Learning One Subprocedure per Lesson

Kurt VanLehn

Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A.

Recommended by Tom Mitchell

ABSTRACT

SIERRA is a program that learns procedures incrementally from examples, where an example is a sequence of actions. SIERRA learns by completing explanations. Whenever the current procedure is inadequate for explaining (parsing) the current example, SIERRA formulates a new subprocedure whose instantiation completes the explanation (parse tree). The key to SIERRA's success lies in supplying a small amount of extra information with the examples. Instead of giving it a set of examples, under which conditions correct learning is provably impossible, it is given a sequence of "lessons," where a lesson is a set of examples that is guaranteed to introduce only one subprocedure. This permits unbiased learning, i.e., learning without a priori, heuristic preferences concerning the outcome.

1. Introduction

Much research in machine learning has concentrated on induction, i.e., learning from examples. Understanding induction is certainly one of the great intellectual challenges of our times. Induction stands at the center of both the psychology of learning and the philosophy of science.

More recently, induction has been heralded as a potential solution to the knowledge acquisition problem of expert systems. However, AI has not had much success at applying induction to practical problems. The difficulty is this: In practical settings, only a finite set of examples can be presented to the learner, but the knowledge representation language usually has enough expressive power that there are always infinitely many representable generalizations that are consistent with the examples that the learner has seen. (The formal results that justify this assertion will be reviewed later.) That is, the examples just do not contain enough information to correctly identify the target generalization. Consequently, the learner must guess. AI research on induction has consisted largely of studying the efficacy and domain independence of various heuristics for guessing. In this respect, the induction problem is quite different

Artificial Intelligence **31** (1987) 1–40

from other kinds of AI problems, e.g., recognition or synthesis, where correct answers are computable in principle, but non-AI techniques would take astronomically long to compute them.

The inherent uncertainty of induction suggests studying forms of quasi-inductive learning, where a teacher gives some extra information to the learner in addition to the examples. Winston argues for "the importance of good training sequences prepared by good teachers. I think it is reasonable to believe that neither machines nor children can be expected to learn much without them." [47, p. 6] As Winston [48–50] has pointed out, there is a spectrum defined by how much extra information is provided. On the one end of the spectrum is induction, where all the learner receives is examples. On the other end is learning by being told, where the learner is given a complete description of the target generalization in some language. The kind of learning studied here falls closer to the induction end of the spectrum end than to the learning-by-being-told end, because very little extra information is supplied. It could be called *learning from lesson sequences*, because the extra information given to the learner is embedded in the way that the examples are partitioned into lessons and the way the lessons are sequenced. In the last section of this article, a variant of learning from lesson sequences will be discussed wherein lessons are omitted, and the example sequence alone carries all the extra information.

One of the most important areas for applying induction is in the learning of procedures. Procedure learning is the central problem in programming by examples [1, 7, 8, 40, 41], in psychological modelling of skill acquisition [2, 3, 33, 44, 45], and in automating protocol analyses [6]. Machine learning of procedures has been suggested as one way to solve the knowledge acquisition problem for expert systems [31], and as a technique for modelling students in intelligent tutoring system [25, 27]. Undoubtedly, there are many other applications for procedure learning systems that have not yet been studied.

An obvious testbed for the new approach of learning from lesson sequences is a system to learn procedures. This article presents SIERRA, a system that learns procedures from lesson sequences.

1.1. The particular learning task solved by SIERRA

SIERRA was built for a specific application, a psychological study of the acquisition of mathematical skills, arithmetic in particular [44, 45]. Although it would make the reader's job easier if SIERRA's techniques were displayed in a toy problem domain, this article presents an unsimplified, accurate picture. First, the kinds of examples given to SIERRA will be described, followed by a description of lesson sequences.

Two general kinds of procedure induction problems have been addressed in the literature. The harder one is learning a procedure from input-output pairs

[1]. The one studied here is learning from action sequences.[1] It is assumed that the agent that executes procedures is like a human or a robot in that its procedures manipulate both (1) an external world that all agents have access to, and (2) an internal state, which is private. The internal state might include a stack, for instance. An action sequence consists of a sequence of state changes to the external world (or equivalently, a sequence of world states). The learner cannot see the internal state of the teacher during an action sequence. Action sequences are "examples" of the target procedure's execution. The induction task is to infer a procedure from such examples. There are two kinds of examples, positive and negative. A *positive* example is an action sequence that an induced procedures should generate when it is run on the problem that appears as the initial state in the action sequence. A *negative* example is an action sequence that an induced procedure should *not* generate.

SIERRA's input is an ordered sequence of lessons, where a lesson is an unordered set of examples. Lessons contain only positive examples. Each lesson is marked with a single bit. The bit is 1 if the lesson is a "normal" lesson, and 0 if it is an "optimization" lesson.[2] In a moment, the semantics of lessons and their marks will be described. The point to note here is that the extra information given to SIERRA over and beyond the set of examples consists only of (1) the partition of the examples into lessons, (2) the ordering of the lessons, and (3) the binary mark on lessons. Although this is very little extra information, it vastly simplifies SIERRA's induction task. The next section discusses why.

SIERRA learns procedures incrementally. Each lesson builds on the procedure learned in a previous lesson. This can be best illustrated with a familiar procedure, such as the procedure for ordinary multicolumn subtraction. Figure 1 shows a lesson sequence for subtraction. There are six lessons. Although a typical SIERRA lesson has about five examples, the figure shows only the first example from each lesson. Each example is shown as a sequence of states. Figure 2 shows a corresponding sequence of procedures, where the procedures are sketched as augmented transition nets (ATNs). The procedures correspond to the lessons as follows: P0 is induced from lesson L0; P1 from P0 and L1; P2 from P1 and L2; etc. The last procedure P5, is a complete correct subtraction procedure.[3]

[1] In early work [7, 8] this form of learning was called learning from traces.

[2] A larger vocabulary of markers could be used, e.g., to indicate a lesson of negative examples.

[3] Actually, SIERRA generates many procedures other than the ones shown here. For instance, the first borrowing lesson, L3, produces two procedures from P2, and only one of them is shown in Fig. 2. Because of such branching, SIERRA actually produces a six-ply tree of procedures from the lesson sequence, with a branching factor of 1.8. This branching models the empirical fact that not all human students learn identical subtraction procedures, even though they receive the same lessons [44].

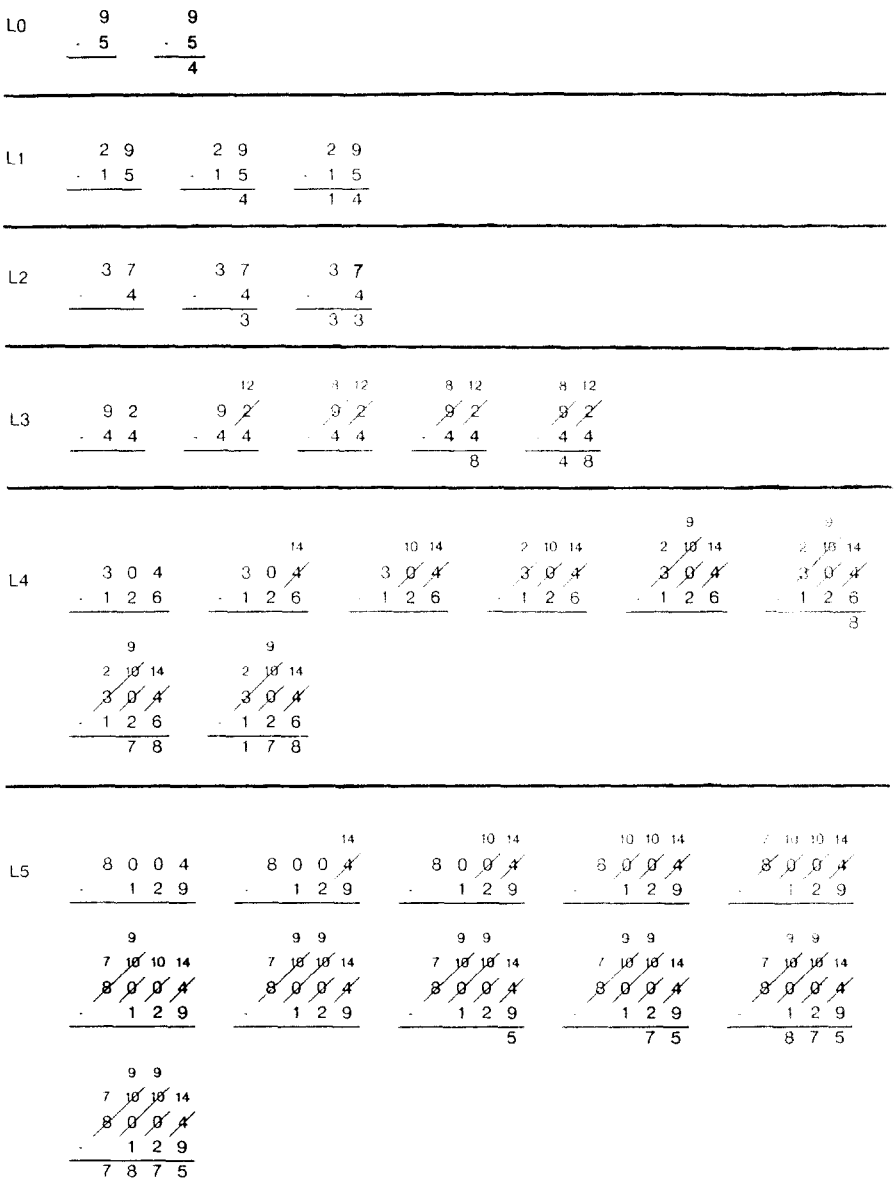


FIG. 1. A simple lesson sequence for subtraction.

