself, must be able to call upon a Multiplication Tutor and a Subtraction Tutor to find errors in these component skills. It may discover that the student should in fact be sent back to these tutors for remediation before division is learned. This knowledge of the subskill analysis of a skill into components which are themselves **complete operations** which can be learned in isolation characterizes good human tutors. Often, the most difficult students to tutor are ones who have a bug which is in a component skill far removed from the skill being taught. For example, a student may be learning how to factor trinomials which involve fractions. He may be confused as to how the factoring rules are to be applied in the case of fractional coefficients, but he may simply not be able to work accurately with fractions themselves. This is not at all uncommon, since reliance on electronic calculators does not allow the development of skills with common fraction operations. If a fundamental difficulty with fractions is encountered, the best strategy is to stop work on trinomials of this type until the basic skills with fractions are mastered. In this way, the skills can be learned independently rather than together, which may prove to be more confusing for the student. The point is that a computer system which is going to tutor the factoring of trinomials will be much more powerful if it can call other tutors which are specialists in the component skills. This capability in turn requires that the control structure knows how to recognize when this is necessary or even worth checking out.

### 6.1.4. Base n Operations

The four basic operations could be extended to base 2 numbers and addition and subtraction could be extended to base 8 and base 16 numbers. In this context, the EXPLANATION mode would be of most use inasmuch as it graphically represents the meaning of the place notation. This is particularly important in making students see that base 10 is only one of an infinite number of possible place notation schemes. One difference that we would encounter in tutors for operations with other bases is the necessity to include a scratch area for working out some operations by converting to base 10. For good reasons, most students do not learn a whole new set of math facts when they learn how to, for example, add in octal. Rather than learning that in base 8, "$4 + 5 = 11$", a student can calculate in base 10, "$4 + 5 = 9$", and then convert the answer to octal: "$9 /8 =$ 1 with 1 remainder, therefore the answer is 11". For hex calculations, the digits greater than 9 are converted to decimal, the calculation is performed, and the result is converted back to hex. Consequently, a scratch space should be used to capture conversion errors as well as errors in decimal operations themselves.

## 6.2. Real Arithmetic

The four operations with real numbers are very similar to the operations with integers, with the following exceptions. In addition and subtraction, the transcription problem is more complicated because decimals must be lined up. Also, padding with zeros must be dealt with. In multiplication, a rule for decimal location must be learned. In division, the decimals must be manipulated at the outset, and padding the dividend with zeros may be necessary. In some cases, the quotient must be rounded off.

## 6.3. Common Fractions

This is the most complicated topic to tutor in a primary or elementary level arithmetic course. In the first place there is the problem of representing the fractions in a clear way on the screen, with horizontal, not diagonal, fraction lines, and with a clear relationship between whole number and fractional components of a mixed number. But this is primarily a technical problem, although an extremely important one. The educationally interesting problem is dealing with the extreme amount of indeterminism in basic operations with fractions.

To illustrate this indeterminism, consider the operation of reduction of fractions to simplest terms. This procedure is used in all four basic operations, since answers must be expressed in simplest form. Therefore, all fractional operations will be at least as difficult as reduction. There are basically two methods for reducing fractions:

[1] Successive Division : Find a number which divides the top and the bottom evenly. Divide it into the top and bottom. Repeat the process on the results.

[2] Factoring : Factor the top and bottom into prime factors. Cancel out the factors which are common to top and bottom. Multiply the remaining factors.

First consider the method of Successive Division. For one, the student is free to choose which division to do first, top or bottom. But he is also free to choose **any** even division. Some people divide with the smallest number, some with the smallest prime, some with the largest divisor they can think of, and some with any number. Some people look for lucky cases, eg., the top or the bottom divides the other evenly. Some people always cancel zeros first (a wise move). Within these options, some people know the tests for divisibility for 2,3,5,9,and 11, and so they can tell by inspection whether one of these numbers evenly divides another. Other people actually have to perform the division (possibly with an electronic calculator) in order to see whether it is even. The point is that though some of these methods may be better than others, they are all acceptable, and a Reduction Tutor needs to be able to follow a person's step by step operations no matter which method or methods he uses†. In light of this, it may be useful to have the student somehow specify on the screen what his current strategy is. The difficulty is that even students who can reduce fractions may not be able to articulate (even by menu choice) what it is that they are doing. But at least we can imagine that a student could indicate the number he is choosing for a common divisor. Thus, a solution may look like:

† See Bregar and Farley 83]: "to be useful in a problem solving context, a problem solving system must provide an appropriate level of explicitness, or granularity of its solutions and must be capable of monitoring a range of competencies." (p. 11) This claim is made in the context of an algebra tutor, but it applies to tutors such as the ones which have been described.

$$\frac{12}{18} \overset{2}{=} \frac{6}{9} \overset{3}{=} \frac{2}{3}$$

where the divisors 2 and 3 are shown explicitly. Thus the tutor could detect errors in tests for divisibility as well as errors in actual divisions. The system should be able to manage a lot of the screen details like displaying fraction lines and equality signs. Since it would then know the correct locations for entries, it could detect errors in placement. On the other hand, every time the system provides a guide like this, it is not able to test the student's ability to perform that sort of operation on his own.. This is not an unimportant point. In many cases students can solve problems once the basic structure of the solution is displayed. The formal method of square root extraction is a prime example of a problem of this type.

The motivation for the Factoring method is that the above method is not capable of producing answers in cases where there is a common factor which is bigger that the numbers in the students repertoire of "goes-into's". For example:

$$\frac{26}{39} \overset{13}{=} \frac{2}{3}$$

Since most students do not know what 13 goes into, they are unable to solve the above problem, without using a brute force approach with an electronic calculator (that is, try everything up to 26 on the calculator!) If factoring is used, the same result can be obtained by knowing only what 2 and 3 go into:

$$\frac{26}{39} = \frac{2 \times 13}{3 \times 13} = \frac{2}{3}$$

This method is also useful in that it can convince you that you are in fact done. That is, if large numbers are left, there may be some uncertainty as to whether there is, in fact, a common divisor. Factoring into prime factors proves that there is no such hidden factor.

The Factoring Method can also be used by factoring into composite numbers, which themselves may be reduced by the Successive Division Method, for example:

$$\frac{48}{64} = \frac{6 \times 8}{8 \times 8} = \frac{6}{8} = \frac{3}{4}$$

It is apparent that even a simple operation involving a reduction of fractions can be done in a very large number of ways if all combinations of strategies are allowed. If we were restricted to the buggy strategy of considering only whole answers as evidence, the number of possible hypotheses for an incorrect answer would be staggering. Another reason for preferring that solutions be examined in detail is that we would expect a greater frequency of careless division errors in a situation where much of the student's attention is focussed on the rules and strategies of reduction itself. Careless errors are the bane of Buggy: its motivation is the conviction that most student errors are systematic. In a Reduction Tutor, the actual division step will be isolatable and division errors will be easily detectable. Moreover, as with the Subtraction Tutor, it would be possible to collect statistics on the distribution of factual errors over the domain of primitive facts.

The Reduction Tutor must be able to simulate the two methods of reduction as well as monitor and correct the student's solutions. In some cases, the student may pick a less than optimal method. In these cases, the tutor needs to know when to suggest a better solution and when not to. For example, if the student uses Factoring or Succesive Division on:

$$\frac{180}{240} = ?$$

instead of immediately cancelling zeros (dividing by 10), he will require 4 extra operations. The Tutor should point this out, but can choose between introducing the idea of cancelling zeros as a new strategy, if the student is not known to be aware of this strategy, or simply reminding him, if the strategy is thought to be part of his repertoire. The importance of this distinction is that we do not want to confuse the student by assuming he knows something. (The student wonders, "Am I supposed to know that? Was I told but just forgot? Was I supposed to be able to figure it out myself?") Many of the tricks used even in arithmetic are not things that the average student could be expected to know intuitively or be able to derive automatically. These items can be best integrated by the student if he has a clear realization such as, "Well, I never would have thought of this in a million years, but when it is explained to me, I can understand it." On the other hand, some strategies can be learned by discovery, especially if hints are provided. For example, "You correctly solved the problem by completely factoring it, but what would have happened if you had divided by 10 right at the beginning? Try it." Whether the student produces the simpler solution or whether it is tutor-generated, ideally both solutions should be present on the screen in order to facilitate comparison.

