IDENTIFYING SOLUTION PATHS IN COGNITIVE DIAGNOSIS

Stellan Ohlsson
Pat Langley

CMU–RI–TR–85–2

The Robotics Institute
Carnegie–Mellon University
Pittsburgh, Pennsylvania 15213

March 1, 1985

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

Stellan Ohlsson's current address is: Learning Research and Development Center, University of Pittsburgh, Pittsburgh, Pennsylvania 15260. Pat Langley's current address is: Department of Information and Computer Science, University of California, Irvine, California 92717. This research was carried out while both authors were at The Robotics Institute, Carnegie–Mellon University.

unclassified

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

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Technical Report No. 2	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Identifying Solution Paths in Cognitive Diagnosis		5. TYPE OF REPORT & PERIOD COVERED Final Report 11/83-11/84 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* Stellan Ohlsson Pat Langley		8. CONTRACT OR GRANT NUMBER*(s)* N00014-83-K-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS The Robotics Institute Carnegie-Mellon University Pittsburgh, Pennsylvania 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 154-508
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research Arlington, Virginia 22217		12. REPORT DATE 1 March, 1985 13. NUMBER OF PAGES 48
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

cognitive diagnosis
problem spaces
production systems
machine learning
heuristic search
subtraction

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

OVER

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

unclassified

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

ABSTRACT

In this paper, we examine the problem of cognitive diagnosis – inferring explanations of observed human behavior. We review previous approaches to cognitive diagnosis, focusing on the methodology of Newell and Simon (1972), which relies on the Newell (1980) problem space hypothesis. This approach proceeds in three stages: (1) identify the problem space in which the subject is operating; (2) identify the subject's path through the problem space; and (3) identify a set of rules that account for the steps along this path. We recount earlier work on automating the diagnostic process, and focus on methods for automating the second of the stages above – identifying the subject's solution path. We describe the Diagnostic Path–Finder (DPF), an AI system that uses heuristic search to hypothesize a solution path to account for a subject's behavior. Unlike most heuristic search systems, DPF employs psychological criteria to direct its search through the problem space. We examine DPF's performance in explaining errors on multi–column subtraction problems, and compare its explanations to those generated by Brown and Burton's (1978) DEBUGGY system. Finally, we discuss the advantages and limitations of our approach to automated cognitive diagnosis.

TABLE OF CONTENTS

LIST OF TABLES

Introduction

Consider the problem of describing the thought processes of a human being with respect to how he solves a particular task. How can we infer his knowledge of the task and the procedure or strategy he is using to solve it from his performance? We will call this the problem of *cognitive diagnosis.*

The problem of cognitive diagnosis can be seen from several different points of view. To the cognitive psychologist, it is the problem of methodology: How should one go about constructing theories to account for empirical data? To the teacher, it is the problem of understanding the misconceptions of students in order to plan remedial instruction. To the Artificial Intelligence researcher involved in the construction of intelligent user interfaces, it is the problem of what information the program should collect about the user, and how it should use that information.

We have no proof that the problem of cognitive diagnosis has a principled, general solution. In other words, we do not know whether there exists a justifiable procedure which can take a behavioral record from an arbitrary task domain, and compute a description of the mental processes which account for the recorded behavior. On the contrary, sophisticated arguments have been produced in support of the idea that cognitive diagnosis is always impossible in principle (Anderson, 1984). However, these arguments have had little impact on the activities of cognitive scientists. The general working assumption of the field seems to be that the difficulties will eventually be overcome one way or the other. Our own research shares in this working assumption.

In most contexts, the term "diagnosis" stands for a semi–formal procedure which is usually carried out "by hand", and which may depend to a considerable extent on the good judgment of the expert doing the diagnosis. This is certainly true for medical diagnosis, which is the paradigmatic example of diagnosis. But it is also true of other instances, such as trouble shooting of electronic machinery. Until recently, it has also been true of the few scattered efforts to construct a methodology for cognitive diagnosis.

However, during recent years, a handful of automatic systems for cognitive diagnosis have been constructed (Brown and Burton, 1978; Sleeman and Smith, 1981; Langley, Ohlsson, and Sage, 1984). A computerized system for cognitive diagnosis has considerable appeal, both for research and practical applications. The research to be reported here shares the ambition to produce a computerized diagnostic system.

The first section below reviews the efforts to produce workable methodologies of cognitive diagnosis to date, and explains why we do not think they are sufficient. The following section puts our own approach to cognitive diagnosis in relation to the others. In the third section, we describe a computer program, the Diagnostic Path–Finder (DPF), which performs (a part of) the task of cognitive diagnosis. In the fourth section, we report on the results from running the DPF program both on ideal and on empirical data. In the final section we discuss a number of issues raised by our explorations with DPF.

A Review of Cognitive Diagnosis

The problem of cognitive diagnosis has a somewhat paradoxical status. To a layman it seems as if the very basis of psychological expertise is to be able to infer a person's thoughts and wishes from his words and actions. The belief that psychologists are capable of doing this is the source of the fear that members of this profession sometimes inspire in laymen. However, the problem of (cognitive) diagnosis has not received much attention within psychology. The two main trends in psychological research, Experimental Psychology and Test Psychology, do not pose the problem of how to treat empirical observations in terms of diagnosis. Let us consider briefly the reasons for this.

First, Experimental Psychology has not emphasized theories of mental processes. The main goal of research in this tradition is to reduce the variation in some quantitative aspect of performance to the variation in one or more stimulus variables. The problem of describing the mental processes which account for the performance is not central. Second, Experimental Psychology has not been interested in detailed descriptions of individual persons, focusing instead on generalities. Third, Experimental Psychology has almost exclusively used quantitative data which are treated statistically. Since standard methods for statistical treatment of measurements have been available for 50 years or more, the question of what one should do with psychological observations, and by what canons they should be interpreted, has receded into the background. Instead of being a substantial question, the answer to which depends on psychological theory, data treatment tends to be viewed as a technical question which is answered with the help of statistical, rather than psychological, expertise.

Unlike Experimental Psychology, Test Psychology has an explicit interest in interindividual differences. But this interest is limited in several ways. First, Test Psychology does not aim for a theoretical description of the individual, or of the mental processes responsible for a test performance, being content to describe the performance empirically, with a so–called test score. Second, Test Psychology conceptualizes test scores as indicative of stable traits or properties of the individual, rather than as the outcome of a performance at a particular moment in time. Third, like Experimental Psychology, this branch of psychology has mainly been interested in quantitative, rather than structured, descriptions.

Theoretically, both Experimental Psychology and Test Psychology have elected to ignore the content of individual thought processes. The goal of capturing that which is general across different contents was taken to imply that thought content should be ignored. This further weakened the interest in, and the need for, detailed descriptions of mental processes.

The upshot of these philosophical, theoretical, and methodological stances is that the two main streams of research psychology during the present century have virtually ignored (what one would expect to be) the most basic of all psychological skills – the ability to infer what a person is thinking on the basis of how he behaves. This problem has only been acknowledged within what is sometimes called the counter traditions (Anderson and Bower, 1973). However, as the following brief review reveals, very little real progress was made towards a well–specified methodology for cognitive diagnosis until the advent of the

information processing approach.

The most powerful counter tradition until the sixties was, of course, the clinical tradition. Psychoanalysis gets its distinctive flavor to a large extent from the various diagnostic techniques pioneered by Freud and his disciples: free association, dream analysis, and projective tests. These methods are content–oriented, and aim at a detailed description of the individual person. However, the resulting description is primarily focused on the emotional experience of the individual, rather than on his knowledge. Also, Psychoanalysis shares with Test Psychology the ambition to find out about stable properties of the individual, properties of his mental life which are independent of the particular tasks that he encounters. Indeed, these methods minimize the influence of the task in the diagnostic situation. They are therefore of less interest to cognitive psychology, which studies task–oriented thinking.

A second counter tradition was that of Gestalt Psychology. The Gestalt psychologists rejected Experimental Psychology on the basis of what they regarded as major theoretical and philosophical errors. Instead, they made diligent use of qualitative data, and aimed for detailed descriptions of mental processes. However, they showed little interest in methodology, perhaps programmatically so. As a result, they never developed a set of methodological canons. The empirical studies of Duncker, Wertheimer, and Kohler stand as a set of examples of how one can arrive at a description of thought processes on the basis of an observed performance, but the methodological principles involved were never made explicit.

In contrast, the Piagetian tradition can be said to have begun with a methodological stand: Having worked on a project of psychological testing, Piaget decided that he wanted to study other questions, for which the methods of Test Psychology seemed inappropriate. As is well known, what then followed was a long series of studies, first with extensive verbal interaction with children, later with the main emphasis on observing the performance of the child on a cleverly designed set of diagnostic tasks.

There has been some effort to formalize Piagetian diagnosis (Smedslund, 1969), but the Piagetian approach to diagnosis is competence oriented. The diagnosis is intended to reveal whether a particular knowledge item is present or not, or whether the person really understands a particular principle or "has" a particular concept or not. Although the observations collected in this tradition are detailed, the final description of the subject is rather course. The formal models used by Piaget are quite abstract. In short, Piagetian methods are not process–oriented.

The first successful attempt to produce a well–specified methodology for the diagnosis of cognitive processes was that of Newell and Simon (1972). Since our own approach to cognitive diagnosis builds extensively on theirs, we will discuss their methodology in some detail.

The Newell–Simon diagnostic method takes a single performance as its unit of analysis. The aim is to arrive at a description of the thought processes behind that performance in the form of a formally specified procedure which can re–create that performance, a so–called simulation model.

The basic principle of their approach is that in order to do cognitive diagnosis, one needs *temporally dense data.* In other words, their methodology prescribes the use of data from process–oriented methods – which monitor a performance as it unfolds, rather than simply record some global property of the performance such as the answer, the correctness of the answer, the time to solution, etc. The most important example of a process–oriented method is the think–aloud method (Ericsson and Simon, 1984), but Newell and Simon (1972) also showed how their methodology could be used with eye–movement recordings. Others have shown how to apply it to recordings of motor actions (Young, 1976).

Another basis for the Newell–Simon approach to cognitive diagnosis is the *problem space hypothesis.* This is a theoretical principle which says that cognitive processes take the form of search through a problem space. A problem space is defined through (a) a notation which encodes those properties of the task that the problem solver takes into account, (b) a list of basic cognitive operators which the problem solver uses to process the task, and (c) a termination criterion which the problem solver uses to decide that he has finished the task. Within the problem space hypothesis, a representation of a thought process takes the form of a strategy for searching the relevant problem space.

The problem space hypothesis implies that the task of cognitive diagnosis consists of three sub–tasks:

- *Identify the problem space.* Scan the protocol for evidence of those aspects of the problem to which the subject attends, for traces of the operators he is using, and for his termination criterion. The resulting space is an hypothesis about how the subject views the task.
- *Having settled on a problem space, map the behavioral record onto a path through that space.* The resulting path (sequence of operator applications) is an hypothesis about which mental steps the subject went through.
- *Invent rules which can account for the individual steps along the path.* The resulting set of rules is a description of the regularities in the solution path, and thereby an hypothesis about the procedure or strategy which the subject used to solve the task.

The strategy hypothesis is evaluated in several ways. For instance, one can take the steps along the solution path which are successfully explained by the rules in proportion to the various kinds of errors. Another evaluation method is to see to what extent the procedure abstracted from one performance by a particular subject can simulate another performance from the same subject (Ohlsson, 1980).

The Newell–Simon method was proposed in the context of research on problem solving. However, the problem space hypothesis is a quite general hypothesis about cognitive processes, and process–oriented data can be collected from any kind of performance (except the very fast ones). Thus, the methodology has great generality. However, it has not been extensively used, in spite of the frequent use of verbal protocols in recent years.

Williams and Hollan (1981) have put forward a method for the analysis of verbal protocols from a long–term memory retrieval task. This method shares several important characteristics with the Newell–Simon methodology. It presupposes protocol data, and it

attempts to capture the regularities in such data in a formal system intended to elucidate the mental processes behind the protocol. As far as we know, this method, too, has been ignored by most other investigators in the field.

The Newell–Simon method is a diagnostic method in the classical sense that although it structures the diagnostic task into distinct steps, it is nevertheless dependent on the good judgment of the psychologist who is using it. Some effort has been put into the automatic analysis of think–aloud protocols (Waterman and Newell, 1972). The idea is to have a computer program which takes the think–aloud protocol as input, and goes through the diagnostic steps discussed above. Some progress has been made towards such a system, but as far as we know, no such system is currently in use. The main difficulty seems to be that in order to construct a problem space on the basis of a think–aloud protocol, the computer program must understand the utterances in the protocol. Thus, future advances in natural language understanding by computers may put the construction of systems for automatic protocol analysis back on the research agenda.