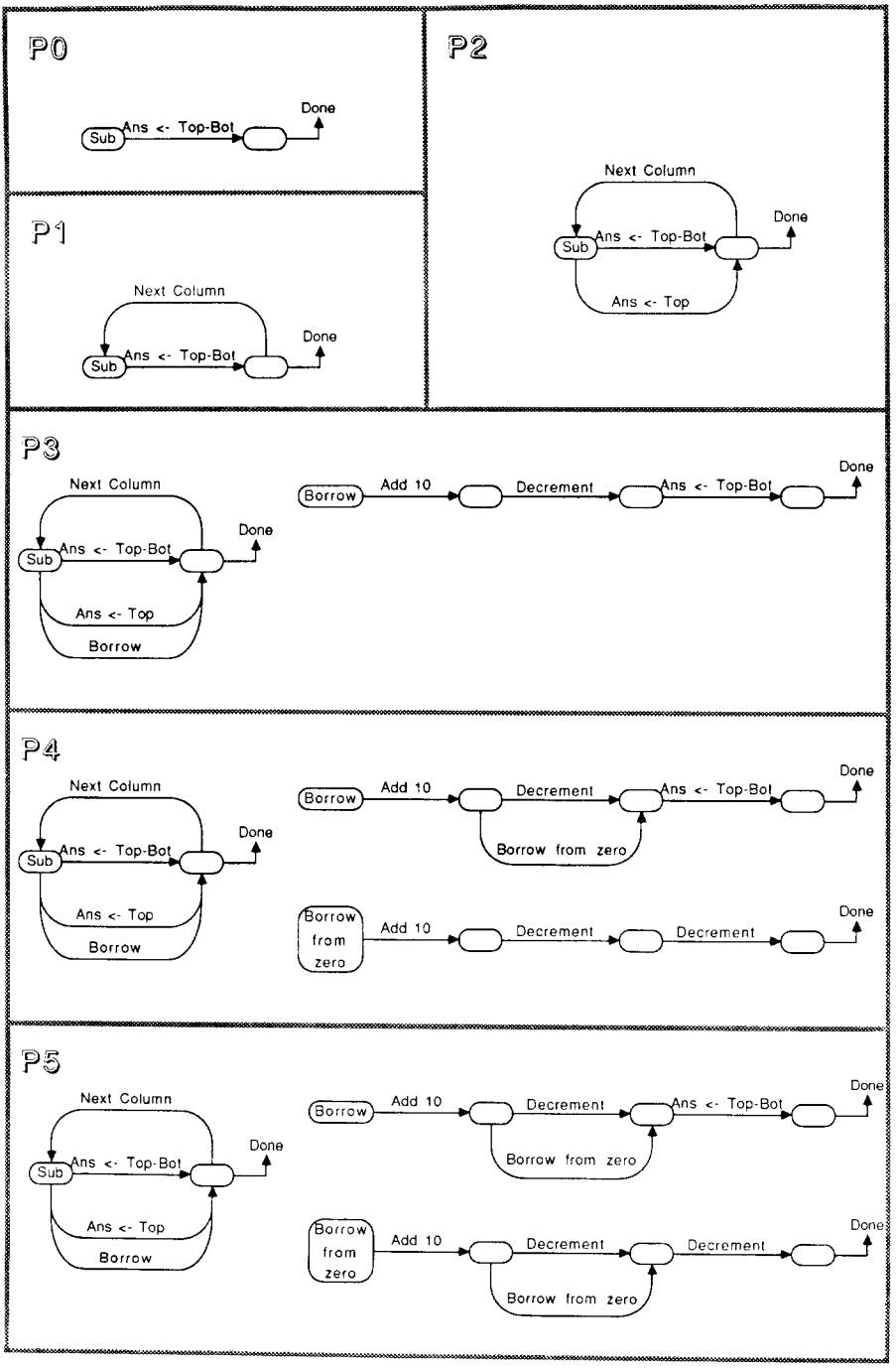


FIG. 2. Procedures produced while traversing the lesson sequence of Fig. 1.

1.2. Why lesson sequences simplify procedure induction

An earlier discussion put learning from lesson sequences on the same scale as learning by being told. This may seem strange. Information encoded in the lesson sequence does not seem like a linguistic expression. However, it functions in exactly the same way. Linguistic expressions have meaning for the learner only under interpretation. The conventions governing the interpretation are known by *both* the teacher and the learner. The teacher generates explanations in such a way that the learner will, ideally, interpret them the way the teacher wants them to be interpreted. The same kind of convention-driven interpretation underlies the use of lesson sequences. The formatting information (i.e., the partition, the order, the marks) is generated by the teacher who understands the interpretation that that learner will place on that information. Two interpretive conventions are explored here:

(1) A normal lesson introduces at one subprocedure. Roughly put, a subprocedure is like one COND clause in LISP: a test, which if true causes an implicit PROGN of function calls to be executed. A precise definition of "subprocedure" will be given after the knowledge representation language for procedures is described.

(2) A normal lesson introduces material that will allow the learner to solve problems that it could not solve before. An optimization lesson shows the learner more efficient ways to solve the same class of problems that it could solve before the lesson. Furthermore, a normal lesson may not introduce optimized methods, wherein some of the procedure's calculations are performed internally and do not appear in the action sequence.

These two conventions are not arbitrary. They directly address two of the worst combinatorial problems in induction. The first convention allows the learner to solve the *disjunction problem*, which involves deciding when and where to place disjunctions. (The disjunction problem will be discussed at length in Section 3.) For procedures, a disjunction is a branching (e.g., a COND) in the flow of control. Convention (1), above, informs the learner that there will be at most one new disjunct per lesson. This cuts down the possible places for disjunctions to a finite, small set and thereby significantly reducing the learner's search space of possible procedures. This makes the learner's job much easier. In fact, it makes it possible as opposed to impossible. The first convention is named *one-disjunct-per-lesson*.

The second convention fulfills a similar function with regard to a second combinatorial problem, the *invisible objects* problem. If two visible objects in a state can be related by arbitarily long chains of calculations with arbitarily many intermediate results, then induction is combinatorially infeasible. The intermediate results are invisible objects because they don't appear in the examples. If they could be seen, the combinatorics would be substantially reduced. The second convention provides this by mandating that normal lessons explicate such chains by showing all the intermediate results. Optimiza-

tion lessons may come along later and show how to suppress the intermediate results and perform the calculations "in one's head." The convention is called *show-work*.

To evaluate the efficacy of this approach, or any approach to learning from material prepared by a teacher, one must evaluate burdens placed on *both* the teacher and the learner. One would expect there to be some work required of each, because learning-from-lesson-sequences lies halfway between induction, where the learner does most of the work, and learning-by-being-told, where the teacher does most of the work.

The teacher's job is to generate a lesson sequence for a given target procedure that satisfies one-disjunct-per-lesson and show-work. This task can be accomplished mechanically if the teacher writes down the target procedure in an appropriate procedural language, e.g., LISP. But writing procedures can be quite a bit of work. A more interesting possibility is that experienced teachers generate such lesson sequences naturally, without even realizing that their lesson sequences obey the two conventions. This is exactly what my research on naturally occuring lesson sequences in mathematics shows [44, 45]. Educators tend to generate well-formed lesson sequences, even though they probably are not aware of the conventions. Apparently, they have an intuitive, unconscious appreciation of the conventions. This allows them to generate appropriate lesson sequences without going through the work of explicating and formalizing the procedures taught by those curricula.

This raises an interesting possibility for applications where a computer system learns from a human user, such as programming-by-example systems (e.g., [8]) or learning apprentice systems (e.g., [30]). Such systems usually assume that there is no meaningful structure in the example sequence that is presented to the system. However, if the users view themselves as teaching the system, they may order their examples in certain ways. At the very least, they will present easy cases before hard cases. If we had a precise definition of the ordering criterion that users tend to employ, and if the learning system were designed to take advantage of these tacit constraints on the instructional material, then it could recover information that is latent in the sequential ordering. This latent information might allow it to converge faster and more reliably on the knowledge that the user is trying to teach it. One-disjunct-per-lesson and show-work are exactly such constraints, and they do make SIERRA a more effective learner. Further research is needed to see how domain-general they are and to see whether there are other constraints like them.

It might seem strange that teachers should obey conventions like one-disjunct-per-lesson and show-work without being aware of them. Looked at in a different way, it would seem strange if they didn't. The teacher-learner situation is an extended communication act. We know that people naturally, unconsciously obey many conventions on natural language communication acts (see, e.g., [37]). It seems entirely likely that the teacher-learner discourse

should also be formatted by conventions. In the hope that this analogy is approximately correct, the conventions that govern learning will be named *felicity conditions*, an early name for certain natural, unconsciously followed language conventions [4].

The point of felicity conditions is to make the learner's job easier without burdening the teacher too much. One-disjunct-per-lesson and show-work make SIERRA's job rather simple, although not trivial. The following is a quick sketch of how it works. The details will be presented later.

1.3. How SIERRA works

SIERRA's algorithm is called *learning by completing explanations*. It begins by trying to parse an action sequence, using the procedure as if it were a grammar, but a grammar with data flow, conditional tests, etc. If parsing succeeds, the resulting parse tree is a trace (i.e., subroutine-calling hierarchy). Looked at in a different way, the parse tree constitutes an "explanation" for the action sequence. For instance, a partial explanation for an individual action in a sequence can be read off the parse tree by walking upwards from the action, i.e., the action was done in order to satisfy the goals of the subprocedure that called it (which is the next node above it in the parse tree), and the caller was executed in order to satisfy its caller, and so on. A complete explanation for an action involves taking into account data flows and side-effects, so explicit links for these effects are included in SIERRA's parse trees. Such links are analogous to the causal links that thread through the hierarchical structures of explanations of, e.g., kidnapping stories [10]. So explaining an action sequence is just parsing it. Such parsing is a form of plan recognition (e.g., [16, 20, 36]). This is a form of *explanation-based learning* [10, 11, 14, 32, 38, 39].

The conditional tests and data flows of a procedure are used to guide SIERRA's parser, significantly narrowing its search for a parse tree. However, the parser may choose to relax such tests or ignore them entirely. This may allow it to find a parse tree when it could not do so otherwise. If so, then a simple form of learning can be performed. The relaxations made by the parser are editted into the procedure. For instance, if certain predicates in a conditional test must be ignored by the parser, then SIERRA removes them from the conditional. This generalizes the condition test. Now if the parser redoes the narrow search, obeying the constraints imposed by conditional tests, etc., it will find the parse tree. SIERRA has generalized the procedure, allowing it to explain examples that it could not explain before. This learning technique is similar to one form of *explanation-based learning* [10, 11, 14, 32, 39].

A more interesting kind of learning occurs when it is impossible to complete a parse, no matter how much the procedure is generalized. In this case, the learner's procedure is fundamentally incomplete. One or more new subproce-

dures must be invented. Learning by *completing* explanations is one approach to accomplishing the learning required in this situation.

SIERRA uses a straightforward technique. A similar approach was employed in three independent investigations [16, 18, 43]. The first step is to parse the action sequence bottom-up as far as possible and top-down as far as possible. The candidate solutions to the learning task consist of any new subprocedure (or set of new subprocedures) that links the top-down parse to the bottom-up parse in such a way that a complete parse tree is yielded. Even for short action sequences, there can be millions of candidates. The challenge is to cope with this large space of candidates. The solution used by the three independent investigators [16, 18, 43] is to place such strong constraints on the parsing that only one (or a few) of the possible candidates are generated. My solution is to (1) use unconstrained parsing, (2) assume that only one new subprocedure will be acquired, and (3) use a factored data structure (similar to LUNAR's well-formed substring table [51] and GSP's chart [21]) to efficiently represent the space of possible candidates. The selection of a candidate from this space is accomplished by a collection of simple filters.

This technique for learning by completing explanations makes it simple to perform induction across several action sequences. Each action sequence yields a space of candidate solutions represented as a GSP-style chart. Induction amounts to intersection of these spaces.

In principle, this technique can be used in any domain which learns hierarchical knowledge structures from sequential examples. Thus, it should extend to learning grammars from strings, learning story-understanding schemata from stories, and learning device models from the operation of machines.