The Tutor must also choose when and how to introduce a new strategy†. For example, although Factoring is a more generally applicable strategy in that it solves more problems given a limited set of division facts), it is normally taught after the Successive Division method. The Tutor could wait for the first problem of the type 26/39=?, let the student struggle, and then use this as motivation for the Factoring method. But some students don't appreciate this kind of manipulation, especially if they do not like to struggle. In this case, the Tutor may introduce an example first and then let the student try one. This is an example of a student learning preference type that would be useful to represent in a tutor.

The Reduction Tutor raises an issue which becomes more problematic as the grade level of the skill increases. In a given curriculum, there is no guarantee that factoring will have been taught. In this case, the Factoring Method should not be taught at all. Moreover, questions which can only be solved by factoring, given the set of division facts included in the curriculum, should not be allowed. Notice, that this suggests the need for a Curriculum Model if the Tutor is to be

† In light of the previous discussion of computer versus student control, there is no reason why the tutor can't simply **ask** the student, for example, "would you like to learn a simpler way to do this?" It may seem that the obvious response would be "yes", but many students find alternate methods confusing if they have not mastered the current one. Even if students would generally answer "yes", it is a matter of politeness, if nothing else, to ask.

capable of functioning in a variety of educational contexts as opposed to being custom built for one curriculum.

## 6.4. More Complicated Mathematical Domains

A natural question to raise is whether the principles of the Subtraction Tutor can be extended to more complicated mathematical domains such as theorem proving. The two main ideas behind the Subtraction Tutor are that student solutions should be monitored in as much detail as possible and that this detail should be represented on the screen in as natural a way as possible. In theorem proving, for example, the steps of the proof must be shown because they constitute the solution. A novel use of the screen in [Anderson, Boyle and Yost 85] is to represent a proof as a tree whose nodes are the definitions and rules which may be used. In this case, the automated tutor uses a representation technique which is standard in automated theorem proving, and which ought to be part of the standard way of teaching the subject to students. The only area that might benefit from more detail is the process whereby the student determines what his next step should be. But the difficulty is that the teaching of theorem proving has not produced a more algorithmic or reasoned out approach to this which can be simulated by the system. (It seems like a black art to the student.) Perhaps research into this area will be a source of new teaching methodology rather than an encapsulation of the existing one as is the case with the subtraction tutor.

## 6.5. Non-Mathematical Domains

We first note that the bulk of the current research into tutoring systems has focussed on mathematical domains. The obvious reason is that answers in this domain are readily calculable compared to, for example, common sense reasoning. A less apparent reason is that most researchers are scientists of some kind and are therefore more familiar with mathematical domains. At any rate, it is of interest whether there are any non-mathematical domains which are procedural in some sense and which would benefit from screen representation of solutions.† One area which exhibits a procedure of sorts is the parsing of

† There are many domains which do not involve procedural skills but which benefit from screen representation. The teaching of graphic design and composition principles could be automated by having students input elements into a frame and having the resulting composition evaluated by a composition tutor. There are undoubtedly many such ex-