The next major advance in cognitive diagnosis came from the field of Artificial Intelligence (AI), rather than from psychology. Brown and Burton (1978) presented DEBUGGY, a computer program which made cognitive diagnoses in elementary school arithmetic (and related domains). When children learn arithmetic, they usually pass through a stage where they get some problems right, and others wrong. In other words, they know something about how to do the task, but they do not yet know the arithmetic procedure the teacher has been trying to teach them. One hypothesis about such superficially inconsistent performances is that the child is executing an arithmetic procedure with one or more "bugs" or errors. For example, if the child has not understood that he should decrement the number from which he borrows during subtraction, he will perform correctly on non–borrowing problems, while producing wrong answers on borrowing problems.

Brown and Burton (1978) pointed out that remedial instruction could benefit from detailed description of the exact procedure the child is using. Therefore, it would be very useful to be able to identify a student's errorful procedure on the basis of his answers. Brown and Burton proceeded to write a computer program, DEBUGGY, which was capable of making such diagnoses in the domain of multi–column subtraction. DEBUGGY represents the first fully formalized method for cognitive diagnosis.

The diagnostic method of DEBUGGY presupposes considerable pre–analysis of the possible procedural errors:

- Select some procedure for the relevant task domain as the correct procedure. In the case of school subjects like subtraction, the correct procedure is simply the procedure the teacher has been trying to teach.
- Analyze this procedure into a set of components or "subroutines".
- For each component, construct a list of erroneous or "buggy" versions of that component.

The list of all erroneous versions of the components we will call a bug library. Given a particular bug in the library, the construction of the corresponding buggy procedure is

simple: replace the relevant component of the correct procedure with the erroneous version. Given the correct procedure and the bug library, a large number of buggy procedures can be generated by "inflicting" one or more bugs on the procedure. The buggy procedure can then be run on a set of problems in order to predict which answers should be observed for a person who is "suffering" from that bug (or combination of bugs).

In order to diagnose a particular person, one finds a set of bugs such that the resulting procedure predicts that person's answers. This can be done in several different ways. In principle, one can go through all bugs in the bug library in a Generate–and–Test fashion. However, the bug library can be very large. For the subtraction domain, more than a hundred distinct procedural errors have been identified (VanLehn, 1982). Also, a child may "suffer" from a combination of two or more bugs, leading to a combinatorial explosion in the simple–minded approach. Thus, the Generate–and–Test strategy, although possible in principle, is not practical. Instead, the DEBUGGY system was equipped with a discrimination net which sorted a child (i.e., a performance) into the right bug. This mechanism also allowed DEBUGGY to consider several different diagnoses for the same performance. This is particularly important for those cases in which no diagnosis covers a child's performance perfectly, so that the final diagnosis has to be made by choosing between several diagnoses, each of which is only partially successful in accounting for the observed answers.

A very similar diagnostic method has been used in Sleeman and Smith's (1981) LMS, a system for the diagnosis of algebra behavior. A library of algebra bugs, here called "mal–rules", is set up on the basis of empirical research, and a computer program finds out which combination of correct rules and mal–rules solves the algebraic tasks in the same way as some particular child. The main difference between the LMS and the DEBUGGY systems seems to be the different representations used for the procedures: a Lisp–like hierarchy of procedure calls in the case of DEBUGGY and a set of production rules in the case of LMS.

Why is the bug library method successful? Its most salient feature is that it ignores the main methodological insight of the Newell–Simon contribution – that cognitive diagnosis demands temporally dense data. The DEBUGGY system does not need process–oriented data in order to do diagnosis. Why is this? We believe that there are three features of the subtraction domain which makes it possible to do cognitive diagnosis based only on answer–data.

First, in a subtraction task, the goal–object (the sought–for number) is not given with the statement of the problem. In this aspect, arithmetic in general differs from the classical problem solving puzzles often used in both AI and psychological research. In the typical puzzle, there is a situation or state of affairs which the problem solver is to achieve. This means that the problem solver can easily determine the correctness of his answer; he needs only to compare the current situation with the goal situation as described in the problem statement. It also means that answer–data are meaningless. The goal–object does not contain any information about the problem solver.

Second, in the subtraction domain, the goal–object is chosen from a very large set of objects. This contrasts with many tasks used in studies of verbal reasoning and question–

answering, where there is a small set of objects which constitute the possible universe of answers. Thus, answer–data for subtraction tasks can be very discriminating.

Third, the answer to a subtraction problem consists of parts, the individual digits, which are organized in a linear ordering. Thus, the answer has internal structure. This is in contrast to many tasks used in psychological research in which the answer is an atomic symbol, such as a name, a word, or a classification.

These three aspects, that the answer is not given with the problem, that it has to be chosen from a very large space of possible answers, and that the answer has internal structure, implies that there is a lot more information about the problem solver in a subtraction answer than in answers to most cognitive tasks. This is why cognitive diagnosis can succeed in this domain even in the absence of process–oriented data. We notice that some algebra tasks share in these properties, but that proof–tasks do not (since the goal object is given with the statement of the problem).

The bug library approach to automatic diagnosis has a proven generality across two dissimilar task domains, further strengthened by the recent application of DEBUGGY to errors in fractions (VanLehn, personal communication). However, each instance of the method is limited to the domain for which the particular bug library was designed. One cannot make diagnoses for algebra tasks with the bug library developed in connection with subtraction, or vice versa. A bug library must be constructed for each task domain where the method is to be applied. The two existing bug libraries (VanLehn, 1982; Sleeman, 1983) have been constructed on the basis of extensive empirical investigations. Over the long haul, one can imagine psychologists having access to standard bug libraries for all task domains in which cognitive diagnosis is frequently done, such as the various school subjects.

However, there is something conceptually disturbing about a bug library, over and above its limited generality across task domains, and the large amounts of work involved in constructing it. How, we must ask, can one construct a bug library without using a diagnostic procedure? The bugs in the two published bug libraries have been found empirically. But what can "finding a bug empirically" mean except to go from the observed performance to the bug which accounts for it? But that *is* the diagnostic inference. Thus, there seems to be a paradox lurking here: Either a bug library is necessary for doing diagnosis, in which case the bugs in the library cannot be found empirically; or else there is some way of doing diagnosis which does not presuppose a bug library, in which case the library is unnecessary.

The two extant bug libraries have presumably come about through a combination of a priori analyses and intuitions informed by large amounts of empirical observations. This leaves us with a set of bugs which have a high probability of being relevant for human performance. However, we have no information about the entire space of possible bugs, or about what part of that space the bug library represents. There is no principled way to answer questions like the following:

- Can the bug library be extended or not?
- If the diagnostic system fails to find a systematic diagnosis for a particular perfor-

mance, could that performance be diagnosed with an extended bug library, or is the performance irregular?

- If the bug library was to be extended, would it alter any of the diagnoses which the system has already performed?

Furthermore, it seems plausible that an adult is limited by his own expertise with respect to subtraction in his thinking about possible subtraction bugs, but we have no way of systematically identify such limits. In short, the problem with the bug library method is that the diagnostic system is exactly as good as its bug library, but we have no systematic method for determining the quality of the bug library.

We also notice that the bugs in a bug library are erroneous versions of distinct components of the correct procedure for the domain. This means that the bug library is relative to a particular correct procedure. Also, it means that the bug library is relative to the particular representation of procedures used. In a different representation, the correct procedure will come apart into a different set of components, implying a different bug library. Thus, the bug library method makes theoretical commitments which are somewhat obscured by the practice of publishing only natural language paraphrases of the bugs.

To summarize, mainstream psychological research during this century has not had a great interest in cognitive diagnosis, basically because there was no interest in detailed descriptions of the thought processes of individual persons. The various counter traditions that did exhibit such an interest either failed to formalize their diagnostic methods or focused on something else than processes. Real progress in cognitive diagnosis came about through the method of Newell and Simon (1972) for the analysis of the think–aloud protocols. The problem space methodology has great generality, but so far no one has produced a working computer system which can use this method, largely due to the difficulties in computerizing the understanding of the natural–language utterances in a verbal protocol. In contrast, the bug library approach taken in the DEBUGGY and LMS systems allow fully automatic diagnostic systems to be built. However, it is unclear how bug libraries should be created and evaluated. In the next section we describe our own approach to cognitive diagnosis.

The Machine Learning Approach to Cognitive Diagnosis

The approach to cognitive diagnosis to be discussed here follows the main outline of the diagnostic method of Newell and Simon (1972). To recapitulate, the Newell–Simon method takes the question "Which procedure did the subject/student execute?" and decomposes it into three sub–questions:

- Which problem space did the subject/student work in? This question is answered by scanning the think–aloud protocol for information about notation and operators.
- Which solution path did the subject/student take through the problem space? This question is answered by mapping the protocol step by step onto the problem space.
- Which procedure did the subject/student use to generate the solution path? This question is answered by inventing production rules which account for the successive

steps in the path, while trying to keep both the number of rules and the number of errors as small as possible.

Our method works with these three questions, but uses other techniques to answer them. The differences have two sources: First, we are aiming for a diagnostic system which can solve the same tasks as the DEBUGGY and LMS systems; thus, the system is to work on answer–data, rather than on think–aloud protocols or other process–oriented data. The diagnostic method has to be adapted to this fact. Second, we are making use of AI results from the field of machine learning which were not available at the time Newell and Simon developed their methodology.

The three questions above will be discussed one at a time. It turns out that it is easier to discuss them in reverse order. Thus, we begin with discussing the last step, that of going from a set of solution paths to a procedure which is capable of re–creating them.

From Solution Paths to a Procedure

Let us assume that the first and second diagnostic tasks have been carried out with respect to a particular set of performances. We then have a set of path hypotheses. The final step to a computer simulation of those performances is to find a procedure which will generate those paths.

A similar problem has been studied in the field of machine learning, under the name of strategy acquisition or heuristics learning. Researchers in this area have constructed a number of learning systems have been that consist of two components – a problem solver and one or more learning mechanisms. The problem solver attacks a problem with some general, weak search method (such as exhaustive search), eventually producing a solution path. The information in the solution path is used by the learning mechanisms to propose some (task–specific) improvement in the problem solver (Mitchell, Utgoff, and Banerji, 1983; Ohlsson, 1983; Langley, 1983; Anzai and Simon, 1979; Hagert, 1982). For instance, the learning component might generalize over the good steps, it might discriminate between good and bad steps, and so forth.

The AI approach to this problem normally assumes that the solution paths produced by the problem solver and used by the learning mechanisms represent *correct* solutions. However, the typical learning mechanisms, such as generalization and discrimination, do not make any use of the intended interpretation of the paths or the procedures they generate. Therefore, they are neutral with respect to correctness. Our approach to the problem of generating procedures from solution paths assumes that AI techniques for automatic strategy acquisition can also work on *incorrect* solution paths, and thereby produce incorrect or "buggy" procedures. This assumption has been supported in runs with a previous diagnostic program called ACM (Langley, Ohlsson, and Sage, 1984; Ohlsson and Langley, 1984; Langley and Ohlsson, 1984).

ACM carried out a breadth–first search through a problem space, finding the shortest solution path accounting for the observed answer. This process was repeated for a number of problem–answer pairs. Operator instantiations lying along a solution path were marked as positive instances of the responsible operator, while instantiations leading one step off of

the path were marked as negative instances. After collecting the set of positive and negative instances for each operator, ACM passed each set to a nonincremental discrimination learning mechanism. For each operator, this method generated a set of conditions that covered all positive instances but none of the negative instances. When these conditions were added to the original conditions for each operator, the result was a production system model capable of reproducing (and thus explaining) the inferred solution path.† ACM's main limitation lay in its method for inferring these solution paths, and our current research has focused on remedying that limitation. Accordingly, let us now turn to this issue.

From a Problem Space to Solution Paths

Suppose that we have identified the problem space used by a subject/student. Our basic technique for finding the solution paths corresponding to an observed (typically incorrect) answer is to do a search through that problem space. Let us pause and consider why search is an appropriate technique. Search is usually used to find the correct answer to some problem. How can we search for an incorrect answer?

Consider a problem space for, say, subtraction. If we develop that space exhaustively, we are guaranteed to find the correct answer. Since every possible sequence of operators have been developed, by definition of "exhaustive search", the particular sequence of steps which lead to the correct answer must have been developed. However, by the same argument, the particular sequence of steps leading to a certain incorrect answer must have been developed as well. In fact, every terminal node in that subtraction space, except those containing the correct answer, represents some incorrect solution to the problem. We will say that the problem space *contains* each of the procedural errors that is represented by some terminal node in the space.

