In M.Polson & J.Richardson (Eds.)
Foundations of Intelligent Tutoring
Systems. Hillsdale, NJ: Erlbaum.
1988. pgs. 55-78.

3
Student Modeling

Kurt VanLehn
Carnegie-Mellon University

This chapter reviews the research literature concerned with the student modeling component of intelligent tutoring systems. An intelligent tutoring system, or ITS, is a computer program that instructs the student in an intelligent way. There is no accepted definition of what it means to teach intelligently. However, a characteristic shared by many ITSs is that they infer a model of the student's current understanding of the subject matter and use this individualized model to adapt the instruction to the student's needs. The component of an ITS that represents the student's current state of knowledge is called the *student model*. Inferring a student model is called *diagnosis* because it is much like the medical task of inferring a hidden physiological state (i.e., a disease) from observable signs (i.e., symptoms). An ITS diagnostic system uncovers a hidden cognitive state (the student's knowledge of the subject matter) from observable behavior.

The student model and the diagnostic module are tightly interwoven. The student model is a data structure, and diagnosis is a process that manipulates it. The two components must be designed together. This design problem is called the *student modeling problem*. This chapter reviews solutions that have been found to the student modeling problem and discusses the techniques that have been discovered.

THE STUDENT MODELING PROBLEM

Most design problems in computer science can be specified by describing the desired output of the program and the available input. The design problem here is not, unfortunately, so neatly circumscribed.

Generally speaking, the input for diagnosis is garnered through interaction with the student. The particular kinds of information available to the diagnosis module depend on the overall ITS application. The information could be answers to questions posed by the ITS, moves taken in a game, or commands issued to an editor. In some applications, the student's educational history is also available to the diagnostic component.

The output from the diagnostic module is even harder to circumscribe. In fact, it doesn't even make sense to talk about the product of diagnosis as "output" (here, the analogy to medical diagnosis breaks down). Rather, the result is a data base, the student model, which accurately reflects the student's knowledge state. The student model is drawn on by other ITS modules for many purposes. Following are listed some of the most common uses for the student model.

Advancement. Some ITSs use a structured curriculum. A student is moved to the next topic in the curriculum only when he or she has mastered the current topic. In such applications, the student model represents the student's level of mastery. Periodically, the ITS asks the student model for the level of mastery on the current topic, weighs it, and decides whether to advance the student to the next topic. This use of student models is called *advancement.* Advancement is useful not only with linearly structured curricula, where instruction dwells on one topic at a time, but also in componentially structured curricula, where a student exercises several topics or skills at the same time. For instance, in the WUSOR ITS (Goldstein, 1982), the student uses several reasoning skills at the same time to hunt a beast in a maze filled with dangerous pits and bats. The techniques for estimating the dangerousness of caves can vary independently of the techniques for determining what caves are likely to contain the beast. The ITS can advance a student through the skill levels for assessing danger independently of advancing the student through the skill of locating quarry. This illustrates how advancement is used in ITSs that do not use a linearly structured curriculum.

Offering unsolicited advice. Some ITSs are like athletic coaches in that they offer advice only when they see that the student needs it.

If the student is performing well, the coach remains silent. A good coach will also remain silent if the student makes a mistake in a situation that is too complicated for a successful pedagogical interaction to take place. In order to offer unsolicited advice at just the right moments, the ITS must know the state of the student's knowledge. For this, it reads the student model.

Problem generation. Some ITSs generate problems for the student dynamically rather than sequencing through a predefined list of problems or letting the student invent problems to solve. In many applications, a good problem is just a little beyond the student's current capabilities. To find out where the student's current capabilities lie, the problem generation module consults the student model.

Adapting explanations. When good tutors explain something to the student, they use only concepts that the student already understands. For an ITS to issue good explanations, it must determine what the student knows already. To do so, it consults the student model.

The preceding functions are some of the most common ways the ITS components use the student model. Because there are so many ways to use the student model, we cannot talk sensibly about the output of the diagnosis module, nor can we classify student modeling problems by the desired input–output relationship. What does make sense is to classify these problems according to the structural properties of the student model. For instance, the student model might represent various levels of mastery of a subskill by a single bit (mastered versus not yet mastered), by a number, or by a complicated qualitative description. Such structural properties of the student model determine how complicated the student modeling problem is and what kinds of techniques are best suited for its solution.

A THREE-DIMENSIONAL SPACE OF STUDENT MODELS

This section reviews existing student modeling systems in the context of a classification based partially on structural properties of the student model and partially on properties of the input available to the diagnosis module. At this writing, approximately 20 student modeling systems have been built, and more are under development. There are many differences among them. The classification presented here is intended to capture the differences in the student modeling problem that really make a difference in the solution techniques. If this classification is correct, it can be used to predict what kinds of student modeling

techniques would be most useful for some new student modeling problem. Needless to say, such a prediction would be only the starting point in a long design process that results in a system adapted to the demands of a particular ITS. Indeed, as more ITSs are constructed, the perception of what differences really matter can be expected to change. That change is one reason why ITS construction is still in the research stage and has not yet become a mature technology. In short, the following classification is both heuristic and tentative.

The classification has three dimensions. The first one relates to the input, and the others are structural properties of the student model.

Bandwidth

The input to the diagnosis unit consists of various kinds of information about what the student is doing or saying. From this, the diagnosis unit must infer what the student is thinking and believing. Clearly, the less information the unit has, the harder its task is. The *bandwidth* dimension is a rough categorization of the amount and quality of the input.