sentences in English grammar classes. If it weren't for the natural language aspect of this task, this would in fact be a mathematical skill and a mechanical one at that. The normal procedure is unlike a typical parsing algorithm in that students are taught to divide the sentence into subject and predicate first and then refine this initial division, introducing direct objects, indirect objects, and ultimately classifying each word according to its part of speech as used in the sentence. While there is a semblance of method in this procedure which can be represented as a sequence of questions such as "What is the action in the sentence?", "Whom or what is doing the action?" and so on, there is much difficulty in explaining why an answer is right or wrong. In the first place, the interaction must take place in natural language, and the system must be able to understand the implied semantics of the sentence. In fact, the whole idea of a student's parsing a sentence is somewhat confusing because we would expect that a student must have done some amount of unconscious parsing if he has understood the sentence at all. At any rate, this would be an interesting area to investigate, but only if the expert component can be developed so that it can not only produce correct parses (which is not especially difficult) but also justify its parses in a human like way. (The use of a definite clause grammar for the expert is suggested, but such a grammar uses a part-of-speech pattern match as its method. It would need to be modified to use semantic information.) This does not mean that it must be able to parse in a top down, semantic way, but only that it can reproduce this sort of parse once it has its own parse (probably based on tables associating words with their part of speech). A further complication is that human parsing exhibits a mix of top down and bottom up parsing, for example, if you know that "of" is a preposition, and you know that prepositional phrases cannot be direct objects, you can rule out the possibility that a phrase beginning with "of" is a direct object without understanding the semantic content of the phrase, that is, in this case whether the phrase denotes something which is receives the action denoted by the bare predicate (verb) of the sentence. Since the whole area of natural language understanding is difficult in itself, it may be premature to contemplate systems which can tutor its analysis.

amples.

But if the complexity of the sentences is restricted, and if the vocabulary and context is restricted so that the system does not have to deal with any ambiguity, then such a tutor may be feasible. It may even be useful for natural language research itself.

## 7. Conclusions

It has been argued that detailed subtraction error diagnosis is more effectively done in a tutorial context than by examining whole answers to diagnostic tests as is done by Debuggy. This has been shown to be even more critical in the case of more complicated, non-deterministic, procedural skills such as integer division and common fraction reduction. It seems unlikely that the Buggy model could be easily or effectively extended to these domains. The tutorial approach, on the other hand, seems readily extensible because it monitors the details of the student's solution and is not especially bothered by the fact that there is more than one way to solve a problem.

As a side benefit, it appears that subtraction is a suitable domain for developing an automated tutor in that it is complicated enough to allow for switching methods by the tutor but is not overly complicated. The only missing element, namely strategy, can be introduced by choosing a domain such as fraction reduction, which is still relatively uncomplicated.

## 8. Directions for Future Research

The production rules of Young and O'Shea appear to be the most promising way to implement the various modes of operation of the Subtraction Tutor. As the higher modes are implemented, a tutorial strategy appropriate to subtraction needs to be designed, along with a student model adequate for the tutor. In the other arithmetic domains, production rule systems could be developed, in order that other tutors be implemented. The tutorial strategy would have to be augmented to deal with the coaching of strategy in solving problems.