Our first hypothesis about how to find the solution path corresponding to a particular incorrect answer was to use exhaustive search in the following way:

1. Develop the entire problem space.
2. Go through the terminal nodes until one is found which contains the incorrect answer.
3. Collect the solution path by traversing the search tree backwards from the terminal node to the initial node.

The above method is attractive for several reasons. First, it is quite general. It can be applied to any performance which can be cast as a search through a problem space. Second, it is foolproof. If the problem space contains the particular error, the method is guaranteed to find a solution path. Third, it does not make any theoretical commitments other than the problem space hypothesis.

However, exhaustive search is a dangerous technique. We originally believed that

† Our work on ACM has been strongly influenced by Young and O'Shea's (1981) production system analysis of subtraction errors. However, our approach differs in relying explicitly on the problem space hypothesis, while Young and O'Shea viewed subtraction in algorithmic terms.

problem spaces for simple task domains, such as multi–column subtraction, were small enough so that exhaustive search could be practical to use. Experience has shown that this is not so, even when partial answer information is used to eliminate branches in the search tree. We therefore abandoned the idea of exhaustive search, in spite of its advantages.

The obvious alternative to exhaustive search is, of course, selective search. Let us consider what this involves. To search the space selectively means that we do not develop all paths until they end in a terminal node. Some paths, the more the better, are rejected before they are fully developed. In order to do this, we must have some criteria by which we can recognize one path as a more fruitful or psychologically plausible hypothesis than another, before either path is fully developed. Expressed differently, a selective search presupposes a formal definition of what is meant by a good psychological hypothesis. Thus, selective search in this case requires more theoretical commitments than exhaustive search.

Our ambition is to construct a system which is domain–independent. The criteria for what is to be regarded as a good path hypothesis should therefore not make use of the particular properties of any one task domain, but be general criteria for the plausibility of path hypotheses. Thus, we must consult psychological theory for criteria by which one can judge one psychological hypothesis as more plausible than another.

Depending upon the set of criteria used, one can search a problem space for an erroneous answer in different ways. The particular selective search that we have implemented will be discussed in a later section.

From a Task to a Problem Space

In the two previous sections, we described our approach to (a) generating a procedure from a set of solution paths, and (b) finding the relevant solution paths, given a problem space. To complete the methodology, we need some way of finding the relevant problem space.

In the Newell and Simon methodology, the problem space is identified by scanning the think–aloud protocol for evidence as to which aspects or properties of the problem the subject is using. The paradigmatic example here is the observation from the cryptarithmetic domain that some subjects made use of the parity property of numbers, while others did not, a fact that can be seen clearly in the protocols (Newell and Simon, 1972). Also, the protocol is scanned for evidence concerning the operations by which new partial or intermediate results are derived or computed.

Since we are aiming for a system which should be capable of working on answer–data, this technique cannot be used. We must identify the problem space in some other way. Currently, we have no principled solution to this problem.

Hayes and Simon (1974) have described UNDERSTAND, a computer program which inputs a task description, and constructs a problem space for that task. The criterion for correctness is that the resulting space can be passed on to a problem solver which then solves the problem. However, the UNDERSTAND program does not handle interindividual

differences. In other words, it constructs one problem space for each task. But any one task can be solved in a variety of problem spaces. We need to find the particular problem space which is used by some particular subject/student. Thus, the space cannot be a function of the task only.

Newell (personal communication) is currently pursuing a new approach to the problem of constructing problem spaces. As far as we know, no results have yet emerged that could be useful to our endeavor.

It is possible to make an empirical study of the problem space used by a particular person. This would involve interviewing him/her as to how he sees the problem, to perform tests to see which operators he has available, etc. However, our diagnostic method is based on the assumption that such an empirical approach is not possible. We want to construct a diagnostic system which can work on a set of answer–data only, with no possibility to extend the empirical information.

Thus, we are left with only one method – a priori analysis. We set up some hypothesis about how the subject/student represents the problem, and what cognitive skills he is using to process the problem that seems reasonable to us. This space is then given as input to the program we describe in a later section. In fact, the program is designed to take a list of problem spaces as input, and to iterate through them, using the one that gives the best diagnosis. However, in the runs to be reported, a single problem space has been used. The particular space will be described in the next section.

Summary

We aim to develop an automated diagnostic system which can map a set of answers onto a computer simulation of the thought processes which led to those answers. We base the system on the problem space hypothesis, which implies that diagnosis consists of three steps: identifying the problem space, finding the relevant solution paths through that space, and inventing a procedure which can re–create those paths. We have no principled solution to the problem of identifying the problem space. We use search through the problem space to find the path terminating with a particular incorrect answer. Finally, we use machine learning methods to invent a procedure which can generate the solution paths. In the next section, we will describe the particular selective search method that we have used in our runs. But first, let us consider the task domain we selected as a testbed, and some problem spaces for that domain.

A Problem Space for Subtraction

We have chosen multi–column subtraction as the first domain in which to try out our diagnostic system. This domain has already been investigated from the point of view of diagnosis. The published bug library (VanLehn, 1982) for this domain gives us a good overview of the kinds of bugs we should expect our system to find. Also, we have available an empirical study of several school children, with complete sets of answers to several

tests, and with the corresponding diagnoses by the DEBUGGY program.† This gives us something to compare the performance of our program against. Finally, arithmetic is a task domain which has inherent interest, especially from a practical point of view.

As discussed in the previous section, we have no principled solution to the problem of identifying the correct problem space. We are therefore left with the possibility of choosing one, and then arguing for its plausibility. In order to put the space we have been using into perspective, we will contrast it with two other spaces – a more basic space and a higher–level space.

TABLE 1

Basic Problem Space for Multi–column Subtraction.

Representation:

Digits in positions within a two–dimensional grid.

Empty positions are possible.

Operators:

MoveEye(direction) Moves the eye one step (i.e., one position) in the given direction. The directions are up, down, left, and right.

WriteDigit(digit, position) Writes a given digit in a specified position.

CrossOut(digit, position) Puts a scratchmark across a digit in a specified position.

FindDifference(n1, n2) Finds the difference between two numbers, the first being at most 19, the second being at most 9.

FindSum(n1, n2) Finds the sum of two numbers, the first being at most 10, the second at most 9.

We assume that the termination criterion for subtraction is that all columns have been processed.

Table 1 summarizes the states and operators the *basic* problem space for multi–column subtraction. This space is basic in the sense that the operators cannot be analyzed into smaller pieces without leaving the area of thinking or problem solving. The first three operators can only be analyzed further by descending to the level of the organization of motor actions. The last two operators represent the addition and subtraction tables, which anyone who is doing multi–column subtraction in schools has presumably learned by heart. Thus, they represent long–term memory retrieval actions, and the further analysis of them is a task for memory theory.

---

† The material has kindly been made available to us by Susan Omansson at the Learning Research and Development Center at the University of Pittsburgh, where the empirical study was carried out. The DEBUGGY diagnoses were computed by Kurt VanLehn.

We would argue that making diagnoses in this basic subtraction space is not reasonable. Recall that the psychological interpretation of an operator in a problem space is that it represents a cognitive skill which the person can carry out or execute without difficulty. But the basic space represents as complex operations processes that we believe most school children actually have available as ready–made skills. For instance, in this problem space decrementing a digit during borrowing would be represented as a complex process consisting of the following steps:

(FindDifference N1 1) ⇒ N2
(CrossOut N1 Pos1)
(MoveEye up) ⇒ Pos2
(WriteDigit N2 Pos2)

But we are willing to believe that the population of subjects relevant for this research – school children about to learn multi–column subtraction – are capable of replacing a digit with the result of subtracting unity from it. Thus, there is no point in the above step–wise representation. Instead, we can make do with the higher–level operator Decrement(digit, position), which performs the above sequence of steps.

A second reason to work with the higher–order operator Decrement is that the above representation has no effect on how the rest of the problem is processed. The child either replaces the digit with another, or else he does not. To consider the internal structure of the decrement operation does not open up any new possibilities for making arithmetic errors. For instance, the child might forget to cross out the decremented digit. However, this does not seem to make any difference for the rest of the solution path, because it does not affect the value being used in the rest of the calculations. Since we are thinking in terms of answer data, it is inappropriate to use a level of analysis in which differences in the solution paths do not give rise to different answers.

A third reason for not working in the basic subtraction space is that one of the operators is too general. The only number added to another number in a subtraction problem is ten. We are willing to believe that children discover this regularity early on. If so, it is unnecessary to include in the subtraction space their general capacity to add numbers. Only the addition of ten is relevant. Thus, instead of FindSum(n1, n2), we can use a AddTen(number, position) operator which replaces a number with the result of adding ten to it.

Finally, the basic subtraction space is inappropriate in that its notation is too fragmented. We would be unwilling to assume that children see the digits of a problem as a set of unorganized entities. Instead, we find it more plausible to assume that they see the digits as organized into rows and columns, as we ourselves do when we look at a subtraction problem. Therefore, the notation should include such entities as the columns, and the three rows with the two subtrahends and the answer, respectively.

In response to these observations, let us introduce a slightly more aggregated problem space, which we will call the *standard* subtraction space. We specify the states and operators for this space in Table 2.

TABLE 2

Standard Problem Space for Multi–column Subtraction.

Representation:

The basic components are digits, columns, and rows.

The columns are designated column–1, column–2, etc.

The columns are numbered from right to left.

The rows are designated top–row, above–row, and bottom–row.

Operators:

Decrement(number, column) Takes a number in the top–row of a certain column, and replaces it with the result of subtracting unity from it.

AddTen(number, column) Takes a number in the top–row of a certain column, and replaces it with the result of adding ten to it.

FindDifference(n1, n2, column) Takes two numbers in a column, and writes the result of subtracting the second from the first in the answer–row of that column.

FindTop(n1, column) Takes a number in the top–row of a column writes that number in the answer–row of that column.

SkipZero(column) Takes a zero in the top–row of a column and puts a "nothing" sign in the answer–row of that column.

ShiftLeft(column) Takes a column as input, and returns the column immediately to the left of it.

ShiftRight(column) Takes a column as input, and returns the column immediately to the right of it.

We believe that this space is appropriate for the diagnosis of subtraction performances. On the one hand, it is difficult to argue that children who are studying multi–column subtraction are typically incapable of executing any of these operators. On the other hand, if we decided to work with even higher–order operators, we would lose the ability to account for a good many procedural errors included in the Brown and Burton bug library.

To illustrate the latter statment, consider a problem space with the following two operators:

ProcessColumn(column) Process any column which does not make borrowing necessary, and shift left to the next column.

BorrowInto(column) Make a correct borrow into the given column.

In this space we are assuming that the subject/student already knows almost everything about multi–column subtraction. Thus, this space is not appropriate for diagnosis, because it contains virtually none of the typical subtraction bugs. Indeed, the only bug from the

Brown and Burton (1978) bug library that can be accounted for in this space is Always–Borrow, in which the child borrows in every column, regardless of whether it is needed or not. This would be the appropriate space if the empirical evidence showed that Always–Borrow was virtually the only mistake children made on subtraction. However, this is not the case. Thus, this space is too aggregated to be useful.

The three spaces we have shown differ mainly in the level of aggregation of the operators. It is reasonable to ask whether there are alternative subtraction spaces that differ along other dimensions. Do there exist a qualitatively different space in which subtraction can be carried out, and which is at approximately the same level of aggregation as the standard space above? If there is, we are unable to think of it.

The Diagnostic Path–Finder

The purpose of this section is to describe a computer program which solves the second of the three diagnostic sub–tasks – that of finding the relevant solution paths, given a problem space, a set of problems, and a set of (possibly incorrect) answers to those problems. We call the program the Diagnostic Path–Finder, or DPF for short. According to the overall approach as presented in the previous section, the DPF is intended to work together with (a) some procedure for identifying the relevant problem space, and (b) some machine learning mechanism which can take the paths and discover a procedure which is capable of re–generating them.

DPF is a general system in the sense that the techniques it uses do not make use of the properties of the particular task domain to which it is applied. All the task–specific information is located in the problem space and in the correct solution paths, which are given as inputs to the program.

There are two main techniques used by the program: a technique which finds the operator applications which are relevant, and a selective search scheme. We will describe each of these techniques in turn.

Instantiating the Operators

The first component of DPF is based on the idea that selective search can be carried out more efficiently if we know in advance which applications of the operators might be involved in the solution path we are looking for. By "operator application" we mean an operator with particular arguments, e.g. (FindDifference 8 2 column–1). The terms "operator application" and "instantiated operator" will be used interchangeably. The question is whether we can identify the relevant operator instantiations ahead of time. It turns out that we can do this by applying the operators backwards.