Many of the basic intuitions behind SIERRA have been presented. The remainder of this article presents the details of how SIERRA accomplishes its task. The first section presents the knowledge representation language used for procedures. The next section reviews the induction problem, and isolates disjunction as a key difficulty. One-disjunct-per-lesson is proposed as a central simplifying constraint for inducing procedures. It in turn leads to the definition of "subprocedure," in Section 4, in terms of three parts: its skeleton, its patterns, and its functions. This reduces the problem of inducing a subprocedure to three subproblems, one for each part. The subsequent sections discuss the SIERRA algorithms, for, respectively, skeleton induction, pattern induction, and function induction. The final sections discuss the generality of SIERRA and speculate on the origins and applications of felicity conditions.

2. The Representation of Procedures

It is convenient to use a mixture of nomenclature from production systems and

AND-OR graphs (AOGs).[4] The production system nomenclature is good for showing details, the AOG view is good for showing the overall structure.

Figure 3(a) sketches the AOG view of a subtraction procedure learned by SIERRA. The nodes are called *goals*, and links are called *rules*. Rules are directed and are always drawn running downward. The nodes just beneath a goal are called its *subgoals*. Currently, there are two types of goals: AND and OR.[5] To execute an AND goal, all the subgoals are executed. To execute an OR goal, just one of the subgoals is executed. AND goals are drawn with boxes around their labels. Drawings of AOGs abbreviate goals whenever they appear more than once. For instance, OVRWRT is called from several places in the AOG of Fig. 3(a), but its subgoals are drawn only for one of these occurrences. Although abbreviation makes this AOG look like a tree, it is really a cyclic directed graph due to the recursive calls of MULTI and REGROUP.

AOG drawings do not indicate several kinds of information. The information is readily visible in the production system view. Figure 3(b) shows the definitions for the nonprimitive goals in the AOG of Fig. 3(a). Goals have arguments. For instance, SUBICOL has three arguments, T, B and A. Arguments have the substitution semantics of lambda calculus. That is, the AOG language is *applicative*. There are no assignment statements. The only side-effect operators are those that change the external state, i.e., writing a digit in an answer. The applicative property has important consequences that will be discussed later.

A goal's rules (i.e., the rules leading from it to its subgoals) are listed in its definition. SUBICOL has three rules. Each rule has a pattern and an action. Patterns are large, so in Fig. 3(b) most patterns have been replaced by English glosses. A pattern is a conjunction of literals (i.e., predicates or negated predicates). Predicate arguments may be either arguments from the enclosing goal or pattern variables. As an example, the pattern

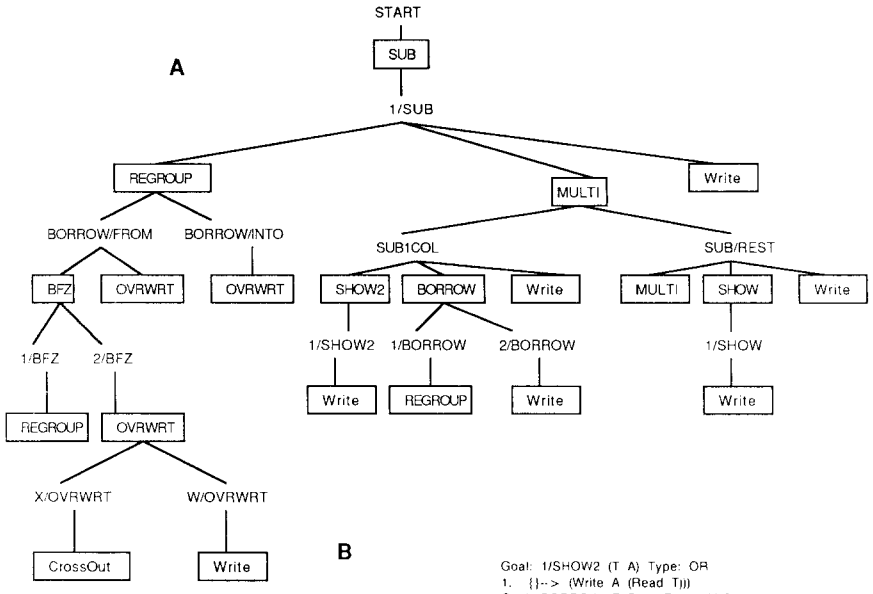
(Column C)&(Top C T)&(Digit T)&
(Bottom C B)&(Digit B)&(LessThan T B)

matches columns that require borrowing. The empty pattern always matches.

A rule's action is a form, in the LISP sense, which calls the rule's subgoal. The action may pass arguments to the subgoal, often by evaluating functions.

[4] It is well known that context-free grammars, push-down automata, and basic transition nets are equivalent. In the same way, attribute grammars [22], affix grammars [23] and ATNs are equivalent, provided that side-effect operators (e.g., SETQ) are not used in the ATNs. The latter equivalence class of representations includes the one used by SIERRA. Although any of the formalisms could be used to describe SIERRA's representation, a mixture of production systems and AOG is used here.

[5] In order to accommodate certain empirical data, a third goal type FOREACH, will be added as a representation for iteration across a sequence of objects. This goal type appears in Fig. 2, P1, as a loop in the ATN's graph. Currently, SIERRA represents iteration with tail recursions.




```
Goal: START(P) Type: OR
1.  {}--> (SUB P)
Goal: SUB(P) Type: AND
1.  Let T, B and A be top, bottom and answer of then
        rightmost column of problem P--> (1/SUB T B A)
Goal:  1/SUB (T B A) Type: OR
1.  Regrouping problem format --> (REGROUP T)
2.  There is a column to the left of T --> (MULTI T B A)
3.  {}--> (Write A (Sub (Read T)(Read B)))
Goal: MULTI (T D A) Type: AND
1.  {}--> (SUB1COL T B A)
2.  Let NT, NB & NA be the top, bottom & answer
        of the left adjacent column to T
        --> (SUB/REST NT NB NA)
Goal: SUB/REST (T B A) Type: OR
1.  There is a column to the left of T --> (MULTI T B A)
2.  B is blank --> (SHOW T A)
3.  {} --> (Write A (Sub (Read T)(Read B)))
Goal: SHOW (T B A) Type: AND
1.  {}--> (1/SHOW T A)
Goal: 1/SHOW (T B A) Type: OR
1.  {} --> (Write A (Read T))
Goal: SUB1COL (T B A) Type: OR
1.  B is blank --> (SHOW2 T A)
2.  T<B --> (BORROW T B A)
3.  {}-->(Write A (Sub (Read T)(Read B)))
Goal: SHOW2 (T B A) Type: AND
1.  {}-->(1/SHOW2 T A)
```

```
Goal: 1/SHOW2 (T A) Type: OR
1.  {}--> (Write A (Read T)))
Goal: BORROW (T B A) Type: AND
1.  {}--> (1/BORROW T)
2.  {}--> (2/BORROW T B A)
Goal: 1/BORROW (T) Type: OR
1.  {}--> (REGROUP T)
Goal: 2/BORROW (T B A) Type: OR
1.  {}--> (Write A (Sub (Read T)(Read B)))
Goal: REGROUP (T) Type: AND
1.  Let NT be the top digit of the left-
        adjacent column to T --> (BORROW/FROM NT)
2.  {}--> (BORROW/INTO T)
Goal: BORROW/INTO (T) Type: OR
1.  {}--> (OVRWRT T (Concat (One) (Read T)))
Goal: BORROW/FROM (TD) Type: OR
1.  TD is zero --> (BFZ TD)
2.  {}--> (OVRWRT TD (Sub1 (Read TD)))
Goal: BFZ (TD) Type: AND
1.  {}-->(1/BFZ TD)
2.  {}-->(2/BFZ TD)
Goal: 1/BFZ (TD) Type: OR
1.  {}--> (REGROUP TD)
Goal: 2/BFZ (TD) Type: OR
1.  {}--> (OVRWRT TD (Sub1 (Read TD)))
Goal: OVRWRT (D B) Type: AND
1.  {}--> (X/OVRWRT D)
2.  Let X be the blank space above D
        --> (W/OVRWRT X N)
Goal: X/OVRWRT (D) Type: OR
1.  {}--> (CrossOut D)
Goal: W/OVRWRT (X D) Type: OR
1.  {}--> (Write X D)
```

FIG. 3. A subtraction procedure shown as (a) an AOG, and (b) a production system.

For instance, SUB1COL's third rule has (Write A (Sub (Read T) (Read B))) as its action. This action writes the difference of the top and bottom digits of a column in the column's answer.

An OR goal's rules are tested in left-to-right order. The first rule whose pattern matches is executed. The learner adds new rules at the left. Hence, the left-to-right ordering convention corresponds to a common conflict resolution

strategy in production systems called "recency in long term memory" [26]. Because the patterns of OR rules test whether to execute a rule, they are called *test patterns*. Although AND rule patterns have the same syntax as OR rule patterns, they are not used to control which rules are executed. The order of execution of AND rules is fixed: the rules are executed in left-to-right order. AND rule patterns are used to retrieve information in the current problem state so that the information can be passed to the rule's subgoal. AND rule patterns are called *fetch patterns*.

Any learning model that describes how knowledge is constructed from smaller units is open to questioning about its set of primitives: what are the units that are assumed to be present when learning begins? For completeness, Table 1 lists the kinds of primitives used by SIERRA, and the particular ones employed to learn the procedures discussed in this article. In addition to these primitives, the initial knowledge state may contain nonprimitive procedures as well. For instance, the initial procedure from which the procedure of Fig. 3 was learned contained the nonprimitive goal OVRWRT, which crosses out a symbol and writes another symbol over it. The multiplication procedure's initial knowledge state included an addition procedure.

The procedural representation language has been presented. The remainder of this section is a "walk through" of the procedure of Fig. 3, which some readers may find helpful as a way of cementing their understanding of the representation.