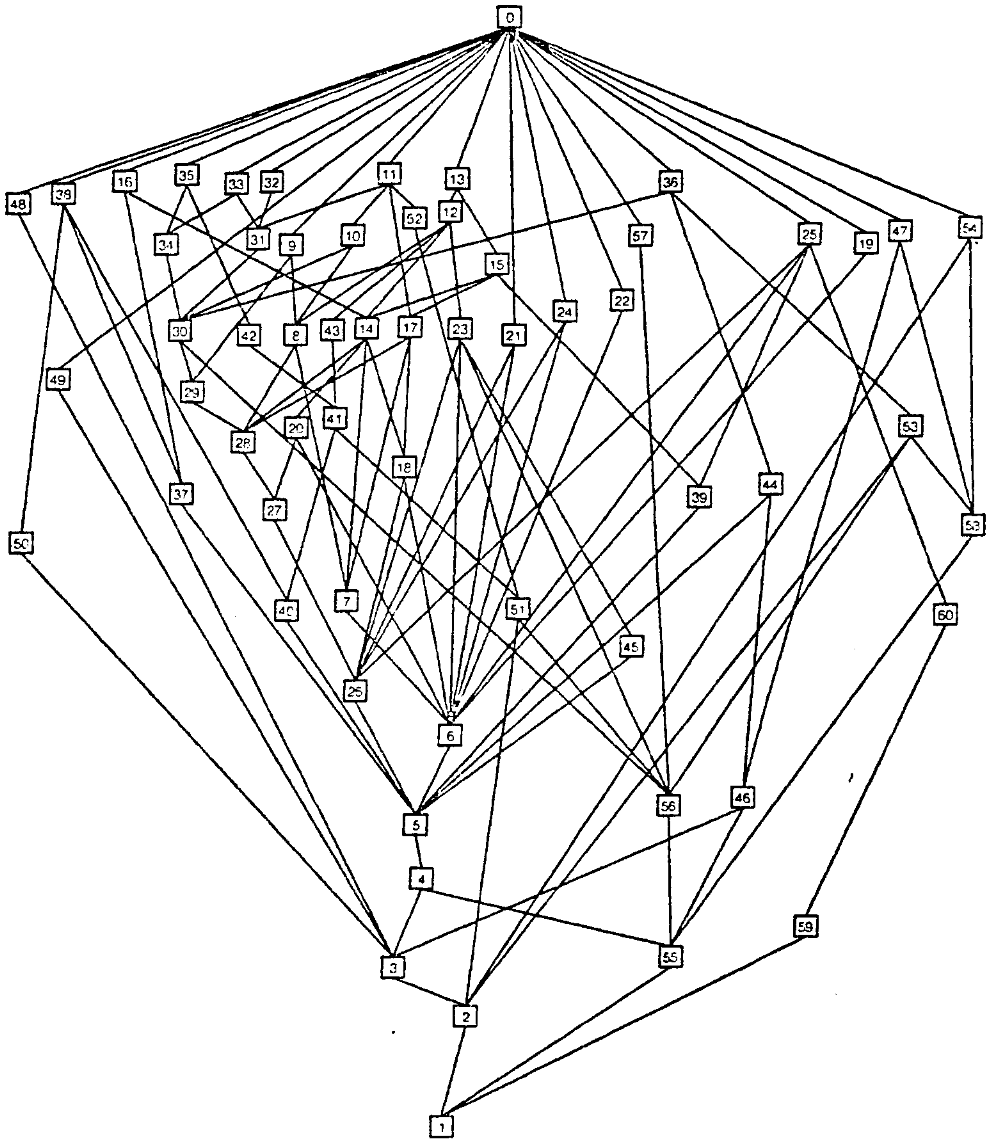

FIG. 4. Skill lattice for Subtraction

# APPENDIX 2

|  |  |
|---|---|
| 0 | Correct skills |
| 1 | No subtraction skills |
| 2 | Subtract 0 from a number whose right-most digit is 0 |
| 3 | Subtract 0 from a number |
| 4 | Subtract columns which are n-n or n-0 |
| 5 | Subtract without borrowing |
| 6 | Borrow from some columns with a digit on the bottom |
| 7 | Borrow from columns whose answer is the same as their bottom digit |
| 8 | Borrow from a column with a 9 or a larger number on top |
| 9 | Borrow from a column with a larger top digit |
| 10 | Borrow from a column with zero on top when blank on bottom |
| 11 | Borrow from a column with zero on both top and bottom |
| 12 | Borrow from a column unless one is on top and a nonzero digit is on the bottom |
| 13 | Borrow from a column unless digit one is on both top and bottom |
| 14 | Borrow in two consecutive columns |
| 15 | Borrow from columns with the same digits on top and bottom |
| 16 | Borrow from columns that have nine as the bottom digit |
| 17 | Borrow from columns with the same nonzero digits on top and bottom |
| 18 | Borrow from columns which have an answer of zero |
| 19 | Borrow from the leftmost column when it has a non-blank in the bottom |
| 20 | Borrow more than once per problem |
| 21 | Can borrow then not borrow |
| 22 | Borrow from columns with two on the bottom |
| 23 | Borrow from leftmost columns or columns that have a non-one on top |
| 24 | Borrow from columns with top digits smaller than the bottom digits |
| 25 | Borrow from columns that have a top digit one less than the bottom digit |
| 26 | Borrow from the leftmost column |
| 27 | Borrow once in a problem |
| 28 | Borrow from columns that have the top digit larger than the bottom digit |
| 29 | Borrow from a column with the top digit greater than or equal to the bottom digit |
| 30 | Borrow from a column with a zero on top |
| 31 | Borrow from a middle (not leftmost) column with a zero to the left |
| 32 | Borrow from leftmost column of a problem whose form one followed by one or more zeroes |
| 33 | Borrow from a column with a zero on top and a zero to the left |
| 34 | Borrow from or into a column with a zero on top and a zero to the left |
| 35 | Borrow into a column with a zero on top and a zero to the left |
| 36 | Borrow from a column with a zero on top and a blank on bottom |
| 37 | Borrow into a column with a nonzero digit on top |
| 38 | Borrow into a column with a one on top |
| 39 | Borrow when difference is 5 |
| 40 | Subtract columns with a one or a zero in top that require borrowing |
| 41 | Subtract columns with a zero in the top number that require borrowing |
| 42 | Subtract a column with a zero on top that was not the result of decrementing a one |
| 43 | Borrow into a column with a zero on top when next top digit is zero |
| 44 | Borrow from a column with a blank on the bottom |
| 45 | Borrow from a column with a one on top |
| 46 | Subtract numbers of the same lengths |
| 47 | Subtract a single digit from a large number |
| 48 | Subtract columns unless the same digit is on top and bottom |
| 49 | Borrow all the time |
| 50 | Subtract when one is in top |
| 51 | Subtract when neither number has a zero unless the 0 is over a 0 |
| 52 | Subtract columns when the bottom is a zero and the top is not zero |
| 53 | Subtract when a column has a zero over a blank |
| 54 | Subtract numbers which have a one over a blank that is not borrowed from |
| 55 | Can subtract a number from itself |
| 56 | Subtract numbers with zeros in them |
| 57 | Subtract problems that do not have a zero in the answer |
| 58 | Subtract numbers when the answer is no longer than the bottom number |
| 59 | Subtract leftmost columns that have top and bottom digits the same |
| 60 | Subtract columns that have top and bottom digits the same*the same ones. |