Briefly, the technique is as follows: For each part of the observed answer (for the subtraction domain, each digit), we compute which applications of the operators in the problem space could have had that partial result as its output. For instance, suppose that the observed answer is "78"; how could the "8" have been produced, given the operators in the standard subtraction space discussed above? There are eleven possibilities:

(FindDifference 17 9 column–1)

...

(FindDifference 8 0 column–1)

(FindTop 8 column–1)

If the child worked in the standard subtraction space, the "8" in column–1 must have been produced in one of these eleven different ways. This assumes that the student has not made a "number fact" error, such as computing 16 – 9 = 8.

Next, we continue the analysis by asking, for each of the arguments of each of these operator applications, how that argument could have been derived. For instance, where could the "8" in the (FindTop 8 column–1) have come from? There are only two possibilities: either the "8" comes from the givens of the problem (i.e., the problem has an 8 as top–digit in column–1 as stated), or else the 8 has been created by applying:

(Decrement 9 column–1)

If the 8 is found among the givens, the derivation is terminated. If not, the same argument is again applied to the 9 in the Decrement operation. The process is repeated until one has considered all possible ways of deriving the digits in the answer from the givens of the problem. We call the result of this process an operator graph.

In order to understand the operator graph, and to understand how it differs from a search–tree, let us extract different objects from it and try to interpret them. First, suppose that we extract a "path" from such a graph, by traversing one link at a time backwards from the observed answer until we reach a termination point. This "path" is *not* a solution path. It does not show how the answer was produced, because it contains no information about the other digits in the answer. Also, it does not show how the particular digit we started with was derived, because it does not explain how the other arguments to the various operators were derived. In fact, such a "path" through the operator graph has no interpretation.

Instead, let us look at the subgraph which "stands" on one of the digits in the answer. That graph represents all possible ways to derive that digit from the givens of the problem. (This is easily seen by imagining the operator graph for a single–digit answer.) Now suppose that we prune this graph by erasing from each node all incoming links but one, beginning with the "lowest" node, and continuing upwards. This pruned subgraph also has an interpretation: It is one possible derivation of that digit. However, notice that the derivation is not represented as a linear ordering of operator applications. The operator applications are only partially ordered.

Thus the operator graph is not the problem space. The interpretation of the single link in the operator graph is not "follows", but "makes possible" or "creates the inputs to". The links show logical dependency, rather than sequential ordering.

The operator graph is, in fact, a compact representation of the problem space. This can be seen by asking what the interpretation of a single node in the operator graph is. A single node in the operator graph stands for an equivalence class of states in the problem space, namely all states in which that application of that operator is legal, but in which

the result of the application is not available. Since such a class of knowledge states can be very large, a small operator graph can summarize a very large problem space.

From the present point of view, the important point is that the operator graph contains all operator applications which could be involved in the derivation of the observed answer from the given problem (assuming we have the right problem space). No other applications of those operators need be considered during the search through the problem space.

What are the prerequisites for this technique? First, we notice that the usefulness of this technique derives from the fact that the answer to a subtraction problem consists of independent "parts" (the digits). If knowledge–states were unitary or atomic, so that operators took a single symbol as input, and delivered a single symbol as output, then this technique would be meaningless. The operator graph would then be identical with the problem space, and the process of constructing it identical with exhaustive, backwards search through the space. Second, notice that the construction of the operator graph presupposes that the operators can be written as functions or processes which take one or more inputs, but which deliver a single expression as output. In most task domains, these two requirements will be fulfilled.

DPF employs a number of tactics to prune its development of the operator graph. Let us consider each of them in turn.

- First, a derivation is continued backwards until some fixed depth, which is a user–controlled parameter. At some distance from the bottom–node, the effort to connect with the givens of the problem is given up as hopeless. For the subtraction domain, we have used 10 as the value for this parameter.
- Second, the user has the option of listing a set of "bad objects" which should not appear as inputs to any operator. Developments which depend on such bad objects are not continued. For the subtraction domain, bad objects would include non–existent columns, such as the fifth column in a three–column problem.
- Third, the development is selective in the sense that it prefers operator applications which occur in the correct solution path above those that do not. For this, we must assume that DPF also has the correct solution path available.
- Finally, DPF tries to connect each development with the givens as soon as possible; in other words, it develops applications with a higher number of given arguments before others.

The operator graph is a very compact representation of the possible ways to derive the observed answer. Problem spaces which contain thousands of knowledge–states can be summarized in an operator graph with, say, 50 operator applications. However, it is important to realize that all possible paths to the observed answer are represented in the operator graph. Thus, the problem of finding the path has not been simplified, since we need only consider a few of the many possible operator applications, but it has not been eliminated. All the different paths are "there" in the graph. The question is which one we should "read out" from the graph. Reading out a path involves (a) selecting which operator applications should be included in the path, (b) deciding how many instances of

each application should be included, and (c) ordering them linearly. The search–scheme we discuss in the next section can be seen as a mechanism for doing this.

Best–First Search through the Problem Space

The purpose of searching through the problem space is to find a solution path which ends in the observed answer. Furthermore, the path should be the psychologically most plausible one. Thus, the problem can be divided into two components. The first of these – deciding on a search scheme – is basically an AI task. The other problem – deciding which path hypothesis is more plausible than another – is basically a psychological one.

In response to the first issue, we are searching the problem space with a best–first search, using the instantiated operators found in the operator graph discussed in the previous section. The search scheme is the standard one: a list of unexpanded knowledge–states is retained. An evaluation procedure iterates through the unexpanded list, and selects one or more nodes which represent the most interesting possibilities. These are expanded, and then removed from the unexpanded list. Their descendents are then added to the unexpanded list, and the process continues. The search continues until a complete path hypothesis has been found (i.e., until some path ends with the observed answer).

In ordinary applications of best–first search, the various alternative paths are evaluated with respect to the possibility that they lead to an answer to the problem. In our application, alternative paths are evaluated with respect to how psychologically plausible they are as path hypotheses.

DPF employs a number of tactics are used to make the search more efficient. First, a one–step look–ahead is used. Paths which can be completed in one more step are always expanded, regardless of the evaluation procedure. Second, the system begins with breadth–first search up to a certain depth, before the evaluation–based best–first search takes over. This is to make certain that the evaluation procedure operates on paths of some minimal length. Obviously, the evaluation criteria cannot rely on paths containing only one or two nodes. Third, new nodes are only compared with the best alternatives from the previous evaluation. Only rarely need the evaluation procedure consider all unexpanded nodes. Fourth, the evaluation criteria are themselves evaluated as the search continues. More discriminating criteria are given more influence over the search than less discriminating criteria. Fifth, the program derives more than one complete path hypothesis, and the final output is selected by running the evaluation procedure one more time on the complete paths only. The exact number is specified by the user.

The Evaluation Procedure

The best–first search is guided by an evaluation procedure which compares the incomplete paths, and selects one or more of them as more likely to lead to a psychologically plausible path hypothesis than the others. Thus, the evaluation procedure amounts to a formal definition of what is meant by a "good hypothesis". The Diagnostic Path–Finder's evaluation procedure uses several different criteria. Let us consider each of them in turn, beginning with three assumptions that constitute a basic platform for cognitive diagnosis

(and for computer simulation in general).

Assumption of Causal Closure:

When DPF expands a node in the problem space, it considers only those operator applications which have all their arguments available in that node. These may be available either because they are part of the givens of the problem, or because they have been produced by some earlier operator. Expressed differently, we are only considering paths in which the arguments to the operators have some rational connection with the task. We do not consider paths in which arguments to operators are generated through imagination (or divine intervention).

It is important to note that this assumption is sometimes false. For instance, a child may get a digit by peeking at his neighbor's worksheet. However, the ambition to account for inputs to operators gives discipline to the diagnostic process (or to any use of computer simulation). If operators can take arbitrary arguments, then any solution path can produce any answer.

Assumption of Purposefulness:

We are assuming that operators are executed because their outputs are needed for some purpose; in other words, they will be used as inputs to some other operator further down the line, or they occur in the final answer. DPF rejects paths in which several intermediate results have not been used by any subsequent operator, even if these paths arrive at the desired answer. In short, we do not consider paths with superfluous steps.

Again, this assumption is sometimes false. It is possible that students are executing all kinds of unnecessary operators, the outputs of which are not used for anything. However, the very fact that the outputs are not used for anything makes it impossible for us to know anything about them. We cannot evaluate hypotheses about extra and unnecessary steps which have no effects on subsequent processing and, in particular, no effect on the answer. Also, the space of such paths is clearly infinite. Thus, we choose to ignore such path hypotheses.

Assumption of No Duplication:

We assume that children do not derive intermediate results which are already available. If an expression is available in a knowledge–state, we do not consider operator applications which generate that expression as output.

Again, this assumption is certainly false in some cases. However, steps which produce already available knowledge–items have no effect on the future processing, since they do not produce any new possibilities for action. Thus, their existence is also impossible to determine. Again, the set of possible paths becomes infinite if we allow repreated derivations of the same result within a path.

Taken together, these three assumptions provide a definition of *rationality*. In practice, these assumptions are presupposed by most cognitive simulations of human behavior. The only major exceptions are computer models which include a long–term memory component. The first principle can then be broken, in the sense that some arguments are produced

by retrievals from long–term memory. (This is not a very plausible alternative for the subtraction domain.)

Note that these three assumptions lead to *absolute* path criteria, rather than relative ones. They are not used to evaluate whether one path hypothesis is better than another. Paths which violate any of these criteria are simply rejected. However, the Diagnostic Path–Finder also employs two kinds of relative criteria: those which are based on psychological theory, and those which are based on other considerations, such as simplicity.

Criterion of Memory Load:

One of the few general principles that has emerged from Cognitive Psychology relates to the limited capacity of humans' short–term memory. There is still considerable debate about the cause of this capacity limitation, but for our purposes, it it is enough to know that such a limitation exists. Given this knowledge, a path hypothesis which requires a smaller short–term memory load seems more plausible than a hypothesis which requires a larger memory load. Thus, given two hypotheses which are comparable on other criteria, DPF prefers the one with the smaller memory load.

We compute memory load as the number of unused intermediate results at each state along the path. An intermediate result which is not used immediately must be kept in memory until it is used. Thus, the number of operator outputs which have not yet been used as inputs to any operator is a reasonable estimate of memory load.

Criterion of Subgoaling:

A second psychological principle of some generality is that human thinking is goal–oriented. We assume that the different parts of the goal (in the case of subtraction, the different digits) are sub–goals for the subject/student. In its evaluation, DPF prefers paths which have more subgoals satisfied.

The user also has the option of specifying some preferred order in which the subgoals are most likely to be satisfied in. For the subtraction domain, we specified right–to–left as the most plausible ordering of the subgoals. During evaluation, DPF prefers paths which have the subgoals in the preferred order, rather than in some other order.

Criterion of Productivity:

A third psychological principle of general importance is that skills of all kinds are continuously being made more efficient through a process called automatization. Several different mechanisms have been proposed to explain how automatization occurs: composition of heuristic rules (Lewis, 1985; Anderson, 1983), chunking of existing skills (Rosenbloom, 1983), and the exclusion of unnecessary steps (Neches, 1981). For present purposes, the exact mechanism behind automatization is not important. It is enough to know that there are processes in human cognition which continually look for short–cuts and weed out unnecessary steps. We take this to imply that during the search we should prefer a path hypothesis with a higher degree of productivity above one with a lower degree. We compute productivity as the number of satisfied subgoals divided by the number of operator applications path (i.e., as "results per effort").

The Diagnostic Path–Finder also uses additional relative criteria related to the notion of parsimony, to which we now turn.

Criterion of Minimal Error:

We want to ascribe to the subject/student as few bugs as possible, consistent with his erroneous answers. This follows one of the oldest methodological canons of science: do not complicate the explanations more than necessary. Given two path hypotheses, DPF prefers the one which more similar to the correct path. (This use of the correct path is one reason why the correct procedure is given as one of the inputs to the program.) We compute the similarity between a path hypothesis and the correct path as the average length of the maximal common subsequences of the two paths.

Criterion of Minimal Length:

A second measure of simplicity involves the *length* of the path hypothesis. Given two path hypotheses, DPF prefers the shorter path, other things being equal. Length is defined as the number of operator applications.