Three levels of information suffice to capture most of the variation among existing ITSs. In order to explain them, we will assume that students are solving problems posed either by themselves (e.g., What cave shall I explore next?) or by the ITS (e.g., What is 283-119?) If the problem solving takes more than a few milliseconds, then we can safely assume that the students go through a series of mental states. The highest bandwidth an ITS could attain would be a list of the mental states that the students traverse as they solve problems. Human mental states are not directly accessable by machines, so no ITS can really achieve this "mental states" bandwidth. However, by asking enough questions or by eliciting verbal protocols, an ITS can obtain indirect information that approximates the students' mental states. So the highest bandwidth category is *approximate mental states.*

In more complicated forms of problem solving, such as solving algebraic equations or playing chess, the students make observable changes that carry the problem from its initial unsolved state to its final, solved state. This results in a series of observable intermediate states, such as the midgame board positions in chess or the equations written before the last equation during algebraic equation solving. Sometimes an ITS has access to these intermediate states, and sometimes it can see only the final state—that is, the answer. The other two categories of bandwidth are *final states* and *intermediate states.*

To summarize, the three categories, from highest to lowest bandwidth, are mental states, intermediate states, and final states. Each category is intended to include the information in the category beneath it. Mental states includes intermediate and final states. Intermediate states includes final states.

The subject domain of programming provides good examples of the bandwidth dimension because an ITS exists for each bandwidth category. Anderson's LISP Tutor (Reiser, Anderson, & Farrell, 1985) contains a detailed model of the cognitive processes that Anderson believes underlie the skill of programming. The tutor uses a menu-driven interface to offer the student choices about what goals to attack next, what strategies to use, what code fragments to write down, and so on. The model aims to offer so many choices that any problem-solving path that a student wants to take is available. The belief is that the menus do not interfere with the path of mental states but merely allow the ITS to track the student's cognitive progress. Thus, the input to the diagnosis component is an approximation to a sequence of mental states. The LISP Tutor nicely illustrates the bandwidth level of mental states.

The Spade ITS (Miller, 1982) was never completed; but if it had been, it would illustrate the second level of bandwidth. Spade acts as a coach who watches a student program. The student uses a structure editor, that is, an editor that knows about the programming language and allows only syntactically legal edits. Spade sees all the intermediate observable steps as the student creates a program. Unlike Anderson's LISP Tutor, Spade cannot see the student's decisions about programming goals and strategies. Its input bandwidth fits squarely in the category of intermediate states.

In contrast, PROUST (Johnson & Soloway, 1984a; Johnson & Soloway, 1984b) is given only the first complete program that the student submits to a PASCAL compiler. PROUST does not have access to the student's scratch work or incomplete programs.

The bandwidth dimension is perhaps the most important of the three dimensions. More so than the others, it determines the algorithm used for diagnosis. As is shown in Diagnostic Techniques, where diagnosis algorithms are discussed in detail, there are nine basic algorithms. Five are useful with final state bandwidth systems, three are appropriate for intermediate state bandwidth systems, and one is appropriate for mental states bandwidth systems.

Target Knowledge Type

Student models can actually solve the same problems that students do and can therefore be used to predict the students' answers. This is a distinguishing characteristic of the student models used in ITSs. Student models used in older systems for computer-based training

cannot actually generate problem solutions, although they may be able to generate a probability of a correct solution.

Solving problems requires some kind of interpretation process that applies knowledge in the student model to the problem. There are two common types of interpretation, one for *procedural* knowledge and one for *declarative* knowledge.[1] The interpreter for procedural knowledge is simple. It does not search but makes decisions based on local knowledge. It is like a little man with a flashlight who can see only a little way from the strand of knowledge he is standing in; based on his view of the knowledge locale and the current state of the problem, he decides which strand of knowledge to turn onto and follow. A declarative interpreter constantly searches over its whole knowledge base. It is like a librarian who searches out the answer to a client's query by searching reference books, assembling the facts, and deducing the answer from them. Procedural knowledge representations have been used for skills such as algebra equation solving (Sleeman, 1982), game playing (Burton & Brown, 1982; Goldstein, 1982; Goldstein & Carr, 1977), multicolumn arithmetic (Brown & Burton, 1978; Burton, 1982; Langley & Ohlsson, 1984), and solving calculus integrals (Kimball, 1982). Declarative knowledge representations have been used for geography (Carbonell, 1970; Carbonell & Collins, 1973; Grignetti, Hausman, & Gould, 1975) and meteorology (Stevens, Collins, & Goldin, 1982).

The distinction between procedural and declarative knowledge is notorious in artificial intelligence as a fuzzy, seldom useful differentiation. Because it is based on how much work the interpreter does, and because work is an essentially continuous quality, the boundary between them is not sharp and clear. For instance, GUIDON's knowledge of medicine (Clancey, 1982) is partly declarative—because it says what symptoms indicate which diseases—and partly procedural—because it says which questions to ask the patient under what circumstances. PROUST's knowledge of programming (Johnson & Soloway, 1984a, 1984b) is even more difficult to classify. It is mostly about which PASCAL code templates to use to achieve what purposes. In this respect, it is declarative knowledge about PASCAL. However, a simple top-down programming strategy readily converts this knowledge into programmer actions.

Nonetheless, the distinction between procedural and declarative knowledge is important here because the complexity of diagnosis is directly proportional to the complexity of interpretation. In fact, diagnosis is the inverse of interpretation. Interpretation takes a

[1]The section on directions for future research discusses the student modeling problem for a third type of knowledge, qualitative mental models of complex systems.

knowledge base and a problem and produces a solution. Diagnosis takes a problem and a solution and produces a knowledge base. When declarative knowledge is interpreted, many items may be accessed in order to produce a solution. When declarative knowledge is diagnosed, the responsibility for a wrong answer may lie with any one of the many items that could be accessed in producing this answer. In general, the more complicated the interpretation, the more complicated the diagnosis.