# APPENDIX 3

| | | | |
|---|---|---|---|
| FD. | M =m, S =s | ⇨ | FindDiff, NextColumn |
| B2A: | S > M | ⇨ | Borrow |
| BS1: | Borrow | ⇨ | *AddTenToM |
| BS2: | Borrow | ⇨ | *Decrement |
| CM: | M =m, S =s | ⇨ | *Compare |
| IN: | ProcessColumn | ⇨ | *ReadMandS |
| TS: | FindDiff | ⇨ | *TakeAbsDiff |
| NXT: | NextColumn | ⇨ | *ShiftLeft, ProcessColumn |
| WA: | Result =x | ⇨ | *Write =x |
| DONE: | NoMore | ⇨ | *HALT |
| B2C: | S = M | ⇨ | Result 0, NextColumn |
| AC: | Result 1 =x | ⇨ | *Carry, Result =x |

Figure 2. Production system for subtraction by decomposition.

| | | | |
|---|---|---|---|
| B2B: | S < M | ⇨ | Borrow |
| B1: | M =m, S =s | ⇨ | Borrow |
| NZN: | M =m, S 0 | ⇨ | Result =m, NextColumn |
| ZNN: | M 0, S =s | ⇨ | Result =s, NextColumn |
| NZZ: | M =m, S 0 | ⇨ | Result 0, NextColumn |
| ZNZ: | M 0, S =s | ⇨ | Result 0, NextColumn |
| SM0: | S > M | ⇨ | Result 0, NextColumn |
| NNN. | M =n, S =n | ⇨ | Result =n, NextColumn |

**Figure 4. Modified and additional rules needed to model errors in Bennett's (1976) data. (N.B. These rules do not form a "Production System"—see text for clarification.)**

| | | | |
|---|---|---|---|
| LHE: | M =m, S blank | ⇨ | Result =m, NextColumn |
| TEN: | M ten, S =s | ⇨ | *TreatMasTen |
| DEC: | Decrement | ⇨ | *LookAtLeftM, *MoveEyeLeft |
| DNZ: | Decrement, NonZero | ⇨ | *ReduceByOne |
| DZ1: | Decrement, Zero | ⇨ | Propagate |
| DZ2: | Decrement, Zero | ⇨ | *ChangeZeroToNine |
| D10: | Decrement, Zero Ten | ⇨ | *ReduceTenToNine |
| PROP: | Propagate | ⇨ | *LookAtLeftM |
| PZ: | Propagate, Zero | ⇨ | Decrement |
| PNZ: | Propagate, NonZero | ⇨ | Decrement |

Figure 5. Additional rules needed for Brown and Burton's (1978) problems.