DPF's evaluation procedure applies these criteria in an "additive" fashion. Whenever a new node is generated, its score for each criterion is computed, and the N best nodes are marked. After each criterion has been applied, the nodes to be expanded are taken from the set of marked nodes. The number of nodes to be expanded in each search cycle is a user–settable parameter.

These eight criteria constitute a formal definition of what we mean by a "good hypothesis". Later, we will propose methods for improving DPF that are based on sharpening this definition. In particular, one would wish psychological theory to contribute more principles that could be used as path criteria. Note that the current criteria are quite general. This gives us some hope that they will be useful even in domains other than subtraction.

The use of length in evaluating hypotheses combines in an interesting way with DPF's preference for using more discriminating criteria in making its decisions. First, note that breadth–first search can be construed as best–first search with minimal length as the evaluation function. In a pure breadth–first search, one always expands the nodes at the least depth. As explained earlier, DPF's evaluation criteria are ordered with respect to how well they discriminate between the unexpanded nodes. Suppose that we have a situation in which none of the evaluation criteria other than length can discriminate very well between the various path hypothesis. In that case, none of them will influence the search very much, and the system will fall back on length as the evaluation criterion. Thus, in situations where other evaluation functions are powerless, the best–first search gracefully degenerates into breadth–first search.

Summary

The Diagnostic Path–Finder (DPF) accepts four inputs: a problem space, a set of problems, a correct procedure for those problems, and a set of (typically incorrect) answers to those problems. As output, it delivers an hypothesis about which solution path the subject used to arrive at each of those answers. The program works in two stages. First, it

derives the set of instantiated operators which could possibly be involved in the derivation of the observed answers. Second, it carries out a best–first search through the problem space, using only those instantiated operators. The search is guided by numeric functions that measure the quality of competing path hypotheses. The evaluation procedure employs several different criteria, which together embody a definition of what is meant by a good hypothesis.

Computational Results

The obvious way to evaluate a diagnostic system is to run it on some data, and examine the resulting diagnosis in order to see how well it is doing. However, this approach leads to a conceptual difficulty similar to that facing a person who wants to calibrate a thermometer: how does one calibrate a thermometer without already knowing the temperature (i.e., without applying an already–calibrated thermometer)? How do we know whether a diagnosis is accurate, without having a fool–proof diagnostic method already available?

This problem can only be attacked in a spiral manner. First, we apply the program to cases where we have some grounds for knowing the "correct" diagnosis. When the program performs satisfactorily on those cases, we apply it to other cases, where we do not have any grounds for believing a particular diagnosis.

Of course, there exists a special case in which the correct diagnosis is known with certainty – the case of correct answers. In this situation, the correct solution paths can be derived by simply running the correct procedure. Thus, a minimal condition of adequacy for a diagnostic system is that if when presented with the correct answers for a set of problems, it should generate the correct solution paths.

In the case of subtraction errors, we also have available the considerable results of Brown, Burton, and VanLehn, and we have relied extensively on their analyses in our efforts to evaluate our program. First, we have applied the program to ideal data, in which we "planted" some selected bugs from the Brown and Burton bug library. Second, we have applied DPF to empirical data consisting of three tests completed by school pupils. The data were collected at LRDC in Pittsburgh, and have been analyzed by VanLehn with the DEBUGGY program. In the following pages, we will compare the analyses generated by DPF with those produced by DEBUGGY.

Ideal Data

In testing the Diagnostic Path–Finder against idealized subtraction data, we proceed as follows:

1. We select a particular subtraction test used in the studies by VanLehn et al.
2. We select a bug from the Brown and Burton bug library.
3. We then compute the expected solution paths and answers for the problems in that test, under the assumption that the person solving them suffers from the selected bug.

4. We then feed the predicted answers to the DPF program and let it produce a path hypothesis for each problem.
5. Finally, we compare the expected solution paths with the ones produced by the DPF program.

In short, the idea is to "plant" some specific bug in the data, and then see whether the DPF program can "recover" that bug.

The reader is reminded that DPF treats each problem–answer pair as a unit, and thus derives a diagnosis for that pair independently of the diagnoses it has produced for other problem–answer pairs (even from the same individual). Thus, for a test with 17 answers, the program is in effect applied 17 times.

We selected a subtraction test with 17 problems which has been developed by Brown, Burton, and VanLehn for use in their research on subtraction errors. The test is reproduced in Table 4. The initial test was to run DPF on the correct answers for the 17 subtraction problems. It generated the correct solution path for each of those answers.† For example, the solution path for problem 3 (127 – 83 = 44) looked as follows:

(FirstColumn)
(FindDifference 7 3 column–1)
(ShiftLeft column–1)
(ShiftLeft column–2)
(Decrement 1 column–3)
(ShiftRight column–3)
(AddTen 2 column–2)
(FindDifference 12 8 column–2)
(ShiftLeft column–2)
(SkipZero 0 column–3)
(Answer: _ _ 4 4)

A few features of this path deserve comment. First, the program always starts out with its attention directed towards the right–most column. This is the meaning of the zero–argument operation "(FirstColumn)" at the beginning of the path. Second, each solution path ends with the pseudo–operator "Answer:" which simply notes that the answer is complete, with underscores representing positions that were left blank. Third, the columns are numbered from right to left, so that "column–1" is the right–most column.

The path for the more complicated problem 17 (10013 – 318 = 9695) looked as follows:

(FirstColumn)
(ShiftLeft column–1)
(Decrement 1 column–2)
(ShiftRight column–2)

† This was not overly surprising, since one of DPF's evaluation functions computes similarity to the correct solution path.

(AddTen 3 column–1)
(FindDifference 13 8 column–1)
(ShiftLeft column–1)
(ShiftLeft column–2)
(ShiftLeft column–3)
(ShiftLeft column–4)
(Decrement 1 column–5)
(ShiftRight column–5)
(AddTen 0 column–4)
(Decrement 10 column–4)
(Shiftright column–4)
(AddTen 0 column–3)
(Decrement 10 column–3)
(ShiftRight column–3)
(AddTen 0 column–2)
(FindDifference 10 1 column–2)
(ShiftLeft column–2)
(FindDifference 9 3 column–3)
(ShiftLeft column–3)
(FindTop 9 column–4)
(ShiftLeft column–4)
(SkipZero column–5)
(Answer: _ 9 6 9 5)

Looking at the path from problem 3, we see a typical sequence for a column in which no borrowing is needed:

...
(FindDifference 7 3 column–1)
(ShiftLeft column–1)
...

If no borrowing is needed, the column is processed by applying FindDifference, and shifting attention left to the adjacent column. At another place in the same path, we see a typical sequence of operators which deal with a column in which borrowing is necessary:

...
(ShiftLeft column–2)
(Decrement 1 column–3)
(ShiftRight column–3)
(AddTen 2 column–2)
(FindDifference 12 8 column–2)
...

In this case, the column is processed by shifting left one step, decrementing the top number in that column, shifting towards the right again to get back to the column being processed, adding 10 to this column, and then applying FindDifference. An even more complex solution path results for problem 17, which borrowing across several zeroes.

Our next test was to run DPF on incorrect answers for the same problems. VanLehn (1982) has published the frequences of the various bugs in the library of subtraction bugs. We selected a handful of the more frequent bugs, and computed for each bug the answers and solution paths that would result on the 17 test problems if they were solved with that bug. The DPF program was then run on the predicted incorrect answers. The results are presented in Table 3.

TABLE 3

DPF's Performance on a 17 Problem Subtraction Test.†

Planted error	Expected paths	Unexpected paths				N
		Alternative	Order	Weird	Incomplete	
Correct	17	0	0	0	0	17
SfromL	12	0	0	0	3	15
BnoD	10	2	1	0	2	15
0 – N = 0	5	2	0	0	0	7
0 – N = N	5	1	0	0	0	6
BonceSfromL	4	1	1	0	0	6
StopBat0	2	1	2	0	1	6
Bfrom0	2	0	0	1	3	6
Bacross0	0	0	2	1	1	4
Bacross0/–	0	0	1	1	1	3

The number of relevant answers in each row, the column marked "N", varies because the number of problems in the test for which a given bug predicts incorrect answers varies. None of the bugs predict incorrect answers to all problems in the test. Since we had already run it on the complete set of 17 correct answers, we did not present DPF with the correct answers again in these cases. The column marked "Expected" represents the number of cases in which DPF generated the solution path that we predicted on the basis of the bug we had "planted".

The three columns marked "Unexpected" contain those cases in which the program output path hypotheses, but not the ones we expected. They are divided into three

† Correct = No bugs, SfromL = Smaller-from-Larger, BnoD = Borrow-no-Decrement, BonceSfromL = Borrow-once-then-Smaller-from-Larger, StopBat0 = Stop-Borrow-at-zero, Bfrom0 = Borrow-from-zero, Bacross0 = Borrow-across-zero, Bacross0/- = Borrow-across-zero-over-blank.

categories. The first category, "Alternative" represents cases in which the path hypotheses output by DPF can be recognized as belonging to some other bug in the Brown and Burton bug library than the one we had planted. On the other hand, "Order" represents cases where the solution path produced by DPF contains exactly the expected steps, but in a garbled order. The third sub–category, "Weird", represents the cases in which the DPF program output a path hypotheses which bears no resemblance to any of the bugs in the Brown and Burton bug library.

Finally, the category marked "Incomplete" represents those cases in which the program could not find an hypothesis within the limits put on the best–first search. For these runs, a limit of 400 nodes was set. In other words, if the system could not find a path hypothesis by expanding at most 400 nodes of the problem space, it gave up (on that particular problem–answer pair).

Let us examine some of the diagnoses. Problem 4 (183 – 95) in Table 4 has the correct answer 88. The correct path is:

(FirstColumn)
(ShiftLeft column–1)
(Decrement 8 column–2)
(ShiftRight column–2)
(AddTen 3 column–1)
(FindDifference 13 5 column–1)
(ShiftLeft column–1)
(ShiftLeft column–2)
(Decrement 1 column–3)
(ShiftRight column–3)
(AddTen 7 column–2)
(FindDifference 17 9 column–2)
(ShiftLeft column–2)
(SkipZero 0 column–3)
(Answer: _ 8 8)

The Borrow–no–Decrement bug, shown in the third row of Table 3, consists in doing subtraction correctly, except that a column which is being borrowed from is not decremented. This bug predicts the incorrect answer 198 to problem 4. Given this answer, DPF outputs the following solution path:

(FirstColumn)
(AddTen 3 column–1)
(FindDifference 13 5 column–1)
(ShiftLeft column–1)
(AddTen 8 column–2)
(FindDifference 18 9 column–2)
(ShiftLeft column–2)

(FindTop 1 column–3)
(Answer: 1 9 8)

In contrast, the Borrow–once–then–Smaller–from–Larger bug, shown in the sixth row of Table 3, consists of only doing the first borrowing operation in each problem correctly. On the following columns in which borrowing is necessary, borrowing is replaced by subtracting the smaller number from the larger, regardless of which is above the other. This bug predicts the incorrect answer 128 to problem 4. Given this answer, DPF produces the diagnosis:

(FirstColumn)
(ShiftLeft column–1)
(Decrement 8 column–2)
(Shiftright column–2)
(AddTen 3 column–1)
(FindDifference 13 5 column–1)
(ShiftLeft column–1)
(FindDifference 9 7 column–2)
(ShiftLeft column–2)
(FindTop 1 column–3)
(Answer: 1 2 8)

Looking instead at problem 5 (106 – 38), the Borrow–No–Decrement bug predicts the incorrect answer 178, which prompts DPF to suggest the following solution path:

(FirstColumn)
(AddTen 6 column–1)
(FindDifference 16 8 column–1)
(ShiftLeft column–1)
(AddTen 0 column–2)
(FindDifference 10 3 column–2)
(ShiftLeft column–2)
(FindTop 1 column–3)
(Answer: 1 7 8)

In contrast, let us look at another bug for the same problem. The Borrow–from–Zero bug consists in changing a top zero to a nine while borrowing, without decrementing any column to the left of the zero. This bug predicts the incorrect answer 168 on problem 5. The DPF diagnosis for this answer is:

(FirstColumn)
(ShiftLeft column–1)
(AddTen 0 column–2)
(Decrement 10 column–2)
(ShiftRight column–2)

(AddTen 6 column–1)
(FindDifference 16 8 column–1)
(ShiftLeft column–1)
(FindDifference 9 3 column–2)
(ShiftLeft column–2)
(FindTop column–3)
(Answer: 1 6 8)

Note that the "changing of a zero into a 9" is accomplished by first adding ten and then decrementing.