These considerations underlie a second dimension in the space of student modeling problems, the *type* of knowledge in the student model. The major distinction—procedural versus declarative—has been mentioned already. It is useful to divide procedural knowledge into two subcategories: *flat* and *hierarchical*. Hierarchical representations allow subgoaling; flat ones do not. For instance, the ACM diagnosis system (Langley & Ohlsson, 1984) uses a flat representation for a subtraction procedure. Operations such as taking a column difference or adding 10 to a minuend digit are selected solely on the basis of the current state of the problem. In the BUGGY diagnosis system (Brown & Burton, 1978), subtraction procedures are represented as goal hierarchies with goals like "Borrow" or "Borrow across zero." Operators are selected on the basis of the problem state *and* the currently active subgoals.

The distinction between flat and hierarchical representations affects the diagnosis. A diagnostic system for flat representation needs to infer what problem-state conditions trigger each operator. This is easy because the system can see both the problem states and the operator applications. A diagnostic system for hierarchical representations needs to infer conditions and both the problem states and the subgoals. But it cannot see the currently active subgoals, so its inference problem is much harder.

In summary, there are three types of knowledge representation: flat procedural, which makes the student modeling problem the easiest; hierarchical procedural, which increases the difficulty of the student modeling problem; and declarative, which makes the student modeling problem most difficult.

Differences Between Student and Expert

ITSs usually employ an expert model as well as a student model.[2] The expert model is used for many purposes, such as providing

[2]In this chapter, "expert" is intended to mean a master of the ITS's subject matter. The subject matter is usually only a fraction of the knowledge possessed by a true expert in that area.

explanations of the correct way to solve a problem. Because students will (one hopes) move gradually from their initial state of knowledge toward mastery, student models must be able to change gracefully from representing novices to representing experts. Consequently, most ITSs use the same knowledge representation language for both the expert model and the student model. Conceptually, the ITS has one knowledge base to represent the expert and a different knowledge base to represent the student.

However, economy and other implementation considerations frequently dictate a merger of the two models. The student model is represented as the expert model plus a collection of differences. There are basically two kinds of differences: missing conceptions and misconceptions. A missing conception is an item of knowledge that the expert has and the student does not. A misconception is an item that the student has and the expert does not.

Some student modeling systems can represent only missing conceptions. Conceptually, the student model is a proper subset of the expert model. Such student models are called *overlay models* because the student model can be visualized as a piece of paper with holes punched in it that is laid over the expert model, permitting only some knowledge to be accessible. A student model, therefore, consists of the expert model plus a list of items that are missing. A variant of overlay modeling puts weights on each element in the expert knowledge base; for example, 1 indicates mastery, -1 indicates ignorance, and 0.5 indicates partial mastery. Overlay models are the most common type of student model.

Other systems represent both misconceptions and missing conceptions. The most common type of student model in this class employs a library of predefined misconceptions and missing conceptions. The members of this library are called *bugs*. A student model consists of an expert model plus a list of bugs. This *bug library* technique is the second most common type of student modeling system. This system diagnoses a student by finding bugs from the library that, when added to the expert model, yield a student model that fits the student's performance.

Assembling the library is the biggest hurdle in the bug library approach. The library should be nearly complete. If a student has a bug that is not in the library, then the student model will try to fit the behavior with some combination of other bugs. It may totally misdiagnose the student's misconceptions.

There are only a few techniques for obtaining a bug library:

1. Bugs can be gleaned from literature, particularly from the older works in the educational literature. For instance, Buswell (1926) listed numerous "bad habits of thought" for arithmetic.
2. Bugs can be found by careful hand analysis of students' behaviors. Hand analysis of several thousand subtraction tests yielded a bug library of 104 bugs for Burton and Brown's DEBUGGY program (Burton, 1982; VanLehn, 1982).
3. If there is a learning theory for the subject domain, it may be able to predict the bugs that students have. For instance, Repair Theory (Brown & VanLehn, 1980; VanLehn, 1982) predicts subtraction bugs. When its predictions were added to DEBUGGY's library and students' tests were reanalyzed, some of the students' answers were fit much better by the new bugs (VanLehn, 1983). So, theory can be a valuable contributor of bugs to a bug library.

An alternative to the bug library approach is to construct bugs from a library of bug parts. Bugs are constructed during diagnosis rather than being predefined. For instance, each bug constructed by the ACM system (Langley & Ohlsson, 1984) is a production rule consisting of a condition, which is a conjunction of predicates, and a single action. The predicates and the action are drawn from predefined libraries. If the predicate library has P predicates, and the action library has A actions, then ACM can represent approximately $A * 2^P$/distinct bugs. As in the bug library approach, a student model may have more than one bug. So ACM can represent a very large number of student models using only two small libraries of bug parts. Of course, the libraries of bug parts must be assembled by the creators of the ITS. The problems of filling these libraries are exactly analogous to the problem of filling a bug library. However, because libraries of bug parts are smaller, the problems may be easier to solve. This approach to representing differences between the student and the expert is the newest and least common. Its properties are largely unknown.

To summarize, the three major techniques for representing differences between the student ahd the expert are overlays, bug libraries, and bug part libraries.