The root goal, START, and its subgoal, SUB, initialize column traversal to start with the units column. I/SUB chooses between three subgoals. MULTI is for multiple-column problems. REGROUP is for "regrouping" exercises that don't involve any subtraction at all. Regrouping is the part of borrowing where one digit is reduced by one and an adjacent digit is increased by ten. This subgoal is left over from learning regrouping separately from multi-column subtraction. (The lesson sequence of Fig. 1 lacks a regrouping lesson, but most textbooks include one. This procedure was learned from Heath's subtraction

TABLE 1. The four kinds of primitives used by SIERRA

1. Primitive actions cause a change in the current problem state. The only primitive actions used in mathematics are the ones that write a given alphanumeric symbol at a given position (Write), or ones that write special kinds of symbols (CrossOut puts a slash over a symbol; Bar writes a bar under a group of symbols).
2. Fact functions return a number without changing the problem state. The following fact functions were employed: Add, Sub, Add1, Sub1, Mult, Quotient, Remainder, One (which always returns 1), Zero (which returns 0), and Concat (which concatenates two numbers, e.g., (Concat 1 4) returns 14).
3. Fact predicates return true or false without changing the problem state. The fact predicates used were: LessThan?, Equal?, and Divisible?.
4. The primitive function, Read, returns the symbol written at a given place.

lesson sequence [13], which has a separate regrouping lesson.) Normally, 1/SUB never calls REGROUP. The third goal, Write, is for single-column subtraction problems. The "main loop" of multi-column traversal is expressed by MULTI as a tail recursion. MULTI calls itself via its subgoals SUB/REST. SUB1COL processes a column. It chooses between three methods for doing so. If the bottom of the column is blank, it copies the top of the column into the answer via the subgoal SHOW2. If the top digit of the column is less than the bottom, it calls BORROW. Otherwise, it writes the difference of the two digits in the answer. BORROW has two subgoals: 1/BORROW calls REGROUP, and 2/BORROW just takes the difference in the column and writes it in the answer. REGROUP is a conjunction of borrowing into the column that originates the borrow (BORROW/INTO) and borrowing from the adjacent column (BORROW/FROM). In this procedure, BORROW/FROM occurs before BORROW/INTO. It would be equally correct to reverse their order, but that is not the way that Heath teaches them. Borrowing into a digit is just adding ten to it. Borrowing from the next column is also easy when its top digit is nonzero: the digit is decremented. If the digit is zero, it calls BFZ. BFZ regroups, which causes the zero to be changed to ten, then it decrements the ten to nine.

3. Disjunction: An Inherent Problem for Induction

For many kinds of induction tasks, there are proofs that the task has no algorithmic solution. In such proofs, the induction problem is defined by specifying a class U of all possible generalizations that the learner can output and a class T of all possible trainings that the learner can receive. (The "U" stands for the learner's universe of generalizations.) There are a variety of theorems for various U and T. For instance, one such theorem is: If U is the class of recursive functions and T is the set of all possible training sequences that contain all possible positive and negative examples, then there is no Turing machine that can learn any given generalizations from U [35, Proposition 5]. Such negative results guarantee that there is no straightforward solution to induction.

The standard attack is to incorporate *biases* into the inducer [28]. There are two kinds. An *absolute bias* is a unary predicate on generalizations that says whether or not the generalization should ever be output by the inducer. A *relative bias* is a binary predicate on generalizations that says which of the two generalizations is preferred for output in case both generalizations are consistent with the given training. Often, absolute biases are implemented by representing generalizations in a limited representation language. If the generalization cannot be expressed in the language (say, because the language lacks the appropriate primitives), then it will never be output by the inducer. A relative bias, on the other hand, is usually defined by comparing two formal expressions that represent the generalizations. Simplicity metrics are a common

relative bias. AI inducers often implement relative biases implicitly by the order in which they search. Because they stop when they get to the first generalization that is consistent with the training, the search strategies act as relative biases. In short, biases correspond to two obvious kinds of constraints: unary and binary predicates on generalizations.

A nonstandard approach is to employ a third kind of constraint, which could be called a *manner* constraint. A manner constraint relates a generalization to the manner in which the training is presented. A manner constraint is a binary predicate: one argument is a generalization and the other is the form (syntax) of the training. Both one-disjunct-per-lesson and show-work are manner constraints.[6]

Manner constraints are a known loophole to most formal learnability results. For instance, a major result [17] is that it is impossible to learn when (1) T employs only positive examples and (2) U contains a generalization for every finite set of examples and a generalization for at least one infinite set of examples, unless (3) the examples sequences in T are ordered by some primitive recursive function. That is, conveying information with the order of the example sequences allows a learner to succeed where it could not otherwise. The manner of example presentation is a factor that hasn't been studied much, but is potentially quite important.

There are dozens of theorems on the learnability of certain U given certain T. In order to obtain such results, it is necessary to be quite specific about what the U and T are. Rather than review all these specific results, it seems more profitable for this article to sacrifice formality in order to uncover the key components of Us and Ts that cause induction to be impossible. In spirit, this strategy is like Newell's knowledge level strategy of stepping back from the details of countless AI problems and representations in order to analyze them from a single perspective, which he suggests should be the perspective of first-order logic [34].

In that spirit, I suggest that one of the inherent problems of induction is disjunction. Whenever U allows a generalization to be built from the disjunction of any two other members of U, induction is infeasible.

More specifically, suppose that g and g' are two generalizations from U. Even without knowing how they are represented, we can define their disjunction. Each generalization has an extension, i.e., the set of all possible examples (instances) consistent with the generalization. Let x and x' be the extensions of g and g', respectively. The disjunction of g and g' is any generalization whose

[6] The terminology could use a little clarification here. Felicity conditions are defined as interpretive constraints on skill acquisition that people obey without being aware of them. Presumably, one could explicitly tell students a manner constraint, in which case it wouldn't qualify as a felicity condition. Presumably, there could be felicity conditions that aren't manner constraints, but, say, relative biases.

extension is the union of x and x'. This is the definition of "disjunction" that will be used in this article.

Disjunctions often correspond to syntactical constructions in representational language. In AOG representations, disjunctions correspond to OR goals. In context-free grammars, a disjunction is present when two or more rules reduce the same nonterminal category. In production systems, a production is potentially disjunctively related to all the other productions; it requires careful analysis to uncover the actual disjunctive relationships.

Induction's trouble occurs when the class of all possible generalizations admits free disjunction. That is, the disjunction of g with g' is in the class whenever g and g' are. When this is the case, induction acquires some strange properties that make it seem quite unlike anything that one would want to call "learning."

Free use of disjunction allows the learner to generate absurdly specific generalizations. One such absurdity is the *trivially specific* generalization: a disjunction whose disjuncts are exactly the positive examples that the learner has received. Thus, if the learner received positive examples a, b and c, then the disjunction (OR a b c) is the trivially specific generalization. The trivially specific generalization is not really a generalization at all. Its extension is just $\{a\ b\ c\}$. The learner didn't really learn, it just remembered. This problem could be solved with an arbitrary prohibition against trivially specific generalizations, if nothing else. But there is another problem that is much worse.

When disjunctions are unconstrained, the learner has to be given the complete extension of the generalization being taught before it can reliably discriminate that generalization from the others. To see this, first assume that for each example, there is a generalization in U whose extension is that example and only that example. This is equivalent to assuming that the examples can be represented in the representation language used for generalizations. For instance, a grammar consisting only of the rule $S \rightarrow w$ is such a generalization, where S is the root category and w is a string of terminals. This grammar's extension is the singleton set $\{w\}$. Using such singleton generalizations and disjunction, any finite set of examples can be described by some generalization. To get the generalization for $\{w_1, w_2\}$, one finds the generalization for $\{w_1\}$ and $\{w_2\}$, and then forms their disjunction. Since all finite sets of examples correspond to generalizations, the learner cannot tell which generalization is correct until it is told exactly what the target generalization's extension is. This means it must be shown all possible examples and be told which are positive examples and which are negative examples. Such conditions, where a "learner" is shown a complete extension of a generalization and asked to identify the generalization, hardly qualify as learning.[7]

[7] In particular, the famous Gold [17] result that any grammar can be "learned" if the learner receives negative examples as well as positive examples does not seem like learning to me, for the learner must be guaranteed to receive all possible examples.

Suppose learning is viewed as the following rather naive search for generalizations. This will provide another perspective on the trouble that disjunction causes. Suppose that the learner receives an initial example and generates a single generalization. Suppose further that the next example is a positive one that the learner's current generalization is not consistent with. The learner has two choices: (1) to modify the current generalization enough so that it becomes consistent with the new example, or (2) to create a generalization specifically for the new example, then disjoin that generalization with the current one. Roughly speaking, these two choices are available at every step, so after N examples, there will be roughly 2^N possible generalizations consistent with the examples. The point is simply that the set of consistent generalizations grows as the learner is given more positive examples. It doesn't shrink, as one would intuitively expect of learning from examples. On the other hand, if disjunction is barred from U, then there is only one choice at each step, and the set of generalizations does not grow unboundedly. So this learner's failure to learn can be blamed squarely on disjunction.

One-disjunct-per-lesson would constrain this naive learner's search while allowing generalizations to contain disjunctions. Most of the time, the learner would have a single choice. However, on the first example of each lesson, it would have two choices. It can either disjoin or not. If it chooses to disjoin, then it may not disjoin again until the next lesson. If it chooses not to disjoin, then the twofold choice is again available on the next example. One-disjunct-per-lesson is indeed a straightforward solution to the disjunction problem.

One-disjunct-per-lesson solves the disjunction problem by modifying the relationship between T and U. Absolute and relative biases solve the disjunction problem by modifying U. A relative bias partially orders the elements of U by preferring, e.g., generalizations with fewer disjunctions. An absolute bias removes from U all generalizations that contain disjunctions (or contain more than 13 disjunctions, etc.). In general, biases modify U, and manner constraints modify the relationship between T and U.

Biases are appropriate for a learning task when the learner can make strong a priori assumptions about U. Manner constraints are appropriate when the learner cannot make assumptions about what's going to be learned, but it can make assumptions about how its going to be taught. For SIERRA's task domain, it is inappropriate to use biases to solve the disjunction problem. There is no reason for the learner to believe that a procedure should have no conditionals (or less than 13 conditionals), so an absolute bias against disjunction is inappropriate. There is no reason for the learner to believe that a procedure with the fewer conditionals is better, so a relative bias is also inappropriate. However, given the felicity conditions hypothesis, there is reason to believe that T and U are related, so a manner constraint such as one-disjunct-per-lesson is appropriate.

The preceding comments were meant to motivate the pragmatic utility of manner constraints in general, and one-disjunct-per-lesson in particular, by

considering how one-disjunct-per-lesson helps solve the disjunction problem, one of the inherent problems of induction. Later, a similar motivation will be presented for the show-work manner constraint, based on another inherent problem of induction, the invisible objects problem. Manner constraints are a new technique for solving inherent problems in induction. Previous AI learners have employed either absolute biases or relative biases. Although any addition to AI's toolkit of techniques is welcome, manner constraints seem particularly welcome, for they are remarkably general, as the preceding discussion argued, and they are quite effective in reducing the complexity of programs for learning procedures, as the remainder of this article shows.

4. Subprocedures

In order to make one-disjunct-per-lesson easy to implement, the AOG representation permits disjunctions in only one place: the OR goals. Thus, a disjunct is an OR goal's rule plus, roughly speaking, whatever that rule calls. Such fragments of AOGs are called *subprocedures*. A subprocedure consists of several components:

(1) A new OR rule that is placed beneath an existing OR goal. The existing OR goal is called the *parent*.

(2) A new AND goal, which is called by the new OR rule.

(3) The new AND has one or more rules. Each rule calls a new OR goal that has just one rule. These ORs are merely a convenience. They provide a place for later subprocedures to attach.

(4) Each such OR has a single rule that calls some existing AND goal. These existing AND goals are called *kids*.