Finally, let us look at a so–called "pattern error", the 0 – N = 0 bug. On problem 6 (800 – 168), this bug predicts the incorrect answer 700. The DPF diagnosis is:

(FirstColumn)
(FindTop 0 column–1)
(ShiftLeft column–1)
(FindTop 0 column–2)
(ShiftLeft column–2)
(FindDifference 8 1 column–3)
(Answer: 7 0 0)

Note that the pattern error is accomplished with the help of the FindTop operator, which is a necessary component of the correct subtraction skill.

In summary, the category "Expected" contains those cases in which the program output the expected solution path. As we can see from Table 3, this happened in 57 cases of 74 completed diagnoses.

Let us next consider the categories under the general heading "Unexpected". The first group, "Alternative" means that DPF output a solution path expected on the basis of some other bug in the bug library than the one we had "planted". This simply means that there is more than one bug in the bug library which predicts a particular incorrect answer, and that the evaluation procedure used by DPF implies that the path belonging to some other bug was to be preferred above the one we expected to see.

For instance, consider the problem 7 (513 – 268), with correct answer 245. The "planted" bug Borrow–no–Decrement predicts the incorrect answer 355, and the solution path:

(FirstColumn)
(AddTen 3 column–1)
(FindDifference 13 8 column–1)
(ShiftLeft column–1)
(AddTen 1 column–2)
(FindDifference 11 6 column–2)
(ShiftLeft column–2)

(FindDifference 5 2 column–1)
(Answer: 3 5 5)

However, the bug Smaller–from–Larger, which involves subtracting the top digit from the bottom digit in any column in which borrowing is necessary, also predicts the incorrect answer 355 to this problem. The solution path actually output by DPF for this problem–answer pair is:

(FirstColumn)
(FindDifference 8 3 column–1)
(ShiftLeft column–1)
(FindDifference 6 1 column–2)
(ShiftLeft column–2)
(FindDifference 5 2 column–3)
(Answer: 3 5 5)

In other words, the DPF system preferred the Smaller–from–Larger diagnosis on this problem–answer pair, in accordance with our intuition that this is a simpler account.

Looking at another example of an alternative diagnosis, consider problem 17, (10013 – 318), with correct answer 9695. The pattern bug 0 – N = 0 predicts the incorrect answer 10005. However, the solution path suggested by DPF for this answer is:

(FirstColumn)
(FindDifference 8 3 column–1)
(ShiftLeft column–1)
(FindDifference 1 1 column–2)
(ShiftLeft column–2)
(FindTop 0 column–3)
(ShiftLeft column–3)
(FindTop 0 column–4)
(ShiftLeft column–4)
(FindTop 1 column–5)
(Answer: 1 0 0 0 5)

As we see from this path, DPF has a different explanation for the digits in the two rightmost columns. The 5 in column–1 has come about, the program suggests, by using the Smaller–from–Larger strategy in that column. The 0 in column–2 then does not need any particular explanation. It follows by correct application of FindDifference, since there is now a 1 in both the top–row and the bottom–row of that column. However, the 0 in column–3 is derived with the 0 – N = 0 bug.

The interesting point of this example is that according to DPF, this hypothesis (which in DEBUGGY's terms postulates two different bugs) is simpler (according to DPF's evaluation functions) than an explanation based solely on the 0 – N = 0 bug. This comes about because DPF does not rely upon Brown, Burton, and VanLehn's notion of bugs; it

finds a solution path which accounts for the given answer, but it does not know how many distinct bugs a human interpreter will find in that path.

In summary, the "Alternatives" consist of cases in which the program produced acceptable hypotheses which just happened to be other diagnoses than the ones we thought we had "planted". As we can see from Table 3, this happened in 7 cases out of 74 completed diagnoses.

The next category of unexpected paths, the "Order" group, involves diagnoses which consist of exactly the same steps as the expected paths. However, the steps are arranged in unexpected order. (We are comparing the actual path output by the program with the expected buggy path, not with the correct path.)

For example, problem 6 (800 – 168) has the correct answer 632, and has the predicted incorrect answer 542 with the Borrow–across–Zero bug (row 9 in Table 3). This bug implies that columns with top zeroes are ignored during borrowing. For problem 6 this means that there should first be an incorrect borrow to deal with column–1: the 8 should be decremented, and ten added to column–1 (while ignoring column–2). Then a correct borrow should be made for column–2 (decrementing column–3 a second time).

However, DPF outputs the following path hypothesis in this case:

(FirstColumn)
(ShiftLeft column–1)
(ShiftLeft column–2)
(Decrement 8 column–3)
(ShiftRight column–3)
(AddTen 0 column–2)
(ShiftRight column–2)
(AddTen 0 column–1)
(FindDifference 10 8 column–1)
(ShiftLeft column–1)
(FindDifference 10 6 column–2)
(ShiftLeft column–2)
(Decrement 7 column–3)
(FindDifference 6 1 column–3)
(Answer: 5 4 2)

As we see, the program has postulated exactly the expected steps, including the double decrementing of column–3. However, the steps have been ordered so that the over–all character of the path is more similar to the correct path. The crucial column–2 is not ignored during the first borrow. It is incremented with 10, but not decremented. This gambit gives the expected value for column–2, but it leaves the final decrement of column–3 without any reasonable motivation. However, according to the evaluation procedure used by DPF, this is the best path hypothesis for this answer.

The problem of evaluating diagnoses now becomes acute: What should we think about

this "oddly–ordered" diagnosis? On the one hand, the path is a valid diagnosis in the sense that the path will, in fact, produce the given answer with the operators of the problem space. Thus, it is at least possible that a person working in that space actually proceeded in the way described by the path. On the other hand, our intuitions rebel against such an hypothesis: the unexpected path "feels" unnatural. However, as long as our intuitive judgement cannot be backed up by good arguments, it is not clear whether we should believe our intuitions or the program.

As a further example of a strangely–ordered solution path, consider the bug Borrow–across–zero–over–blank, which is a special case of the Borrow–across–zero bug, in which the error only occurs when the zero is over a blank in the borrow row. For instance, problem 17 (10013 – 318) has the correct answer 9695, while the Borrow–across–zero–over–blank bug predicts the answer 659. Given this answer, DPF outputs the path:

(FirstColumn)
(ShiftLeft column–1)
(Decrement 1 column–2)
(ShiftRight column–2)
(AddTen 3 column–1)
(FindDifference 13 8 column–1)
(ShiftLeft column–1)
(AddTen 0 column–2)
(FindDifference 10 1 column–2)
(ShiftLeft column–2)
(ShiftLeft column–3)
(SkipZero 0 column–4)
(ShiftLeft column–4)
(Decrement 1 column–5)
(SkipZero 0 column–5)
(ShiftRight column–5)
(ShiftRight column–4)
(AddTen 0 column–3)
(Decrement 10 column–3)
(FindDiffernce 9 3 column–3)
(Answer: _ 6 5 9)

Again, the hypothesis postulates all the right steps, but their order feels unnatural. As we can see from Table 3, 7 out of the 74 completed diagnoses consisted of the expected steps, but in an unexpected order.

The third group of unexpected paths we have labeled "weird" because they show no relationship to any recognizable procedural bug, and because they are difficult to interpret. There are only three such cases, which we discuss in turn.

First, consider problem 15 (608 – 209), which has the correct answer 399. For this

problem, the Borrow–from–Zero bug predicts the incorrect answer 499. However, when given this problem–answer pair, the DPF program produces the following solution path:

(FirstColumn)
(ShiftLeft column–1)
(ShiftLeft column–2)
(Decrement 6 column–3)
(Shiftright column–3)
(AddTen 0 column–2)
(Decrement 10 column–2)
(ShiftRight column–2)
(AddTen 8 column–1)
(FindDifference 18 9 column–1)
(ShiftLeft column–1)
(FindDifference 9 0 column–2)
(ShiftLeft column–2)
(Decrement 5 column–3)
(FindTop 4 column–3)
(Answer: 4 9 9)

In this path, everything is done correctly until the processing of column–3. In order to reconcile number 4 as the answer in this column with the previously correct steps, DPF invents the clever trick of arbitrarily decrementing the top digit (5) in column–3 one more time, and then simply copying the result as the column answer, equally arbitrarily ignoring the 2 in the bottom row of that column. This is a valid diagnosis in the sense that the path generates the given answer, but it does not make much sense.

The second example of a weird diagnosis concerns problem 14 (700 – 5), which has the correct answer 695. For this problem, the Borrow–across–Zero bug predicts the incorrect answer 605. However, when given this answer, DPF generates the path:

(FirstColumn)
(FindDifference 5 0 column–1)
(ShiftLeft column–1)
(FindTop 0 column–2)
(ShiftLeft column–2)
(Decrement 7 column–3)
(FindTop 6 column–3)
(Answer: 6 0 5)

The pattern here is similar, although the path is not correct even initially. DPF begins by postulating the Smaller–from–Larger bug to account for the answer in column–1: the smaller number is simply subtracted from the larger number, although the smaller is above the larger. This gives the right digit in column–1, but leaves the problem of how to account

for the rest of the digits unsolved. When it gets to column–3, DPF again resorts to the trick of performing an arbitrary decrement to get the desired value. The special case of this bug, Borrow–across–zero–over–blank, predicts the same incorrect answer for this problem, so DPF output the same weird path.

In summary, DPF's performance on the ideal data was mixed. Out of 74 completed diagnoses (one diagnosis for each problem–answer pair), some 57 were as expected. Another 7 consisted in a different diagnosis than the one expected, and 7 more included the expected steps in an unexpected order. A strict evaluation of the program would count only the 57 expected diagnoses as correct, in which case the system performed correctly on 77% of the cases. A more lenient evaluation would count all 71 of the above paths as correct, in which case the program performed correctly 96% of the time.

In addition, there were 12 cases in which the program could not complete the diagnosis within the limits set for exploring the problem space, and this is a serious matter. They indicate that the program must be made considerably more selective, since the current evaluation criteria are simply not sufficient. Of course, there is no way of knowing which way these incomplete diagnoses would have gone had they been allowed to run to completion.

Empirical data

We have the data from an empirical study of several school children taking the test in Table 4, as well as other subtraction tests. We also have the diagnoses by DEBUGGY for these children. The diagnoses have been produced by Kurt VanLehn. We thank Susan Omansson at the Learning Research and Development Center at the University of Pittsburgh for making the material available to us.

We have selected three children, and run DPF on their answers. The first child, here called "Tommy" was diagnosed as "mostly unsystematic" by DEBUGGY, while "James" is diagnosed as "systematic" with the two bugs Borrow–across–Zero and Borrow–dont–decrement–top–smaller. The third child, "Judy" was diagnosed as "highly systematic" with the bug Stops–Borrow–at–Zero.

Diagnosis of Tommy

Tommy solved all 17 problems in the test in Table 4. He solved 12 of them correctly, delivering only 5 incorrect answers. DEBUGGY considered 19 different diagnosis for this subject. The best diagnosis turned out to be Borrow–across–Zero in combination with Treat–Top–Zero–as–Ten, plus the local number–fact error $8 - 1 = 5$, applied to problem 6. This diagnosis accounts for 4 of the five incorrect answers, while at the same time predicting correct answers to 7 problems that Tommy also solved correctly. However, this diagnosis fails to account for one of Tommy's incorrect answers, and predicts incorrect answers to five problems which Tommy solved correctly. The other 18 diagnoses considered by DEBUGGY for this child account for even less of the behavior.

We ran DPF on the five problems which Tommy solved incorrectly. It was unnecessary to run the program on the correct answers, since we already knew that DPF would output

the correct solution paths for them. DPF generated solution paths for four of the five problems.

On problem 5 (106 – 38), Tommy answered "72". None of the 19 diagnoses considered by DEBUGGY can explain this answer. The solution path suggested by DPF is the following:

(FirstColumn)
(ShiftLeft column–1)
(ShiftLeft column–2)
(Decrement 1 column–3)
(ShiftRight column–3)
(AddTen 0 column–2)
(FindDifference 10 3 column–2)
(ShiftLeft column–2)
(SkipZero 0 column–3)
(ShiftRight column–3)
(ShiftRight column–2)
(FindDifference 8 6 column–1)
(Answer: _ 7 2)

Thus, DPF postulates that Tommy did Smaller–from–Larger in column–1, but solved the rest of the problem correctly. The steps seem at first to be arranged in a completely arbitrary order.

However, closer inspection reveals an interpretation of the solution path. The steps and the order in which they appear are consistent with the hypothesis that Tommy began to make a correct borrow from column–3 into column–1. However, at column–2 he forgot which column was the "current column" (i.e., the column he was working on). Thus, he processed column–2 and then column–3 correctly. He then discovered that the answer "looked funny", because of the missing digit in column–1. He then returned to column–1 and treated it in isolation from the rest of the problem, in the only way possible. Of course, the value of such speculation in a practical context is questionable. However, here it serves points out that an interpretable explanation for the mysterious "72" answer is possible, and that DPF can generate it.