A Chart of the Space

The preceding section defined three dimensions of student models, each with three distinguished values. Figure 3.1 summarizes them. Under each dimension, the order of the categories corresponds to the difficulty of the diagnostic problem, easiest first. There are 3^3 possible student models. The student models that make diagnosis easiest are

overlay models on flat procedural knowledge, where the student's mental states are available to the diagnostic program. The hardest problem is a bug-parts-library student model over declarative knowledge when only the final result of the student's reasoning is available to the diagnostic program.

Not all of the 27 possible types of student models have been implemented. Figure 3.2 shows some of the existing student modeling systems and their location in the space of the student models. The bandwidth dimension is the *Y*-axis and the knowledge type dimension is the *X*-axis. The student-expert differences dimension is indicated by asterisks: ** means a bug parts library, * means a bug library, and no asterisks means an overlay. The ITSs referenced in the figure are all quite complex, and there is ample room for disagreement over how they should be classified.

DIAGNOSTIC TECHNIQUES

Nine diagnostic techniques have appeared so far in the ITS literature. This section reviews them one by one. Most techniques have been used in just a few kinds of student models. As a framework for further discussions, Figure 3.3 shows how the diagnostic techniques align with the student models. The space of student models is shown in the same format as Figure 3.2; but the cells are filled with the names of the diagnostic techniques that have been employed in the corresponding student modeling systems. It is important to note that this chart is based on actual systems and the diagnostic techniques they use. It is likely that some of the techniques can be used with other types of student models.

Model Training

The model-tracing technique (Anderson, Boyle, & Yost, 1985) is probably the easiest technique to implement because it assumes that all of the student's significant mental states are available to the diagnostic program. The basic idea is to use an underdetermined interpreter for modeling problem solving. At each step in problem solving, the underdetermined interpreter may suggest a whole set of rules to be applied next, whereas a deterministic interpreter can suggest only a single rule. The diagnostic algorithm fires all these suggested rules, obtaining a set of possible next states. One of these states should correspond to the state generated by the student. If so, then it is

1. Bandwidth -- How much of the student's activity is available to the diagnostic program?
    a. Mental states -- All the activity, both physical and mental, is available.
    b. Intermediate states -- All the observable, physical activity is available.
    c. Final states -- Only the final state -- the answer -- is available.
2. Knowledge Type -- What is the type of the subject matter knowledge?
    a. Flat procedural -- Procedural knowledge without subgoaling.
    b. Hierarchical procedural -- Procedural knowledge with subgoals.
    c. Declarative.
3. Student-Expert Difference -- How does the student model differ from the expert model?
    a. Overlay -- Some items in the expert model are missing.
    b. Bug library -- In addition to missing knowledge, the student model may have incorrect "buggy" knowledge. The bugs come from a predefined library.
    c. Bug part library -- Bugs are assembled dynamically to fit the student's behavior.

FIGURE 3.1 The three dimensions of student models.

Knowledge type / Bandwidth	Procedural-flat	Procedural-Hierarchical	Declarative
Mental States		**Kimball's calculus tutor **Anderson's LISP tutor **Anderson's Geometry tutor	GUIDON
Intermediate States	WEST WUSOR	**The MACSYMA Avisor **Spade **Image	*SCHOLAR *WHY *GUIDON
Final States	**LMS **Pixie **ACM	*BUGGY *DEBUGGY *IDEBUGGY	*MENO *PROUST

FIGURE 3.2 The space of student models.

Knowledge type / Bandwidth	Procedural-flat	Procedural-Hierarchical	Declarative
Mental States		Model tracing	
Intermediate States	Issue tracing	Plan recognition	Expert system
Final States	Path finding Condition Induction	Decision tree Generate and test Interactive	Generate and test

FIGURE 3.3 Diagnostic techniques.

reasonably certain that the student used the corresponding rule to generate the next mental state and so must know that rule. The student model is updated accordingly. The name *model tracing* comes from the fact that the diagnostic program merely traces the (underdetermined) execution of the model and compares it to the student's activity.

Obviously, the model of problem solving must be highly plausible psychologically for this technique to be applicable. Even if such a model is available, practical deployment of this technique requires solving several tricky technical issues. Here are just three: (a) What should the system do if the student's state does not match any of the states produced by the rules in the model? (b) Suppose the student generates a next state by guessing or by mistake; the system will erroneously assume that the student knows the corresponding rule; (c) When should the system change its mind about its student model?

Path Finding

If the bandwidth is not high enough to warrant the assumption that the student has applied just one mental rule, then model tracing is inapplicable. However, it is feasible to put a path-finding algorithm in front of the model-tracing algorithm. Given two consecutive states, it finds a path, or chain of rule applications, that takes the first state into the second state. The path is then given to a model-tracing algorithm, which treats it as a faithful rendition of the student's mental state sequence.

The main technical problem with path finding is that there are usually many paths between the two given states. Should the path finder send all the paths to the model tracer and let it deal with the ambiguity? Should it use heuristics to reject unlikely paths? (Ohlsson's DPF system [Ohlsson & Langley, 1985] takes this approach.) Should it ask the students what they did? These issues deserve further research.

Condition Induction

Model tracing assumes that any two consecutive states in the student's problem solving can be connected by a rule in its model. This puts strong demands on the completeness of the model. Overlay models often will not work. Bug library models must contain a large number of bugs. Bug part libraries are therefore used as the basis for student modeling. Given two consecutive sates, the system *constructs* a rule that converts one state to the other. Although there are potentially many ways to construct such buggy rules, the only technique that has been tried so far is *condition induction* (Langley & Ohlsson, 1984).