Figure 4 illustrates these components of a subprocedure by showing an AOG before and after a subprocedure has been added. This subprocedure was acquired from a lesson that teaches how to borrow across zeros. The pre-lesson AOG (Fig. 4(a)) can borrow only from nonzero digits; the post-lesson AOG (Fig. 4(b), which is the same as Fig. 3) can borrow across zeros. BORROW/FROM is the subprocedure's parent. The new OR rule connects BORROW/FROM to BFZ. The new AND is BFZ. The kids are REGROUP and OVRWRT. 1/BFZ and 2/BFZ are the new OR goals that are added as places to attach future subprocedures.

One-disjunct-per-lesson takes us a long way towards solving the whole procedure induction problem. Inducing a procedure is reduced to a series of subprocedure induction problems, one per lesson. A subprocedure induction problem reduces to three subproblems:

– *Skeletons*: Skeleton induction determines the parent and the kids of the new subprocedure. This establishes the topology of the new subprocedure (i.e., the connectivity of the goals and rules). Because it doesn't determine the conditions and action arguments of the new rules, skeleton induction is like inducing only the bones and not the flesh of the subprocedure.

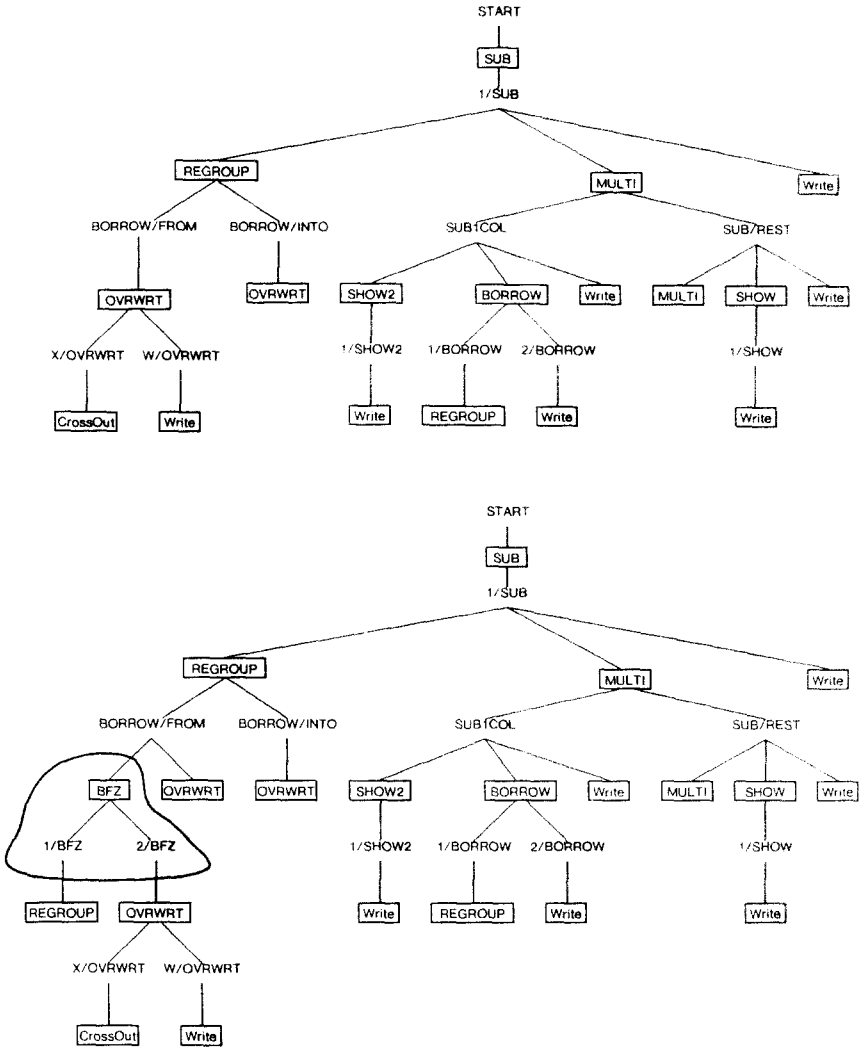


FIG. 4. An AOG before (a) and after (b) a lesson. The new subprocedure is circled.

– *Patterns*: One-disjunct-per-lesson entails that a rule's conditions have no disjunctions. This means that they can be induced by standard, disjunction-free pattern induction techniques.

– *Functions*: One-disjunct-per-lesson entails that disjunctions are not permitted in the nests of functions that express the data flow in AOGs. This makes inducing those function nests easier. However, Section 7 shows that function nest induction is still infeasible unless more constraints are added. The show-work felicity condition is the key to SIERRA's solution.

These three induction tasks will be discussed serially in the following sections. In SIERRA, skeleton induction happens first using one pass over the lesson's examples. Pattern and function induction occur together on a second pass over the examples.

5. Skeleton Induction

To see what skeleton induction involves, a computer science fixture is needed: the trace of a procedure's execution. Figure 5 shows the trace tree for a correct

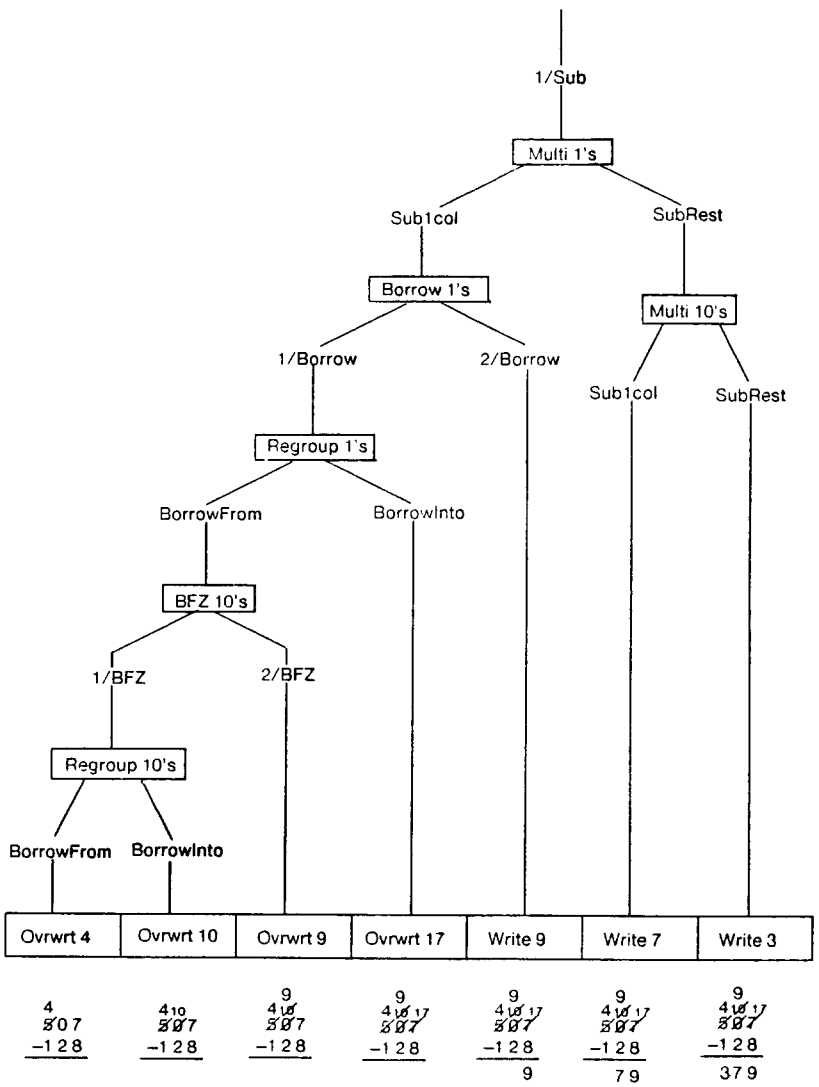


FIG. 5. Trace tree for solution of a BFZ problem.

subtraction procedure (the procedure is shown in Figs. 3 and 4(b)) solving a BFZ (i.e., borrow from zero) problem. Each call is shown as a tree node, with its arguments abbreviated. A trace tree is just a parse tree for the action sequence, using the procedure as the grammer.

Roughly speaking, a skeleton is a hole in a trace tree. If the procedure is missing the BFZ goal, then the trace tree would have a hole in the middle of it, as in Fig. 6. The gap is right where the BFZ node would be. From the figure,

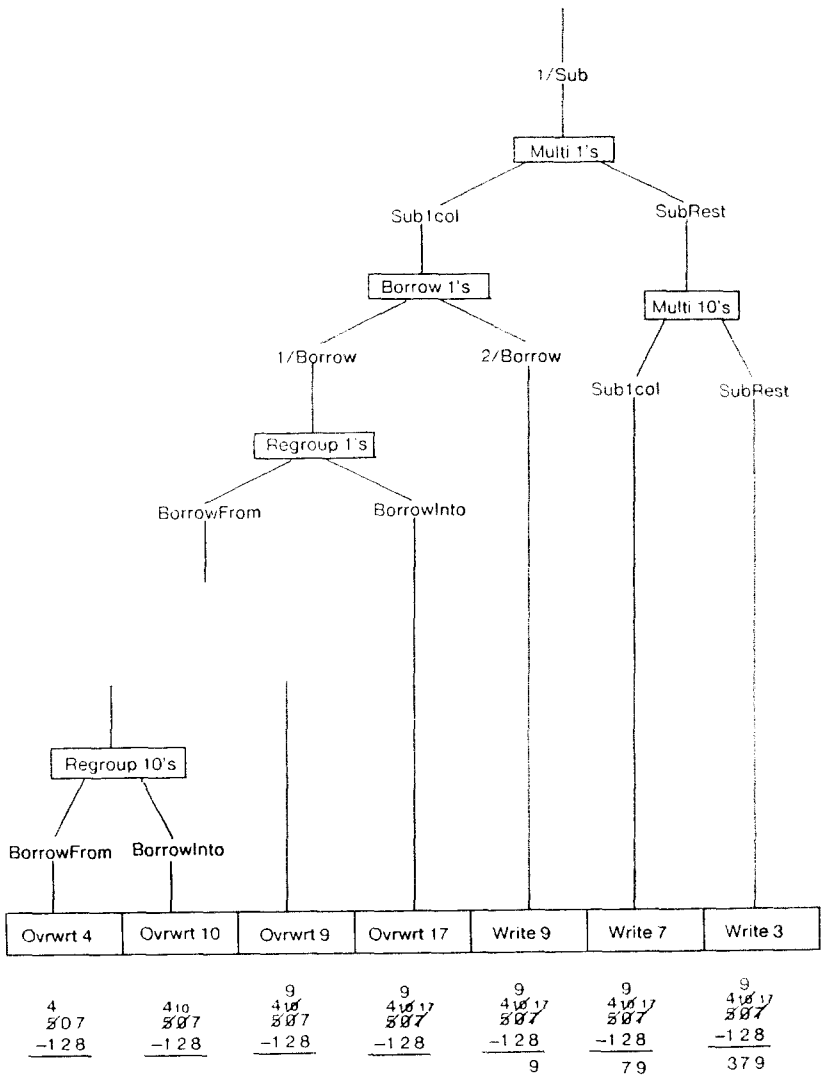


FIG. 6. The skeleton is right where the BFZ node and its daughters were.

one can see that a skeleton can be characterized by the link coming into it from above and the links leaving it from below. Thus, a skeleton is uniquely specified by the parent and the kids.