The second buggy answer by Tommy is "542" as the answer to problem 6 (800 – 168). This answer is predicted by four of the 19 diagnoses considered by DEBUGGY. DPF outputs the path:

(FirstColumn)
(ShiftLeft column–1)
(ShiftLeft column–2)
(Decrement 8 column–3)
(ShiftRight column–3)
(AddTen 0 column–2)

(ShiftRight column–2)
(AddTen 0 column–1)
(FindDifference 10 8 column–1)
(ShiftLeft column–1)
(FindDifference 10 6 column–2)
(ShiftLeft column–2)
(Decrement 7 1 column–3)
(FindDifference 6 1 column–3)
(Answer: 5 4 2)

This path belongs in the category of "weird" diagnoses. Part of the path is systematic enough: Tommy forgets to decrement the 10 which is added to column–2. However, DPF also infers that he applied an extra decrement to column–3, to make the left–most digit come out right. The path as a whole does not seem to yield any systematic understanding of what Tommy did.

However, his behavior on the next problem does yield to a systematic interpretation. Tommy gave "2087" as the answer to problem 12 (3005 – 28). This answer can be explained only by two of the 19 diagnoses considered by DEBUGGY, in both cases by combinations of two bugs. In contrast, DPF gives the path:

(FirstColumn)
(AddTen 5 column–1)
(FindDifference 15 8 column–1)
(ShiftLeft column–1)
(AddTen 0 column–2)
(FindDifference 10 2 column–2)
(ShiftLeft column–2)
(FindTop 0 column–3)
(ShiftLeft column–3)
(Decrement 3 column–4)
(FindTop 2 column–4)
(Answer: 2 0 8 7)

This path suggests a new subtraction procedure, which is not listed in the Brown and Burton bug library: In each column where borrowing is necessary, add ten to the top digit, and then subtract; when you reach the left–most column, decrement the top digit before subtracting. Thus, the columns are processed from right to left, and at the end all the "loans" are "paid back" in a single decrement. In the style of Brown and Burton's way of naming bugs, this might be called Add–Ten–Freely–Decrement–Last.

This bug also predicts the solution path constructed by DPF for the next buggy answer, "309", to the problem (608 – 209):

(FirstColumn)

(AddTen 8 column–1)
(FindDifference 18 9 column–1)
(ShiftLeft column–1)
(FindDifference 0 0 column–2)
(ShiftLeft column–2)
(Decrement 6 column–3)
(FindDifference 5 2 column–3)
(Answer: 3 0 9)

Unfortunately, DPF left the last diagnosis uncompleted. However, the Add–Ten–Freely–Decrement–Last bug does not predict the fifth and final incorrect answer by Tommy. On the whole, that bug does not make a better over–all diagnosis of Tommy than any of the other diagnoses considered by DEBUGGY.

However, if we look more abstractly at the four different solution paths suggested by DPF, two regularities emerge: First, Tommy never decremented a 10 which he added during a borrow. Second, he always decrements the top digit in the left–most column. These regularities may be useful, even if they are not sufficient to specify a particular procedure.

In summary, DPF's analysis mainly agrees with the DEBUGGY verdict that Tommy's subtraction behavior is unsystematic. However, the bottom–up nature of DPF allows it to find a novel explanation for the "72" answer. Moreover, two of DPF's proposed paths suggest a systematic subtraction bug not mentioned in the Brown and Burton bug library. Finally, the set of paths as a whole suggests two weak regularities in Tommy's behavior.

Diagnosis of James

James delivered answers to all 17 problems in the test. Eight of these were correct, and nine incorrect. DEBUGGY considered twelve different diagnoses for this child, and classified him as "systematic". The best diagnosis accounted for 6 of his correct answers, and 8 of his incorrect ones, a success rate of 14 out of 17. The diagnosis is a complex one, consisting of two main bugs supplemented with two assumptions of local pattern errors on problems 13 and 16. The two main bugs are Borrow–across–Zero and Borrow–dont–Decrement–Top–Smaller. These two bugs account for 12 of the observed answers. In order to make 14 hits, this hypothesis was supplemented with the assumptions that the child made the local number–fact error $8 - 2 = 7$ on problem 13, and the error $10 - 2 = 9$ on problem 16.

Before considering what DPF has to say about this child, let us pit the bug that we discovered while diagnosing Tommy against the best DEBUGGY diagnosis. It turns out that Add–Ten–Freely–Decrement–Last predicts five of James' incorrect answers, and five of his correct ones. It would rank as second best among the 12 diagnoses considered by DEBUGGY.

Can we find some supplementary hypothesis which helps Add–Ten–Freely–Decrement–Last to account for Tommy's behavior? Inspection of his answers suggest the pattern bug $1 - 1 = 9$, restricted to the tens column. This bug implies that a one over a one is treated

as a ten. Especially in the tens column, this does not seem like an unreasonable bug.

If we extend our account by assuming this new pattern bug, which we might call Treat–One–Over–One–As–Ten, then our hypothesis predicts two additional answers, one correct and one incorrect, to make a total of 12 answers correctly accounted for. In short, the bug combination Add–Ten–Freely–Decrement–Last plus Treat–One–over–One–As–Ten competes favorably with the best DEBUGGY diagnosis for this particular child. How useful this hypothesis would be on a large sample of children we do not know.

Let us now have a look at the paths actually output for James by the DPF system. For 8 out of James' 9 incorrect answers, DPF generated complete path hypotheses, but recall that the program treats each problem–answer pair separately. Thus, it tries to find the best path hypothesis to account for each particular answer, making it quite insensitive to the implications of that hypothesis for the over–all picture of James.

Because of this piecemeal and bottom–up strategy, no consistent picture of James emerges from the eight solution paths. Two of these follow perfectly the Add–Ten–Freely–Decrement–Last bug. The answer 605 to problem 14 (700 – 5) is, on the other hand, accounted for with a perfect example of Borrow–across–Zero:

(FirstColumn)
(ShiftLeft column–1)
(ShiftLeft column–2)
(Decrement 7 column–3)
(ShiftRight column–3)
(ShiftRight column–2)
(AddTen 0 column–1)
(FindDifference 10 5 column–1)
(ShiftLeft column–1)
(FindTop 0 column–2)
(ShiftLeft column–2)
(FindTop 6 column–3)
(Answer: 6 0 5)

However, the remaining five path hypothesis from James are more or less weird. One is based on Smaller–from–Larger, followed by a decrement of the top–digit in the last column. Three paths include the psychologically implausible trick of decrementing the top digit in a column until it has the expected value, and then copying it as the answer for that column (ignoring the number in the bottom–row in that column). (These three include the two answers which DEBUGGY could not explain, except by postulating local pattern errors.) The last one is a different variant of the "too many decrements" theme. In short, DPF's path hypotheses for James do not reflect the regularities that we know exist in his answers.

Diagnosis of Judy

Judy solved all 17 problems, and her performance is easy to describe: All her answers, 6 incorrect and 11 correct, are in accord with the Borrow–across–Zero bug. DEBUGGY

classified her as "highly systematic".

We ran DPF on the 6 incorrect answers, and the system completed 5 diagnoses. Of these, four are consistent or nearly consistent with the Borrow–across–Zero bug; two have the correct order, and two have the steps in slightly garbled order.

However, on problem 14 (700 – 5), DPF suggests that the Smaller–from–Larger is a better diagnosis:

(FirstColumn)
(FindDifference 5 0 column–1)
(ShiftLeft column–1)
(FindTop 0 column–2)
(Shiftleft column–2)
(FindTop 7 column–3)
(Answer: 7 0 5)

Thus, in this case, the two systems essentially agree on the diagnosis of the child, though again DPF's blindness to the overall picture kept it from generating completely systematic paths.

Discussion

To summarize, we have presented an approach to automated cognitive diagnosis which builds on the problem space hypothesis. This hypothesis implies that cognitive diagnosis consists of three distinct steps: identifying the problem space, finding the relevant path in that space, and inventing a procedure which can re–create that path. The first problem is still far from being automated. We have explored the second problem through the Diagnostic Path–Finder (DPF), a computer program that constructs a solution path for an observed answer. Approaches to the third problem have been examined by AI machine learning researchers, and in earlier papers (Langley, Ohlsson, and Sage, 1984; Langley and Ohlsson, 1984) we have shown how one such approach can be used in cognitive diagnosis.

DPF inputs a problem space, a set of problems, a correct procedure for those problems, and a set of (typically incorrect) answers to those problem. It delivers as output a path hypothesis – some sequence of steps (operator applications) which lead from the problem to the given answer. In generating this path, the program carries out a best–first search through the problem space, using an evaluation function to guide its search. This evaluation function estimates the psychological plausibility of alternative paths, rather than their likelihood of leading to a solution. The evaluation metric is composed of a set of criteria which together constitute the system's definition a "good psychological hypothesis".

We applied the program to both ideal and empirical data in the domain of multi–column subtraction. In the ideal data test, we planted selected bugs from the Brown and Burton bug library for subtraction, and then tested the program to see whether it could recover the planted bugs. The program completed 88% of the diagnoses. Of these completed diagnoses, 77% were the expected ones, 19% were unexpected (but still

acceptable) diagnoses, while 4% of the diagnoses were weird or uninterpretable.

In the empirical data test, we applied the system to the answers given by three different school children, classified by the DEBUGGY system as "mostly unsystematic", "systematic", and "highly systematic", respectively. DPF managed to reveal a couple of weak regularities in the performance of the supposedly unsystematic child. However, with respect to the systematic child, DPF failed to emphasize the known regularities, producing a diverse set of diagnoses, including five weird solution paths. Finally, DPF essentially agrees with DEBUGGY with respect to the highly systematic child.

The most interesting achievement of DPF on this set of runs was that it suggested a new subtraction bug. Several solution paths proposed by the system can be summarized by the following buggy procedure: Process the columns from right to left; for each column in which borrowing is necessary, simply add ten to the top digit before subtracting in that column; when you reach the left–most column, decrement the top digit once before subtracting in that column. We suggest the name Add–Ten–Freely–Decrement–Last for this bug. This bug does a fairly good job of accounting for the "systematic" child, which DEBUGGY diagnosed as suffering from a combination of Borrow–Across–Zero and Borrow–Dont–Decrement–Top–Smaller. Thus, it may be a worthwhile addition to the Brown and Burton bug library for subtraction.

Among the strong points of DPF, we would like to emphasize the following:

- DPF is able to construct solution paths for some incorrect answers which DEBUGGY cannot account for at all. There are incorrect answers in the performance of the three children which are not accounted for by any of the bugs or bug combinations considered by DEBUGGY. The solution paths output by DPF in those cases are frequently weird and unnatural. However, the extent to which those paths have psychological validity is an empirical question.
- DPF does not postulate unanalyzed errors. For instance, so–called pattern bugs like 0 – N = 0 are simulated with the help of the FindTop operator. The point is that the FindTop operator is a necessary component of subtraction skills, since some problems have many more digits in the top number than in the bottom number. As a second instance, the operation of "changing a zero to a 9" (which is at the heart of the Borrow–from–Zero bug) is represented as an application of AddTen, followed by an application of Decrement. Thus, there are no black boxes which simply output wrong results. Every incorrect digit is derived through a sequence of steps, each of which represents the correct application of some operator.
- Since DPF produces solution paths, it is possible to find regularities in a performance even when it is too unsystematic to be simulated. Even if a child keeps changing his subtraction procedure in an erratic way, it may be that there are still some properties of his performance which remain stable under those changes. Such a performance cannot be simulated by any subtraction procedure, but given the solution paths, the regularities may still be recovered. For instance, in diagnosing the child classified as "unsystematic", DPF's solution paths suggested the regularity that the tens which are added during borrowing are never decremented. For practical purposes, such regular-

ities may well be useful.

Among the negative points of the DPF program, the following two dominate:

- Too many of the diagnoses were left incomplete. DPF could not find any solution path within the limits set on the search in 12% of the cases. This suggests that the evaluation procedure guiding the best–first search is not selective enough, and should be improved.
- The pure bottom–up approach taken in DPF is clearly unreasonable. The solution path for each answer is computed independently of any other answer by the same problem solver. In the case of the "systematic" child, this approach led to a blindness for the over–all regularities in that subject's performance. The program should be augmented with the capacity to attend to the over–all consistency of its diagnosis of a particular subject.