This technique requires two libraries. One is a library of operators that convert one state to another. The other is a library of predicates. The technique assumes that the operator library is rich enough that any two consecutive mental states can be matched by applying some operator. That operator becomes the action side of the production rule that will be generated. The hard job is determining what logical combination of predicates should constitute the condition side of the production. The condition should be true of states in which the rule was applied and false otherwise. The system currently has one state for which it is true; that is, the first state in the state pair. In order to reliably induce a condition, it needs to examine more states. These states can come from a record of the student's past problem solving. The system can also delay construction of the rule until more states are examined in later problem solving. This technique seems to require much more data on the student's problem solving than diagnostic techniques for overlay models or bug library models do. This is just what one would expect from information theory. The bug part library can represent many more hypotheses than the other kinds of models can, so more data is needed to discriminate among them.

Plan Recognition

In principle, path finding followed by model tracing, with or without rule induction, can diagnose anything, However, when the paths

between observable states get long, diagnosis may become infeasible or unreliable. Plan recognition is a diagnostic technique that is smiliar to path finding in that it is a front end to model tracing. However, it is more effective than path finding for the special circumstances in which it applies.

Plan recognition requires that the knowledge in the student model be procedural and hierarchical and that all or nearly all of the physical, observable states in the student's problem solving be made available to the diagnostic program. These two requirements together dictate that an episode of problem solving can be analyzed as a tree. The leaves of the tree are primitive actions, such as moving a chess piece or writing an equation down. The nonleaf nodes in the tree are subgoals, such as trying to take the opponent's queen or factoring $x^2 + 3x - 1$. The root node of the tree is the overall goal (e.g., Win this chess game, or solve $x[x + 4] - x = 1$). Links between nodes in the tree represent goal-subgoal relationships. Such a tree is often called a *plan*—a misnomer from its early development in robotics. Plan recognition is the process of inferring a plan tree when only its leaves are given. Computationally, plan recognition is similar to parsing a string with a context-free grammar—a parse tree is constructed whose leaves are the elements of the string. The CIRRUS system, (VanLehn and Garlick, 1987) uses parsing for plan recognition.

When plan recognition is used for diagnosis, it serves as a front end to model tracing. Assuming that plan recognition can find a unique plan tree that spans the student's actions, then the student's mental path is assumed to be a depth-first, left-to-right traversal of the tree. This path can be input to a model-tracing algorithm, which updates the student model accordingly.

There are two technical issues to confront: What if the plan recognizer finds more than one tree that is consistent with the student's actions? What if it doesn't find any? To avoid the second situation, plan recognition systems often use bug library models rather than overlay models. Bug part library models could also be used by taking advantage of a machine-learning technique called *learning by completing explanations* (VanLehn, 1987). The diagnosis programs that have used plan recognition (Genesereth, 1982; London & Clancey, 1982; Miller, 1982) have been more concerned with the first problem, that is, determining which plan tree among several trees consistent with the student's actions is most plausibly the student's mental plan. These programs use a variety of heuristics.

Issue Tracing

The model-tracing technique assumes that the rules in the student model are a fairly accurate psychological model of the units of knowledge employed by a student. In some cases, such a detailed model of student cognition is infeasible or unnecessary. In particular, a fine-grained student model is probably more work than it's worth if the tutoring cannot be adapted to the intricacies of a particular student's misconceptions. For instance, a perfect model of a student's subtraction bug is necessary if the tutor's remedy is merely to teach the procedure over again. In general, the level of diagnosis and tutoring should be the same.

If a coarse-grained student model is desired, then a variant of model tracing called *issue tracing* is appropriate. It is based on analyzing a short episode of problem solving into a set of microskills or issues that are employed during that episode. The analysis does not explicate *how* the issues interacted or what role they played in the problem solving. It claims only that the issues were used.

The WEST system (Burton & Brown, 1982) pioneered this diagnostic technique. Its task is to teach a simple board game. A turn consists of choosing an arithmetic combination of three randomly chosen numbers in such a way that the value of the expression, when added to the current position of the player's token, results in a new position that is closer to the goal position. Expressions may contain any arithmetic operation or parentheses. There are several tricks involving "bumping" an opponent or taking a shortcut. WEST analyzes a student's move into several issues, including *plus, minus, times, divide, parentheses, bump,* and *shortcut.* If a student forms the expression 5*2 - 1, then the move is analyzed as involving the issues *times* and *minus,* and not involving the others. The student's actual problem solving probably involved trying several expressions, seeing where they moved the token, and selecting the expression that maximized progress toward the goal. A model-tracing technique would have to model this trial-and-error search in gory detail. The issue-tracing technique ignores the details. Its analysis claims only that the student apparently understands these two issues because the student's move embodied them.

The first step in issue tracing is to analyze the student's move and the expert's move into issues. Each issue has two counters, used and missed. Used counters are incremented for all the issues in the student's move. Missed counters are incremented for all the issues in the expert's move that are not in the student's move. If the used counter is high and the missed counter is low, the student probably understands the issue. If the missed counter is high and the used counter is low, then the student probably does not understand the issue. If both

counters are zero, the issue has not come up yet.[3]

This simple diagnostic procedure has a hidden problem. Ignorance of any one of the issues involved in an expert's move is sufficient to cause the student to overlook that move; yet issue tracing blames all the issues evenly by incrementing all their missed counters. This introduces some inaccuracy into the student model. WEST's solution is to require that the ratio missed/used be fairly high before it assumes that the student needs tutoring on that issue. WUSOR (Goldstein, 1982; Goldstein & Carr, 1977) has a more complicated scheme. It has a system of expectations about what issues are likely to be learned first and what issues typically follow later.

These prior probabilities are folded into the evaluation of whether a student knows an issue or not. Evaluations based on statistical functions have been used in Kimball's calculus tutor (Kimball, 1982) and other systems for similar purposes.