Almost all action sequences, including the example of Fig. 6, admit more than one skeleton. Most of the ambiguity is due to the fact that one can almost always make a skeleton bigger. The kids can be lower in the tree (e.g., Fig. 7);

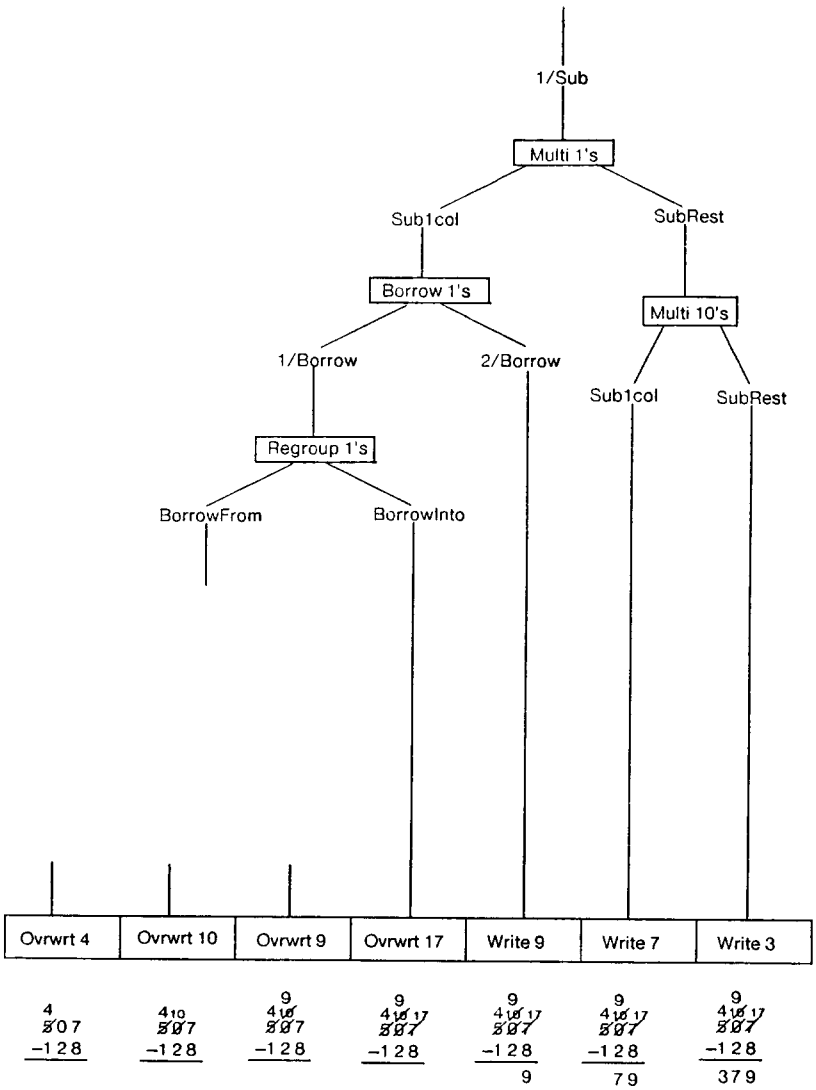


FIG. 7. The kids of the skeleton can be lower.

the parent can be higher (e.g., Fig. 8). Any node that would complete an otherwise incomplete trace tree is a legitimate skeleton.

SIERRA uses two context-free grammar parsing algorithms to enumerate the skeletons. A top-down, recursive descent parser is used to find all possible parents. (It is actually just a nondeterministic version of the regular AOG

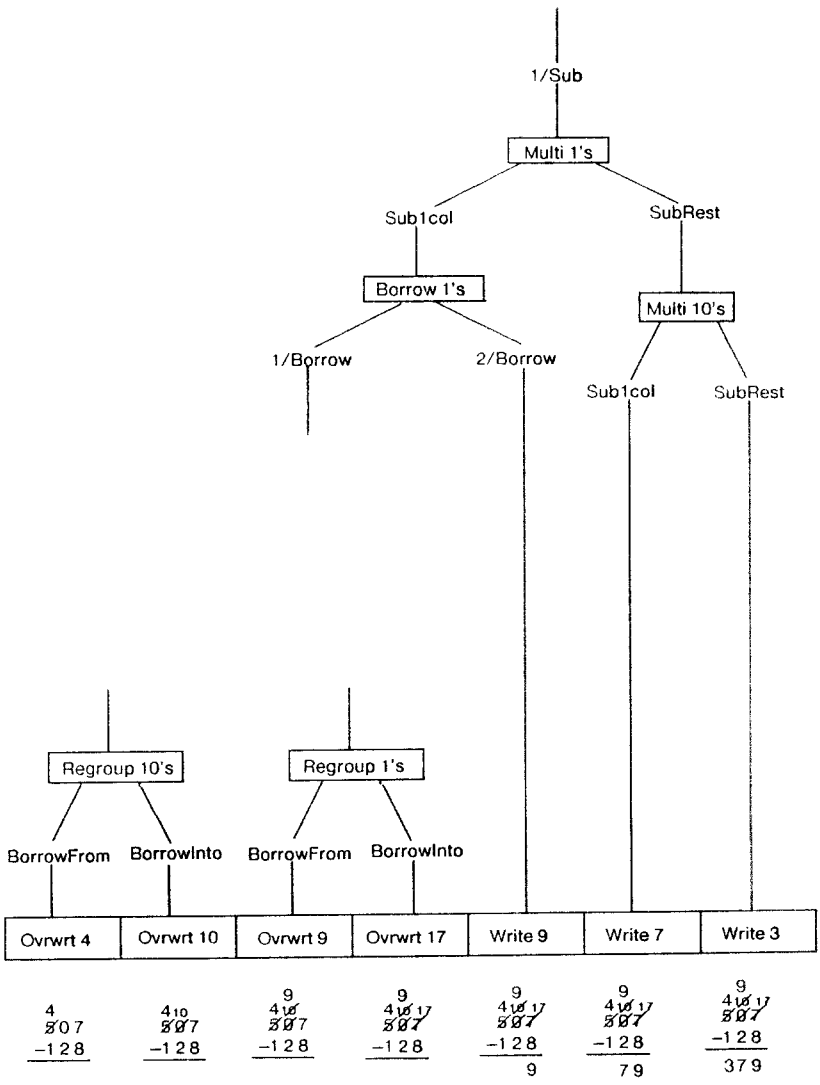


FIG. 8. The parent of the skeleton can be higher.

interpreter.) A bottom-up, breadth-first parser is used to find all possible kids. For each parent, all possible tuples of kids are collected, where a kid tuple is a sequence of adjacent kids that together span the same part of the action sequence as the parent. This generates the set of all possible skeletons. In general, there can be thousands of possible skeletons (e.g., (50 possible parents) × (3 kids per kid tuple, on average) × (30 possible kids for each tuple position on average) = 4500 possible skeletons). SIERRA represents this set implicitly, in order to save space.

Notice that there would be many more possibilities if it weren't for one-disjunct-per-lesson. For instance, if two new subprocedures were allowed, then SIERRA would have to collect all possible pairs of parents, each with their possible kids, etc.

Parsing SIERRA's AOGs is equivalent to parsing attribute grammars, for which there are many algorithms [46]. Such parsing is quite simple because the language is applicative (i.e., no side-effects, no assignment statements). Once a goal's arguments are bound by its caller, those values never change. The arguments act as a subcategorization of the goal. This makes parsing nearly as simple as context-free grammar parsing. If side-effects were allowed, parsing would still be possible, but it would be combinatorially costly, because the left (preceding) context of a goal would have to be included in the goal's subcategorization.

Bottom-up parsing requires inverse execution of AOG code. SIERRA must be able to figure out the arguments of a caller from the arguments of its callee's. This requires matching fetch patterns "backwards" and inverse evaluation of functions. The first is easy. Matching patterns backwards is the same as matching them forwards. Backwards evaluation of arithmetic functions, such as (Add *x y*), is accomplished by hand-coded inverse functions that produce sets of tuples that represent possible input values. This technique would collapse if it were not feasible to assume that examples only use small numbers. If numbers could be arbitrarily large, then other techniques (e.g., symbolic execution, followed by solution of a system of polynominal equations) would have to be employed.

The techniques just mentioned generate one set of candidate skeletons per action sequence. One-disjunct-per-lesson entails that skeleton induction can be performed simply by intersecting these sets. Because a lesson may introduce just one subprocedure, all the skeletons' parents must be the same. Because the new subprocedure is disjunction-free, each skeleton's list of kids must be equal to each other skeleton's list of kids. In particular, two kid tuples (A B C) and (A D C) cannot be merged by using disjunction on the middle kid to form something like (A (OR B D) C).

Skeleton intersection is powerful enough that it is usually possible to devise a lesson that yields a unique skeleton when its examples' skeletons are intersected. However, in some cases, this is not possible. In fact, the BFZ skeleton that

is our running illustration cannot be uniquely specified by examples.[5] In such cases, there is no choice but to guess. This means that relative biases for skeletons are required.

SIERRA has been used to test various relative biases in order to find the ones that explain the skeleton choices that people make. SIERRA first generates all consistent skeletons using parsing and skeleton intersection. In manual mode, it displays them in a menu and allows the user to choose one. This is useful for exploration. In automatic mode, SIERRA partially orders the skeletons using the biases that are being tested. Usually there is just one skeleton that is maximal in the partial order. If so, SIERRA just takes it and goes on. If there is more than one, SIERRA chooses the first one, then stores the learner's state so that it can come back later and take the other choices. There is nothing new about this architecture, but it is remarkable how easy it makes it to search the space of hypotheses. The current best hypothesis is that people choose the smallest, most deeply embedded skeleton. (See [44, Chapters 18 and 19], for a complete discussion.)

5.1. Prior solutions to the skeleton induction problem

Skeleton induction determines the procedure's goal/subgoal-calling hierarchy. Inducing such hierarchies has proved to be a tricky problem in machine learning. Neves [33] used hierarchical examples to get his procedure learner to build hierarchy. However, subtraction teachers rarely use such examples. Badre [5] recovered hierarchy by assuming examples are accompanied by a written commentary. Each instance of the same goal is assumed to be accompanied by the same verb (e.g., "borrow"). This is a somewhat better approximation to the kind of input that students actually receive, but again it rests on delicate and often violated assumptions. Anzai and Simon [3] used production compounding (chunking) to build hierarchy. However, to account for which of many hierarchies would be learned, Anzai used domain-specific features, such as the pyramids characteristic of subgoal states in the Tower of Hanoi puzzle. SIERRA's technique, learning by completing explanations, is less domain-specific than Anzai and Simon's technique, and requires less information from the teacher than Neves' and Badre's technique.