TABLE 4
Production System Analysis of
Brown and Burton's (1978) Most Frequent Bugs

| *N* [a] | *Bug type* | *Drop rules* | *Add rules* |
|---|---|---|---|
| 57 | Borrow–from–0 | DZ1, D10 | |
| 54 | Smaller-from-larger | CM | |
| 50 | Borrow–from–0 and left-10-OK | DZ1 | |
| 34 | 0 − N = N and move-over-0-borrow | DZ2 | ZNN |
| 14 | 0 − N = N and stops-borrow-at-0 | DZ1, DZ2 | ZNN |
| 13 | Smaller-from-larger and 0 − N = 0 | CM | ZNZ |
| 12 | 0 − N = 0 and move-over-0-borrow | DZ2 | ZNZ |
| 11 | Borrow-from-0 and N − 0 = 0 | DZ1, D10 | NZZ |
| 10 | 0 − N = 0 and N − 0 = 0 | | ZNZ, NZZ |
| 10 | Borrow-from–0 and 0 − N = N | DZ1, D10 | ZNN |
| 10 | Move-over-0-borrow | DZ2 | |
| 10 | N − 0 = 0 | | NZZ |
| 10 | 0 − N = N | TEN | ZNN |
| 9 | 0 − N = N and Left-10-OK | | ZNN |
| 8 | Borrow-from-all-0 | PNZ | |

[a] Number of children (out of 1325) who consistently exhibited the bug.

## References

[Anderson Boyle and Yost 85]
Anderson J. R., C. F. Boyle, G. Yost, "The Geometry Tutor", *IJCAI Proceedings*, 1985.

[Bregar and Farley 83]
Bregar, W. S. and A. M. Farley, "Knowledge Sources for an Intelligent Algebra Tutor", unpublished manuscript, 1983.

[Brown and Burton 78]
Brown, J. S. and R. R. Burton, "Diagnostic Models for Procedural Bugs in Basic Mathematical Skills", *Cognitive Science* , 2, p. 155-192, 1978.

[Brown and VanLehn 80]
Brown, J. S., and K. VanLehn, "Repair Theory: A Generative Theory of Bugs in Procedural Skills", *Cognitive Science*, 4, p. 379-426, 1980.

[Brown, Burton and deKleer 82]
Brown, J.S., R.R. Burton, and J. de Kleer, "Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II, and III", *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown (eds), Academic Press, New York, 227-282, 1982.

[Burton 82]
Burton, R. R., "Diagnosing bugs in a simple procedural skill", *Intelligent Tutoring Systems* , D. Sleeman and J. S. Brown (eds), Academic Press, New York, 157-185, 1982.

[Chambers and Sprecher 83]
Chambers, J. A., and J. W. Sprecher, *Computer Assisted Instruction - Its Use in the Classroom*, Prentice Hall, New Jersey, 1983.

[Clancey and Buchanan 82]
Clancey, W. J. and B. Buchanan, *Exploration of Teaching and Problem Solving Strategies, 1979-1982,* Report No. STAN-CS-82-910, Department of Computer Science, Stanford University, 1982.

[Colbourn (Jones) 83]
Colbourn (Jones), M., "Computer-Guided Diagnosis of Reading Difficulties", *Australian Journal of Reading,* Vol.6, No. 4, pp 199-212.

[Colbourne (Jones) and McLeod 84]
Colbourn (Jones), M. and J. McLeod, "Computer Guided Educational Diagnosis: A Prototype Expert System", *Journal of Special Education Technology,* Volume VI, Number 1, Winter 1984.

[Colbourn (Jones) and McLeod 83]
Colbourn (Jones), M. Jones and J. McLeod, "The Potential and Feasibility of Computer-Guided Educational Diagnosis", *Information Processing,* 83 R.E.A. Mason (ed) Elsevier Science Publishers, B.V. (North Holland) pp. 891-896, 1983.