Expert Systems

Clancey's GUIDON system (Clancey, 1982) uses a large-grained student model just as WEST and WUSOR do. Instead of issues, GUIDON uses inference rules. The rules concern medical diagnosis and model moderately large chunks of knowledge that summarize a variety of cognitive operations. A typical rule is:

Rule 545

if 1. the infection was acquired while the patient was hospitalized, and
2. the white blood cell count is less than 2.5 thousand,

then

a. there is strong evidence that the organism is E. coli, and
b. there is suggestive evidence that the organism is Klebsiella pneumonia, and
c. there is suggestive evidence that the organism is Pseudomonas.

Because such rules are more complicated than issues, the diagnosis problem is harder. For instance, if a student is given a case that matches the antecedent clauses in Rule 545, and yet the student hypothesizes only one of the conclusions (e.g., conclusion (a), that the organism is E. coli) but not the other two, then it is not clear whether the student has used the rule or not. Another rule, triggered by some other feature of the case, may have led the student to conclude that the organism was E. coli.

There are many possible ways for rules to interact. To handle the myriad of combinations, GUIDON uses an expert systems approach. It has dozens of diagnostic rules such as this one:

if 1. the student's hypotheses include ones that can be concluded by this rule, and
2. the student's hypotheses do not include all the conclusions of this rule,

then

a. decrease the degree of belief that the student knows this rule by 70%.

This particular diagnostic rule applies in the situation just described. GUIDON, which uses an overlay model with continuous weights, accordingly downgrades the weight in the student model for Rule 545.

The basic idea of the expert systems approach to diagnosis is to provide diagnostic rules for all the situations that arise. Some technical issues are: If two diagnostic rules match the current situation, how are their conclusions combined? What if no diagnostic rule matches? How much will diagnostic rules have to change if the rules in the knowledge base for the task domain change?

Decision Trees

All diagnostic techniques must deal with the fact that students rarely have just one knowledge deficit. They usually have several. Some of the techniques described earlier—notably model tracing, path finding, and plan recognition—assume that at most one rule fires between consecutive mental states, so each deficit will show up in isolation as a buggy rule application. Because bugs appear in isolation, each bug can be accurately diagnosed even when there are several of them. Systems like WEST and GUIDON, which have less bandwidth, use a less accurate description of knowledge deficits (e.g., weakness on issues), which allows them to model combinations of deficits simply.

The next three techniques aim for highly accurate diagnoses with low bandwidths. They all work with final states, which constitutes

[3] If both counters are high, the model is inadequate in some way. This situation is called *tear* (Burton & Brown, 1982). In WEST, it occurred when the student's objective in the game was not what it was assumed to be (i.e., some students did not care about winning but just wanted to bump their opponent as often as possible). WEST is equipped to handle this. It searches for the student's objective by generate and test. It has lists of possible student objectives from which it can choose, and it then reanalyzes the entire game using that objective. If the tear is reduced, then WEST has found the student's objective.

Problems:	50	712
	-28	- 56
Answers:		
0 - N = 0	30	656
N - M = |N - M|	38	744
Both	30	744

FIGURE 3.4 Two bugs, in isolation and co-occurring.

the lowest bandwidth in the student model space. The student models are based on bug libraries. The bugs are highly accurate: When installed, they predict the sequence of intermediate states and perhaps even the sequence of mental states.

Diagnosis of multiple bugs would be simple if systems could generate the symptoms of co-occurring bugs by taking the union of the symptoms they display in isolation. This is not always possible. To illustrate, Figure 3.4 shows two subtraction bugs, in isolation and co-occurring. On the first problem, 50 - 38, the answer of the co-occurring bugs, 30 equals the answer of the first bug in isolation. On the second problem, 712 - 56, the answer matches the answer of the second bug in isolation, even though the first bug also gets this problem wrong when it occurs in isolation. When the second bug occurs in isolation on 712 - 56, the borrow in the units column changes the tens column to 0 - 5, which triggers the bug. When the second bug occurs together with the first, it suppresses the borrow, so the tens column remains 1 - 5, and the first bug is not triggered. In this simple case, there is a causal interaction between the two bugs that makes them manifest differently. In general, bugs can interact in even more complex ways.

The decision-tree technique is a brute force approach to bug compounding. It was employed by the BUGGY diagnostic system (Brown & Burton, 1978). BUGGY enlarged the library of bugs by forming all possible pairs. Because there were 55 bugs, this expansion generated about 55^2(= 3025) bug pairs. In order to efficiently diagnose this many bugs, BUGGY preanalyzed the subtraction test that students were given and formed a decision tree that indexed the bugs by their answers to the problems. The top node of the tree corresponds to the first problem. Answers from all possible diagnoses (a diagnosis is a bug or a bug pair) are collected. Most answers will be generated by several diagnoses. For each answer, a daughter node is attached to the root node, labeled by the answer. Associated with each node

are the diagnoses that gave that answer. The tree-building operation recurses, once for each new node, using the second test problem. When BUGGY is finished, a huge tree has been built. Each diagnosis corresponds to a path from the root to some leaf. If the test items are well chosen, then every such path is unique—each leaf corresponds to exactly one diagnosis. In general, it is very difficult to find a short test with such high diagnostic capabilities. Burton (1982) discusses this important issue further.