6. Pattern Induction

The rules in the new subprocedure must be given appropriate conditions. The

[5] The proof relies on the distribution of branches in the AOG that is input to the learner. The AOG is shown in Fig. 4(a). If the examples are designed to exemplify BEZ, then they will always allow the skeleton to have BORROW/FROM as parent. However, because BORROW/FROM is the first subgoal of REGROUP, a second skeleton is always possible. It is a skeleton whose parent is 1-BORROW. This skeleton is shown in Fig. 8. It will always be legal no matter what the example. In general, when all possible paths from the root to the target parent pass through some OR that is not the target parent, then skeleton intersection can't eliminate the latter OR as a possible parent. In such cases, the intersection of skeletons over all examples will always have both skeletons.

new OR rules require test patterns, and the new AND rules require fetch patterns. Because patterns have no disjunctions nor other representational devices that trouble induction, patterns can be induced using standard techniques. This section presents the ones that SIERRA uses. Although pattern induction is not particularly interesting from a theoretical standpoint, it has turned out to be the bottleneck in SIERRA's efficiency.

Parsing an action sequence with the new subprocedure installed will pair the new OR rule with a problem state where its test pattern would have to be true in order for the parse to go through. Such states are positive instances for the test pattern. Similarly, parsing can uncover states where the test pattern would have to be *false* in order for the parse to go through. These states are the negative instances. The test pattern induction problem is to find a test pattern that matches all the positive instances and none of the negative instances. Parsing also collects positive instances for the fetch patterns, together with the values that the fetch patterns should return. The fetch pattern induction problem is to find a pattern that matches all the positive instances and returns the appropriate values each time.

Both induction problems are solved using version spaces [29]. A version space is represented by a pair $\langle S, G \rangle$, where S is the set of maximally *specific* patterns consistent with the instances received so far, and G is the set of maximally *general* patterns consistent with the instances. To use version spaces, several application-specific functions must be defined. The most important two are Update-S and Update-G. Given a version space and a positive instance, Update-S generalizes the patterns in S so that they match the instance and remain maximally specific. To implement Update-S, SIERRA uses an algorithm that finds the largest common subgraph of two labelled directed graphs.[9] Given a version space and a negative instance, Update-G augments the patterns in G so that they do not match the negative instance. Update-G is implemented with an algorithm for generating minimal covers of a set.[10]

[9] SIERRA's patterns are conjunctions of literals, where a literal is a predicate or a negated predicate. Such patterns are similar to labelled directed graphs, where the variables are nodes and the relations are labelled arcs. If all the predicates in a pattern are binary, the correspondence is exact.

[10] Given a g from G that matches the negative instance N, and an s from S that doesn't match N, Update-G needs to find relations in s to add to g such that the revised g doesn't match N. Update-G first generates all possible mappings of the variables of s into the variables of N. Each such mapping becomes an element of the set of maps, M, to be covered. Each relation in s is paired with the subset of M that contains the maps under which the relation is *not* a member of N. The relation is said to "cover" that subset of M. A key fact is that a conjunction of relations covers at least the union of their individual covers of M. The main goal is to find the smallest conjunction of relations that covers all of M. Such a conjunction is not a member of N under any map, so adding it to g will yield a pattern that doesn't match N, which is what we want. So the algorithm for finding minimal covers of M yields candidates for the revised g. Some of these candidates may generalize others, so Update-G has one more task, which is to filter out the candidates that are not maximally general.

SIERRA is designed so that pattern induction never finishes. A rule keeps the version space of its pattern so that induction can continue whenever the pattern requires more refinement. It is almost always the case that introducing a new subprocedure will cause the patterns in older subprocedures to be modified. Those older patterns will be "seeing" problem states that they have never "seen" before, namely, the ones that trigger the new subprocedure and the ones that the new subprocedure produces. In order to continue to function properly, the older patterns must be generalized to match these new states. The generalization of older patterns to match new situations is a simple form of explanation-based learning, as the term is used by some authors (cf. [11]).

Since pattern induction occurs so often, it needs to be fairly efficient. However, the Update-S computation is NP-hard, and the Update-G routine calls it as a subroutine. More specifically, if pattern P has n variables, and it is matched against a problem state with m objects, then Update-S takes $O(m^n)$. These combinatorics reflect the usual AI matching problem: Each variable in P can be paired with any object in the problem state.

One way to deal with the complexity of pattern induction is to use a small n and/or m. For instance, blocks-world inducers (e.g., [12, 48]) typically have an n and m of less than 5. Using a small n and m is impossible in SIERRA's case. Its task domain requires problem states with 10 to 50 objects. The larger patterns in the version spaces have about the same number of variables as objects in the problem states they were induced from. The combinatorics for straightforward pattern inducer can go as high as 50^{50}.

A second solution is to impose constraints on which variable-object mapping will be considered. SIERRA uses two constraints. First, pattern variables have an implicit inequality relationship between them. That is, distinct variables must match distinct objects. This lowers the combinatorics to the binomial coefficient function, $O(n!/[(n-m)!m!])$. Second, the patterns and problem states are split into two components, a part-whole tree and the rest. In a problem state, the part-whole tree is simply the usual parse tree for the mathematical notation. For instance, a subtraction problem's parts are its columns and a column's parts are its digits. In patterns, there are variables for each of the components (i.e., a variable for the problem, for each column and for each digit), and their part-whole relationships are kept separately from the main pattern. Pattern induction (and pattern matching, too) considers only variable-object mappings that do not violate the tree topologies.[11] This cuts the complexity down to $O(B! \log_B n)$, where B is the branching factor of the part-whole trees, typically about three. When this constraint is turned off in SIERRA, an Update-S that normally takes 10 seconds takes hours. This con-

[11] If pattern variable x is paired with object o in a state, and (Part-of y x) is in the pattern (which means that y is a part of x), then y can only be paired with objects that are parts of o in the state. If there are no such objects, then y is left unpaired.

straint, or something like it, is a practical necessity. Even with it, most of SIERRA's time is spent running the pattern induction algorithms.[12] In a typical run, about 70% of the time is spent doing pattern induction.

7. Function Induction

Some of the rules in the new subprocedure may require function nests to be induced for their actions. Functions are used to represent number facts, such as (Sub 17 8) = 9. A typical function nest is (Sub (Add 7 10) 8). This section discusses how function nests can be learned. This learning task is called *function induction*. Function induction involves discovering which function or nest of functions will yield the numbers shown in the examples. For instance, suppose the learner already knows how to do single-column subtraction problems, and it is taking a lesson on two-column subtraction. After seeing examples (a) and (b),

(a)
$$\begin{array}{r} 72 \\ -41 \\ \hline 31 \end{array}$$

(b)
$$\begin{array}{r} 74 \\ -21 \\ \hline 53 \end{array}$$

there are many function nests that explain where the tens column answer comes from. Here are three candidates:

(1) $A_{10} = T_{10} - B_{10}$,
(2) $A_{10} = T_1 + B_1$,
(3) $A_{10} = ((T_{10} + T_1) - (B_{10} + B_1)) - A_1$,

where the subscripts indicate the column, and T, B and A stand for the top, bottom and answer. The first generalization is the correct one. The second generalization is that the ten's answer is the sum of the units column's digits. This second generalization, although consistent with examples (a) and (b), is inconsistent with (c):

(c)
$$\begin{array}{r} 36 \\ -12 \\ \hline 24 \end{array}$$

Many such accidental generalizations can be eliminated by giving lots of examples. However, generalization (3) can never be eliminated that way. It is true of any subtraction problem. This may seem like a peculiarity of this case,

[12] SIERRA uses a further trick. It avoids as much pattern matching and pattern induction as possible by parsing action sequences in two passes. The first pass ignores the subcategorization provided by goal arguments, and therefore does no pattern matching. The second pass takes the parse trees produced by the first pass and instantiates them by doing all the requisite pattern matching and function evaluation. Because some parse trees are pruned on the first pass, the pattern matching that they would have required is avoided.

but it isn't. There are infinitely many polynomials consistent with any finite set of input-output number tuples. (In particular, there are infinitely many n-degree polynomials consistent with any n points.)

The underlying problem has nothing to do with the fact that functions and polynomials are the representation language of generalization. Any functional expression can be easily converted to a relational one. For instance, generalization (3) above could be expressed as

(AND (PLUS X T_{10} T_1)
(PLUS Y B_{10} B_1)
(PLUS Z A_{10} A_1)
(MINUS Z X Y)
(INVISIBLE X)
(INVISIBLE Y)
(INVISIBLE Z))

where (PLUS u v w) means $u = v + w$. The special relation INVISIBLE is needed because X, Y, and Z do not match any of the visible objects in the examples. Under normal confirmation conventions for relational descriptions [19], the variables match only visible objects, so variables that designate invisible objects must be specially marked, and that is what INVISIBLE does.

Looked at in this way, the underlying induction problem is clear: if the representation allows invisible object designators, then there will always be far too many generalizations consistent with any finite set of examples. Some constraint must be placed on the use of invisible objects in examples. This induction problem will be called the *invisible* objects problem.

7.1. Prior solutions to the invisible objects problem

AI's most common solution to the invisible objects problem is to ban invisible object designators from the representation. For instance, Winston's blocks-world representation language [48] could have employed an elegant expression of the arch concept if it allowed invisible objects:

(AND (ISA LINTEL 'PRISM)
(ISA LEG1 'BRICK)
(ISA LEG2 'BRICK)
(ISA GAP 'BRICK)
(INVISIBLE GAP)
(SUPPORTS LEG1 LINTEL)
(SUPPORTS GAP LINTEL)
(SUPPORTS LEG2 LINTEL)
(ABUTTS LEG1 GAP)
(ABUTTS GAP LEG2))

This says that the lintel rests on three abutting bricks, and the middle one is invisible. Using a differently shaped invisible block for the gap is a simple way

to describe pyramidal, trapezoidal, and circular arches as well as the rectangular arch above. However, the invisible objects problem makes it impossible to induce such descriptions. Once invisible blocks are allowed, they could be anywhere. The inducer would have no way of knowing whether there was just one invisible block, the gap, or dozens lying around all jumbled up. Winston avoids the problem by omitting invisible object designators from the representation, and employing the relationship (NOT (TOUCHING LEG1 LEG2)) to express the gap between the arch's legs.

Banning invisible object designators is one way to solve the invisible objects problem. But it won't work in the mathematical domain. Invisible object designators are needed for representing procedures such as multi-addition. In the problem $1+3+5=9$, the intermediate result, either a 4, 8 or 6, is invisible.