There are several issues which need to be discussed further. They include questions concerning the existence of competing path hypotheses, problems of generality, and alternative goals in constructing diagnostic systems.

Competing hypotheses

The work on errors in subtraction exemplifies a general rule – for any given (incorrect) answer to a task, there exists more than one (buggy) procedure that can generate that answer. This is only one instance of an even more general rule – that most facts can be explained by more than one theory. How should a system for automated cognitive diagnosis respond to this possibility of competing hypotheses?

The traditional scientific stance regarding competing hypotheses is that one should use the hypotheses to make differential predictions, and then collect more observations. The hypothesis which most successfully accounts for the new data should be preferred.

This approach might be plausible, if people were completely stable in their thought habits over long periods of time. We could then derive the expected performance on a new set of problems, reobserve the subject, and check the predictions. However, the experience so far suggests that people change their strategies frequently. The fact that subject S applied procedure P to task T at time t is not a Law of Nature, nor is it a Law of Psychology. Thus, the method of designing critical experiments is not necessarily meaningful in this field, though it may have limited usefulness.

However, one could imagine diagnostic systems which are not confined to answer–data alone. Automated diagnostic methods have been successful in some task domains using only answer–data, but this does not mean that other types of data could not be useful as well. In fact, DPF and similar systems would benefit greatly from process–oriented data. For instance, best–first search through the problem space could be guided by a rough outline of the student's path, derived from some form of process–oriented data.

However, this approach introduces problems of its own. First, developing task independent methods for the treatment of process–oriented data is an AI research problem in

its own right. The second drawback is a practical one; process–oriented data generally requires extensive recording equipment, the ability to observe one subject at a time, ensuring that subjects are not disturbed by the recording equipment, and so forth. These conditions are usually fulfilled in a research laboratory, but seldom in real–world contexts.

Another response to the problem of competing hypotheses is to make the diagnostic system interactive. When the program cannot choose between two or more competing hypotheses, it can explain its choice to the user, who then indicates which alternative should be used. The success of such a scheme depends on the intended user; e.g., it may require a level of sophistication beyond the range of most grade school teachers.

Even should such a solution work well in practice, it would not be very intellectually satisfying. It implies that the diagnosis will still be only as good as the intuitions of the person interacting with the system. Also, it means that the actual problem of determining a good hypothesis has been circumvented, rather than solved.

In the end, the only honest approach to this problem is to supply the program with some procedure by which it can choose between competing hypotheses. Of course, in DPF, such a procedure is necessary anyway because the hypotheses are constructed though selective search. Thus, the problem of selecting between competing hypotheses and the problem of making the program more efficient are one and the same: how can we make the evaluation procedure better?

Our difficulties in constructing the evaluation procedure for the DPF system has brought home to us the poverty of current psychological theorizing. The number of theoretical principles that can be turned into useful evaluation criteria is very small. Most psychological theorizing seems to be have no implications for how to evaluate psychological hypotheses!

Considering our intuitions about why we disbelieve the "monster diagnoses" that our program has produced, the two concepts of justification and genesis come to mind. The path hypotheses which we find natural and psychologically plausible are those in which we can imagine either how the child justifies the steps to himself, or how the bug came about. If we cannot imagine either of these things, then we experience the explanation as weird. Future improvements of the evaluation procedure should probably take its inspiration from these intuitions. For example, one way to proceed would to be formalize the justification (i.e., the semantics) of the correct procedure, and then search for diagnoses which violate that justification as little as possible.

Units of Analysis

The most glaring difference between DPF and the DEBUGGY system lies in the diagnostic unit employed by the two systems. DEBUGGY makes a diagnosis of a subject, considering all answers produced by that subject. DPF makes a diagnosis of an answer, ignoring other answers from the same subject.

Since we want to understand a subject, it might seem as if the subject is the most natural unit of diagnosis. This would be so, provided subjects were completely consistent.

If we could trust that a subject executed exactly the same procedure on all the problems in a test, then it would be very inefficient indeed not to take all solution paths into account while doing diagnosis.

However, the situation is otherwise. Subjects/students (a) change their procedures in "midflight", replacing one buggy procedure with another in the midst of solving a problem, (b) learn while doing, and (c) make a fair number of random mistakes. This means that a diagnostic system cannot assume that it will be able to find a single procedure which will account for all the answers by a particular subject.

DEBUGGY's response to this situation is to consider a number of diagnoses for each subject, apply each to the entire range of answers, and tabulate the hits and the misses. The diagnosis with the most hits is regarded as the best. If there is no diagnosis which accounts for a significant proportion of the performance, the subject is classified as "unsystematic".

In the bottom–up approach of the DPF system, there is no assumption of inter–problem consistency. This gives us a handle on inconsistent performances. For example, if a subject solves 10 problems with bug A, and then another 10 problems with bug B, DPF should produce ten paths consistent with A followed by ten paths consistent with B.

However, the runs made with DPF indicates that doing diagnosis on each problem–answer pair separately may not be a reasonable ambition. Unless the paths are compared with each other, larger regularities may be ignored completely.

The problem, then, is to develop a method for using inter–problem consistency to facilitate diagnosis, while at the same time avoiding unwarranted assumptions about systematicity on the part of the subjects. Currently, we have no handle on this problem.

Generality

Our aim is to develop a domain–independent system for automated cognitive diagnosis. The usefulness of such a system for both research and practical work should be obvious. This makes the diagnostic problem very hard. If we had allowed ourselves to use subtraction–specific evaluation criteria in DPF, it would have performed considerably better. Indeed, the generality goal makes the problem so hard that one might reasonably ask whether this is not an overly ambitious goal. There are several aspects to this issue.

First, let us point out that the machine learning methods that we plan to use for the third diagnostic step – generating a procedure from a set of paths – are domain–independent. Generalization, discrimination, and related mechanisms have already been cast in content–independent form in functioning AI programs and cognitive simulations of human behavior. Thus, one part of the problem is already solved at a high level of generality.

Second, we have already discussed the properties of the task which might facilitate diagnosis of the kind that DPF represents. Diagnosis on the basis of answer–data only is likely to be possible in task domains in which (a) the goal–object is not given in the statement of the the problem, (b) the goal–object is chosen from a very large set of objects, and (c) the goal–object has internal structure, the richer the better.

Presumably, the ambition to construct a very general diagnostic methodology implies that we must give some thought to what we should do in domains which do not share these properties. The obvious answer is to provide in the system for the use of process–oriented data as well.

However, the automatic treatment of process–oriented data is a difficult matter. Some efforts have been spent on constructing a system for automatic analysis of think–aloud protocols (Waterman and Newell, 1972), but without any great success. The problem lies not in making use of the data, but in designing a procedure that is domain–independent. One would have to formally define the various forms of process–oriented data that one might collect in a psychological investigation (think–aloud protocols, eye–movements, motor actions). For each type of data, one would have to specify general procedures for treating each such data–type. We are not currently pursuing this line of research.

However, the generality of an AI system is always a matter of degree. Obviously, some description of the task must always be given to any system. The relevant questions are rather what kind of task–specific input a system requires, how difficult it is to prepare, and how much of the system's intelligence is actually hidden in that hand–crafted input. Even if our ambition is to construct a general methodology, we do not believe it is possible without some task–specific input. The question, then, is what kind of input.

Comparing the DEBUGGY and DPF systems on this question, we see that differ markedly on this point. Presumably, the main machinery of the DEBUGGY system could be used in any task domain for which a bug library is available. Thus, it is the bug library that carries the task–specific information. In the DPF system, the only task–specific part is the problem space. How can the problem space replace the bug library, and what has been gained thereby?

First, let us recapitulate a previous argument. Suppose that we have a problem space and develop it exhaustively. One or more terminal nodes will then contain the "correct" answer to the problem. All other terminal nodes will represent some "incorrect" answer to the problem. Obviously, the only incorrect answers which can be accounted for in that space are those which are represented by some terminal node. In other words, a problem space implicitly defines a bug library. The reason why DPF can make diagnoses without an explicit bug library is that is has an implicit bug library instead.

Has anything been gained by going from an explicit to an implicit library? We want to argue that several important things have been gained.

First, the problem space has several functions in the system. It is not just an implicit bug library. It also constitutes an hypothesis about the subjects' representation of the task. The problem space hypothesis is not an arbitrary construct, independent of psychological theory. By adopting this hypothesis, we link our diagnostic methodology to a theory of problem solving, and thereby to research inspired and guided by that theory. Hopefully, this means that advances in the theory can be easily related to the diagnostic task.

Second, notice that according to the problem space hypothesis, we always need to specify the problem space the subject is working in, whether we are doing diagnosis or something else. Thus, the problem space is not an extra addition to our arsenal required

by the diagnostic task, but something we must have anyway in order to produce a cognitive simulation.

Third, and more importantly, perhaps, is that a problem space is a specification or characterization of the bug library that is implicit in it. The standard subtraction space used in DPF's runs represents all bugs consistent with the assumption that the problem solver knows the operators of that space. Thus, we can discuss the differences between bug libraries in terms of the differences between the operators. When we run the DPF program we know that the program, by construction, "considers" all the bugs in the implicitly defined bug library.

Fourth, and most important of all, a problem space, being an hypothesis, can be researched and evaluated. One can gather information relevant to the notation used in the problem space by interviewing subjects about how they view the task, recording their eye–movements in order to observe what aspects of the task they attend to, and so on. One can collect information about which operators one can plausibly postulate in the problem space by observing which sub–skills subjects can execute without difficulty. Thus, a problem space can be backed by empirical evidence independent of the buggy answers for which it accounts.

In short, we feel that there are considerable advantages to using problem spaces as the carrier of the task–specific information in a diagnostic system. The approach makes a clear separation between task–specific and general knowledge: the problem space represents the task domain; the generality of the system lies in the task–independent method used to search the space, and in the evaluation procedure used to direct that search.

TABLE 4

Subtraction Test Used in Evaluating DPF.

N	Problem	Correct answer
1	43 – 7	36
2	80 – 24	56
3	127 – 83	44
4	183 – 95	88
5	106 – 38	68
6	800 – 168	632
7	513 – 268	245
8	411 – 215	196
9	654 – 204	450
10	5391 – 2697	2694
11	2487 – 5	2482
12	3005 – 28	2977
13	854 – 247	607
14	700 – 5	695
15	608 – 209	399
16	3014 – 206	2808
17	10013 – 318	9695

References

Anderson, J. R. *The Architecture of Cognition.* Cambridge, Mass.: Harvard University Press, 1983.

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

Anderson, J. R. and Bower, G. H. *Human Associative Memory.* Washington, D.C.: Winston, 1973.

Anzai, Y. and Simon, H. A. The theory of learning by doing. *Psychological Review*, 1979, *86*, 124–140.

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

Brown J.S. and VanLehn K. Repair theory: a generative theory of bugs in procedural skill. *Cognitive Science*, 1980, *4*, 379–427.

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

Ericsson, K. A. and Simon, H. A. *Protocol Analysis: Verbal Reports as Data.* Cambridge, Massachusetts: MIT Press, 1984.

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

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

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

Langley, P. and Ohlsson, S. Automated cognitive modeling. In *Proceedings of the National Conference on Artificial Intelligence*, 1984, 193-197.

Langley, P., Ohlsson, S., and Sage, S. A machine learning approach to student modeling. Technical Report CMU–RI–TR–84–7, The Robotics Institute, Carnegie–Mellon University, 1984.

Lewis, C. Composition of productions. To appear in D. Klahr, P. Langley, and R. Neches (Eds.), *Production System Models of Learning and Development.* Cambridge, Massachusetts: MIT Press, 1985.

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

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

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

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

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

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

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

Rosenbloom, P. The Chunking Model of Goal Hierarchies: A Model of Practice and Stimulus–Response Compatibility. Dissertation, Department of Computer Science, Carnegie–Mellon University, 1983.

Sleeman D. Inferring student model for intelligent computer–aided instruction. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, Ca: Tioga Press, 1983.

Smedslund, J. Psychological diagnostics. *Psychological Bulletin*, 1969, *71*, 237–248.

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

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

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

Waterman, D. A. and Newell, A. Preliminary results with a system for automatic protocol analysis. CIP Report No. 211, Department of Psychology, Carnegie–Mellon University, Pittsburgh, Pennsylvania, 1972.

Williams, M. D. and Hollan, J. D. The process of retrieval from very long–term memory. *Cognitive Science*, 1981, *5*, 87–119.

Young, R. M. *Seriation by Children: An Artificial Intelligence Analysis of a Piagetian Task.* Basel, Switzerland: Birkhauser, 1976.

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