All this tree building occurs before any students are seen. It is the most expensive part of the computation. Diagnosis of a student's answers is simple, at least in principle. If a student makes no careless errors, then his or her answers are used to steer BUGGY on a path from the root to the diagnosis that is appropriate. Of course, most students do make unintentional errors (often called *slips*, to distinguish them from bug-generated errors), such as subtracting 9 - 5 and getting 3. Slips mean that a simple tree-walk will not always lead to a leaf, so BUGGY performs a tree search to find a diagnosis while allowing a minimal number of slips.

The advantage of the decision-tree approach is that the tree search is simple enough to be implemented on a microcomputer. A larger computer can be used for the computationally intensive tree-building process. The disadvantage of this technique is that it does not really handle multiple co-occurring bugs. Instead, it computes in advance all possible combinations (pairs, in BUGGY's case) and treats the bug combinations just like primitive bugs. This is usually too expensive if more than two bugs can occur together. Burton's hand analysis of the data uncovered students with four co-occurring bugs. For BUGGY to diagnose these students would require approximately 55^4(= 9 million) bug tuples, which means a diagnostic tree with trillions of nodes.

Generate and Test

DEBUGGY (Burton, 1982) was designed to diagnose up to four or five multiple co-occurring bugs. Unlike BUGGY, it does not calculate the answers of co-occurring bugs in advance. Rather, it generates bug combinations dynamically. It begins by finding a small set of bugs that match some, but not necessarily all, of the student's answers. There might be 10 bugs in this set. It then forms all these bugs (about 100 bug pairs). It also makes pairs using a stored list of bugs that are known to be difficult to spot because they are often covered by other bugs. From this set of perhaps 200 bugs, DEBUGGY selects the ones that best match the student's answers. Using these favorites,

the bug-compounding process occurs again and again until no further improvement in the match is found. The resulting tuple of bugs is output as DEBUGGY's diagnosis of the student.

DEBUGGY's algorithm is a species of a very general technique for diagnosis, called *generate and test*. The diagnostic algorithm generates a set of diagnoses, finds the answers that each predicts, tests those answers against the student's answers, and keeps the ones that match best. In general, generate and test is rather inefficient. Domain-specific heuristics are often needed in order to speed it up.

Interactive Diagnosis

DEBUGGY and BUGGY work with a predefined subtraction test and the student's answers to it. Thus, they can be used as off-line diagnostic systems: The teacher administers the test, mails the answers to DEBUGGY, gets the diagnosis a few days later, and administers the appropriate remedial instruction. VanLehn (1982) reported the results of such a use of DEBUGGY.

Within a tutoring system, there is no need to stick with a fixed list of test items. The system can choose a problem whose answer will help diagnosis the most. IDEBUGGY (Burton, 1982) is such a system. Given a set of diagnoses consistent with the students' answers so far, it tries to construct a subtraction problem that will cause each diagnosis to generate a different answer. Thus, the problem splits the hypothesis space, so to speak. It is not always possible to find such a problem, so IDEBUGGY puts only a fixed amount of effort into this strategy, then presents the best problem it has found so far to the student. Still, the student can sometimes wait too long for IDEBUGGY to present the next problem. Interactive diagnosis, where the diagnosis algorithm drives the tutorial interaction, puts heavy demands on the speed of the diagnostic algorithm. Nonetheless, it can yield highly accurate diagnoses with many fewer test items than a fixed-item test would require in order to achieve the same accuracy. Reducing the length of the diagnostic session may reduce students' fatigue and increase their willingness to cooperate.

RESEARCH ISSUES

Cognitive diagnosis is a new field, and there are vast numbers of questions for research to address. There are many questions of the form "Does technique X work well with student models of form Y on subject domain Z?" From an engineering and educational standpoint, these are the most important questions to address, for they turn a miscellaneous collection of techniques, each of which has been used once or twice, into a well-understood technology. To this collection of issues, I would nominate a few more that are not of the XYZ form.

Most research has gone into finding diagnostic techniques that can produce very detailed descriptions of the students' knowledge. Simpler techniques such as issue tracing produce less detailed descriptions. There is a tacit assumption that tutoring based on fine-grained student models will be more effective than tutoring based on coarse-grained models. No one has attempted to check this assumption. We need to know when fine-grained modeling is worth the effort. This is not really a question of how to do student modeling but, rather, when to do what kinds of student modeling. In order to address this question, one could situate two or more student modeling systems inside the same ITS and see which one tutors more effectively.

Research oriented toward improving student modeling could go in several directions. One is to employ explicit models of learning. This topic was touched on in the WUSOR ITS (Goldstein & Carr, 1977; Goldstein, 1982). Incorporating models of learning into diagnosis has much potential power because it can radically reduce the space that the diagnostic algorithm must search.

Interactive diagnosis, where the diagnostic program selects problems to pose to the student, is another technique that has great potential power. It has been briefly explored with the IDEBUGGY student modeling program (Burton, 1982), the GUIDON ITS (Clancey, 1982), and the WHY project (Stevens, Collins, & Goldin, 1982). This topic—the skill of posing problems—seems almost as rich as diagnosis, which is the skill of interpreting the student's answers to problems.

As user interfaces improve and powerful personal computers become cheaper, we are likely to see more ITS designers choosing the high bandwidth option, where the student's behavior is very closely monitored by the system. The amount of time between the student's actions is one type of information that is available for free but that so far has been ignored by every ITS I know of. Chronometric data has been used in psychology for years as a basis for deciding between potential models of human cognition. It would be interesting to see whether chronometric data would favor fine-grained student modeling.