[Goldstein 82]
Goldstein, I. P., "The Genetic Graph: a representation for the evolution of procedural knowledge", *Intelligent Tutoring Systems*, D. Sleeman and J.S. Brown (eds), Academic Press, New York, pp. 51-77, 1982.

[Jones and Poole 85]
Jones M. and D. Poole, "An expert system for educational diagnosis based on default logic, *Proceedings of the Fifth International Workshop on Expert Systems and their Applications*, May 13-15, Palais des Papes, Avignon, France, pp. 573-583, 1985.

[Jones, Poole and Wasson 85]
Jones, M., D. Poole and B. Wasson, "Student Models: the Genetic Graph Approach", unpublished manuscript, 1985.

[Laurillard 83]
Laurillard, D. M., "Styles of Computer-Based Learning and Training", *Computer based learning - State of the Art Report*, Pergamon Infotech Limited, Maidenhead, Berkshire England, 1983.

[Papert 80]
Papert, S., *Mindstorms:children, computers and powerful ideas*, Harvester Press, 1980.

[Peachey 83]
Peachey, D. R., "An Architecture for Plan-Based Computer Assisted Instruction", MSc Thesis, Department of Computational Science, University of Saskatchewan, Saskatoon Saskatchewan, 1983.

[Poole 84]
Poole, D. L., "A Logical System for Default Reasoning", *AAAI Workshop on Non-Monotonic Reasoning* NY, October 1984, pp.373-384.

[Poole,Aleliunas and Goebel 85]
Poole, D., R. Aleliunas, and R. Goebel, "Theorist: a logical reasoning system for defaults and diagnosis", Logic Programming and Artficial Intelligence Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1985.

[Reiser, Anderson and Farrell 85]
Reiser, B. J., J. R. Anderson and R. G. Farrell, "Dynamic Student Modelling in an Intelligent Tutor for LISP Programming", *IJCAI Proceedings 1985*, pp. 8-14.

[Reiter 80]
Reiter, R., "A Logic for Default Reasoning", *Artificial Intelligence*, Vol 13 pp, 81-132, 1980.

[Rushby 83]
Rushby, N. J., *Computer Based Learning - State of the Art Report*, Pergamon Infotech Limited, Maidenhead, Berkshire, England, 1983.

[Self 74]
Self, J. A., "Student Models in Computer-Aided Instruction", *International Journal of Man Machine Studies*, (1974) 6, pp. 261-276.

[Sleeman 83]
Sleeman, D., "Assessing aspects of basic competence in algebra", *Intelligent Tutoring Systems*, D. Sleeman and J. S. Brown, eds., Academic Press, New York, 1983.

[Sleeman and Brown 83]
Sleeman D. and J. S. Brown, eds., *Intelligent Tutoring Systems*, Academic Press, New York, 1983.

[Steinberg 84]
Steinberg, E. R., *Teaching Computers to Teach*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, London, 1984.

[Suppes 69]
Suppes, P., "Computer Technology and the Future of Education", *Computer-Assisted Instruction*, ed., R. C. Atkinson and H. A. Wilson, Academic Press, New York, 1969, pp. 41-48.

[Suppes and Morningstar 70]
Suppes, P. and M. Morningstar, "Four Programs in Computer Assisted Instruction", *Computer-Assisted Instruction, Testing, and Guidance*, ed. W. H Holtzman, Harper and Row, New York, 1970, pp. 233-265.

[Wasson 85]
Wasson, B. J., "Student Models: The Genetic Graph Approach", Research Report CS-85-10 May 1985, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

[Yob 75]
Yob, G., "Hunt the Wumpus", *Creative Computing*, September/October, pp. 51-54, 1975.

[Young 70]
Young, G. S., "Comments on Social, Psychological, and Mathematical Aspects of the Suppes-Morningstar Chapter", *Computer-Assisted Instruction, Testing and Guidance*, ed., W. H. Holtzman, Harper and Row, New York, 1970, pp. 266-276.

[Young and O'Shea 82]
Young, R. M., and T. O'Shea, "Errors in Children's Subtraction", *Cognitive Science*, Vol II, pp. 153-177, 1982.