Much of the early ITS research concerned students learning about physical systems. The SOPHIE project (Brown, Burton, & deKleer, 1982) studied students learning about electronic circuits. The WHY project (Stevens, Collins, & Goldin, 1982) studied rainfall. The Steamer

project (Hollan, Hutchins, & Weitzman, 1984) studied naval steam plants. These projects gradually evolved into long-term, basic research on the mental models that people seem to employ for mentally stimulating physical systems (Gentner & Stevens, 1984). Research on mental models has progressed to the point that it might be worth reopening the investigation into ITSs for physical systems. The student modeling problem will be very difficult. The students' responses depends on mentally running a model constructed from their understanding of the device. If the response is wrong, it could be because of a bug in how they ran their mental model, or in how they constructed it, or in both. Relative to the three-dimensional space of student models, mental models are a brand-new knowledge type—a new column in the chart of Figure 3.2—with unique new technical issues to conquer.

REFERENCES

Anderson, J. R., Boyle, C., & Yost, G. (1985). The geometry tutor. *Proceedings of Ninth International Joint Conference on Artificial Intelligence* (pp. 1-7). Los Altos, CA: Morgan Kaufmann.

Brown, J. S., & VanLehn, K. (1980). Repair Theory: A generative theory of bugs in procedural skills. *Cognitive Science, 4*, 379-426.

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

Brown, J. S., Burton, R. B., & deKleer, J. (1982). Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 227-282). New York: Academic Press.

Burton, R. B. (1982). DEBUGGY: Diagnosis of errors in basic mathematical skills. In D. H. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 157-183). New York: Academic Press.

Burton, R. B., & Brown, J. S. (1982). An investigation of computer coaching for informal learning activities. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 79-98). New York: Academic Press.

Buswell, G. T. (1926). *Diagnostic studies in arithmetic*. Chicago, IL: University of Chicago Press.

Carbonell, R. (1970). AI in CAI: An artificial intelligence approach to computer aided instruction. *IEEE Transactions on Man-Machine Systems, 11*, 190-202.

Carbonell, J. R., & Collins, A. (1973). Natural semantics in artificial intelligence. *Proceedings of the Third International Joint Conference on Artificial Intelligence* (pp. 344-351). Los Altos, CA: Morgan Kaufmann.

Clancey, W. J. (1982). Tutoring rules for guiding a case method dialogue. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 201-225). London: Academic Press.

Genesereth, M. R. (1982). The role of plans in intelligent teaching systems. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 137-155). New York: Academic Press.

Gentner, D., & Stevens, A. (1984). *Mental models*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Goldstein, I. (1982). The genetic graph: A representation for the evolution of procedural knowledge. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 51-77). New York: Academic Press.

Goldstein, I., & Carr, B. (1977). The computer as coach: An athletic paradigm for intellectual education. *Proceedings of ACM77* (pp. 227-233).

Grignetti, M., Hausman, C., & Gould, L. (1975). An intelligent on-line assistant and tutor: NLS-Scholar. *Proceedings of the National Computer Conference* (pp. 775-781).

Hollan, J. D., Hutchins, E. L., & Weitzman, L. (1984). Steamer: An interactive inspectable simulation-based training system. *The AI Magazine, 5*(2), 15-27.

Johnson, L., & Soloway, E. (1984a). Intention-based diagnosis of programming errors. *Proceedings of American Association of Artificial Intelligence Conference* (pp. 162-168). Los Altos, CA: Morgan Kaufmann.

Johnson, L., & Soloway, E. (1984b). PROUST: Knowledge-based program debugging. *Proceedings of the Seventh International Software Engineering Conference* (pp. 369-380).

Kimball, R. (1982). A self-improving tutor for symbolic integration. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 283-308). New York: Academic Press.

Langley, P., & Ohlsson, S. (1984). Automated cognitive modeling. *Proceedings of American Association of Artificial Intelligence* (pp. 193-197). Los Altos, CA: Morgan Kaufman.

London, B., & Clancey, W. J. (1982). Plan recognition strategies in student modeling: Prediction and description. *Proceedings of the American Association of Artificial Intelligence Conference* (pp. 193-197). Los Altos, CA: Morgan Kaufmann.

Miller, M. L. (1982). A structured planning and debugging environment for elementary programming. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 119-135). New York: Academic Press.

Ohlsson, S., & Langley, P. (1985). *Identifying solution paths in cognitive diagnosis* (Tech. Rep. CMU-RI-TR-84-7). Pittsburgh, PA: Carnegie-Mellon University, Robotic Institute.

Reiser, B. V., Anderson, J. R., & Farrell, R. G. (1985). Dynamic student modeling in an intelligent tutor for lisp programming. *Proceedings of Ninth International Joint Conference on Artificial Intelligence* (pp. 8-14). Los Altos, CA: Morgan Kaufman.

Sleeman, D. H. (1982). Assessing competence in basic algebra. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 186-199). New York: Academic Press.

Stevens, A., Collins, A., & Goldin, S. E. (1982). Misconceptions in students' understanding. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems* (pp. 13-24). New York: Academic Press.

VanLehn, K. (1982). Bugs are not enough: Empirical studies of bugs, impasses and repairs in procedural skills. *The Journal of Mathematical Behavior, 3*(2), 3-71.

VanLehn, K. (1983). *Felicity conditions for human skill acquisition: Validating an AI-based theory* (Tech. Rep. CIS-21). Palo Alto, CA: Xerox Palo Alto Research Center.

VanLehn, K. (1987). Learning one subprocedure per lesson. *Artificial Intelligence, 31*(1), 1-40.

VanLehn, K., & Garlick, S. (1987). Cirrus: An automated protocol analysis tool. In P. Langley (Ed.), *Proceedings of the Fourth Machine Learning Workshop*. Los Altos, CA: Morgan Kaufmann.