As mentioned earlier, absolute and relative biases are the two customary ways to succeed at induction. An absolute bias, banning invisible objects, was just discussed. For a relative bias, the obvious candidate is to prefer generalizations with the fewest invisible objects. This is roughly what BACON3 does [24]. It induces physical laws given tables of idealized experimental data. For instance, it can induce the general law for ideal gases when it is given "experiments" such as this one:

```
(AND  (MOLES 1.0)
      (TEMPERATURE 300.0)
      (PRESSURE 300000.0)
      (VOLUME 0.008320))
```

This formal representation describes the experiment in the same way that Winston's representation described a scene in the blocks world (this is not the representation that BACON3 uses, by the way). The expression above says that there is one mole of gas at a certain temperature and pressure, occupying a certain volume. The goal of BACON3 is to find a description that is a generalization of the experiments that it is given. For experiments of this type, the generalization that it induces is:

```
(AND  (MOLES N)
      (TEMPERATURE T)
      (PRESSURE P)
      (VOLUME V)
      (TIMES X1 P V)
      (INVISIBLE X1)
      (TIMES X2 N T)
      (INVISIBLE X2)
      (QUOTIENT X3 X1 X2)
      (INVISIBLE X3)
      (CONSTANT X3))
```

That is, PV/NT is a constant. This is one way to express the ideal gas law, which

is more widely known as $pV = nRT$, where $R = 8.32$. The intermediate results PV, NT, and PV/NT do not appear in the "scene" described earlier. This is what makes BACON3's job hard. BACON3's method for solving this induction problem is, very roughly speaking, to guess useful invisible objects descriptors and enter their values in the scenes. It might start by forming all binary functions on the visible objects, e.g., NT, P + V, PP, P/T, etc. Since none of these yield values (invisible objects) that are constant across all the scenes, it tries further compositions: NT/PV, NT + V, NTPV, etc. At this level, it succeeds, since PV/NT turns out to be the same value, 8.32, in all the scenes. Essentially, BACON3 solves the invisible object problem by choosing a generalization with a minimal number of invisible object designators.

BACON3 is not an incremental inducer. It assumes that it has the total example set at the beginning. There is a reason for this. Any inducer that seeks a generalization with the fewest invisible object designators would clearly want to entertain generalization with $N + 1$ invisible object designators only after it had disconfirmed all the generalization with N invisible object designators. However, adding an invisible object designator to a disconfirmed generalization won't help it a bit. That is, if $f(g(x))$ doesn't match a certain example, then wrapping an $h(\cdot)$ around it won't help.[13] This means that failure at the level of N invisible objects doesn't tell one anything about what generalizations to use at the $(N + 1)$th level. If the inducer is incremental, and it is at the Nth level, and the Mth example exhausts the level, then the inducer must start over at the $(N + 1)$th level and re-examine all M examples. It would be better off just waiting until the teacher told it that all the examples were presented, then do a nonincremental induction. This would require a manner constraint. The teacher would have to mark the example presentation, and the learner would have to understand such marks as indicating that it was okay to begin nonincremental induction. As it turns out, naturally occuring mathematical curricula do employ a manner constraint, but it is not the one just mentioned. The next subsection describes the one that actually occurs.

7.2. The show-work felicity condition

In almost all cases, textbooks do not require the student to do invisible object induction. Instead, whenever the text needs to introduce a subskill that has a mentally held intermediate result, it uses two lessons. The first introduces the subskill using special, ad hoc notations to indicate the intermediate results. Figures 9 and 10 show some examples. Since the intermediate results are written out in the first lesson, the students need guess no invisible objects in

[13] Unless $h(\cdot)$ is the identity function almost everywhere, so that $h(f(g(x))) = f(g(x))$ except for the x that disconfirm $f(g(x))$. Most vocabularies of functions, including those of BACON3 and SIERRA, exclude such functions.

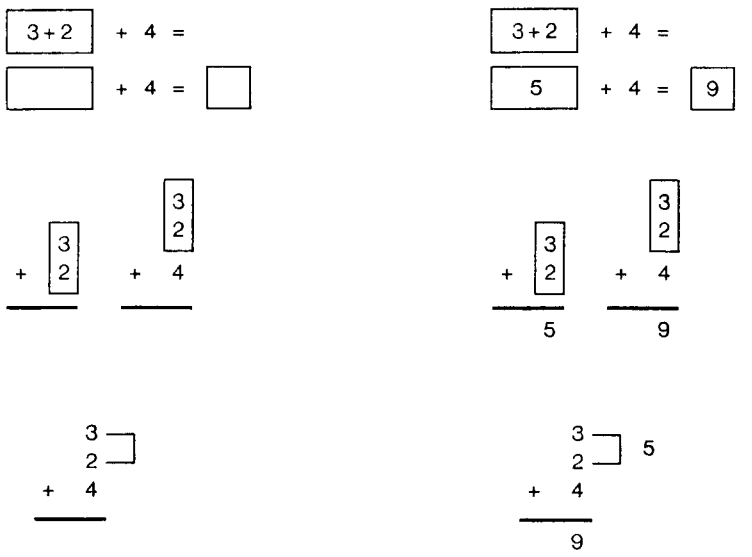


FIG. 9. Three formats for column addition obeying the show-work principle. Exercises appear unsolved on the left, solved on the right.

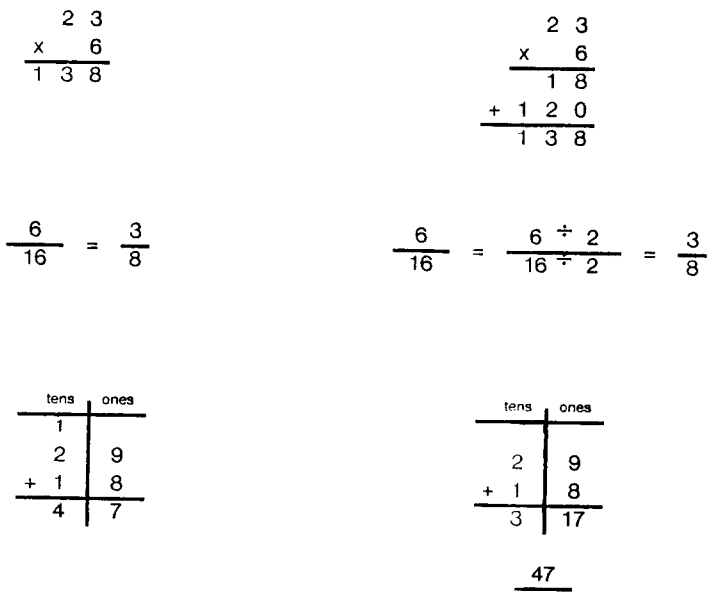


FIG. 10. Other exercise formats obeying the show-work principle. Exercises appear in normal format on left, in show-work format on right.

order to acquire the subskill. The learning of this lesson may proceed as if invisible object designators were banned from the representation language.

The second lesson teaches the subskill again, without writing the intermediate results. The second lesson is almost always headed by the key phrase, "Here is a shorter way to X" where X is the name of the skill. The students are being instructed that they will be doing exactly the same work (i.e., the path of fact functions is the same). They are left with the relatively simple problem of figuring out how the new material relates to the material they learned in the preceding lesson. This kind of learning is a kind of optimization. They learn how to do the same work with less writing. So, the normal lessons are "show-work" lessons; the learner does invisible object-free induction. The marked lessons are "hide-work" lessons; the learner does optimization learning. The felicity condition is called "show-work/hide-work" or just show-work for brevity.

7.3. Parallels between the invisible objects problem and the disjunction problem

The invisible objects problem and the disjunction problem are similar in many respects.

(1) Both the invisible objects problem and the disjunction problem are impossible to solve using unbiased induction. If the class of all possible generalizations allows free use of them, then there are far too many generalizations consistent with any finite set of examples. Hence, both the disjunction problem and the invisible object problem require extra constraints.

(2) Both can be solved trivially with absolute biases: bar their representational devices from the representation language. This is not a viable option in the mathematics domain because the target procedures use both disjunctions and invisible objects.

(3) In both cases, a relative bias that works is to minimize the use of the respective devices (i.e., to prefer generalizations with the fewest disjuncts and the fewest invisible object designators).

(4) For empirical reasons, minimization biases are not included in the theory. More accurate hypotheses are based on felicity conditions, i.e., tacitly held manner constraints.

8. Task Generality

Subtraction is the only task for which I have extensive data from human learners, so SIERRA was developed primarily to simulate the learning of subtraction. In order to make a rough assessment of its task generality, SIERRA was given lesson sequences for three new task domains, whose lesson sequences were drawn from a popular elementary mathematics textbook [13]:

(1) *Addition* of multi-column, multi-addend problems, such as

```
 307
  81
+620
```

(2) *Multiplication* of multi-column multiplication problems, such as

```
 307
×25
```

(3) *Sixth grade algebra*: The skill is to solve linear equations with one occurrence of one unknown, with natural number solutions. At the end of the sixth grade, students are expected to be able to solve $5(3x + 1) = 20$ but not $3x + 2x = 10$ or $5(3x + 1) = 9$.

SIERRA learned correct procedures for all three skills, although there are some caveats to this assertion that will be discussed in a moment. Rather than go through SIERRA's learning in detail, this section describes the difficulties encountered and the kinds or revisions that would be required to resolve them.

One problem is that a FOREACH goal type is needed. Given a sequence of objects of the same type (e.g., a sequence of columns), a FOREACH would execute a subgoal on each object. This new goal type is needed because some naturally occurring lesson sequences are not quite right for learning the tail-recursions that SIERRA currently uses to implement loops. Although the lesson sequences can be easily modified, that would harm SIERRA's cognitive fidelity. To keep SIERRA empirically accurate, a new relative bias towards iteration is needed for skeleton induction. This bias is most effectively implemented by including the FOREACH goal type in the representation language.

A problem was discovered during the last multiplication lesson. At the time of the lesson, the learner can solve single-digit multiplier problems, such as (a) below.

(a)
```
  3
 57
×  5
285
```

(b)
```
 57
×15
285
+570
855
```

Note the use of the scratch mark to indicate carrying. The lesson teaches how to do two-digit multiplier problems, such as (b) above. The addition subproblem presents no difficulties for SIERRA, because the initial knowledge state for multiplication includes a multi-column addition procedure that was learned from the addition lesson sequence.

The problem with the lesson is that it does not use scratch marks to indicate carrying. This causes two difficulties for SIERRA. First, the textbook does not

include an optimization lesson to teach how to suppress carry marks (using examples such as (c) below

(c)
$$\begin{array}{r} 57 \\ \times\ 5 \\ \hline 285 \end{array}$$

Perhaps teachers present such examples on their own initiative, without the guidance of a textbook lesson. A more serious problem is that even if there were an optimization lesson, there is no easy way to modify the multiplication procedure in order to suppress the scratch marks. The procedure has a loop, which iterates leftward through the top row, multiplying the row's digits by the single-digit multiplier. A carried digit is written during one invocation of the loop body and read during another invocation of the loop body (the next one, in fact). For this dataflow to happen applicatively, without writing on the page, the second invocation must somehow be called from the first invocation. There is no way to achieve this without a complete overhaul of the procedure's calling structure. SIERRA cannot do such a major overhaul. I'm not sure what students do, because I have no data for multiplication. However, I suspect that they do not radically overhaul their understanding of the procedure just to suppress scratch marks. I suspect that they use their fingers to hold the carried digit between multiplies, or they use some short-term memory resource to do so. Implementing the latter possibility would entail making the representation language nonapplicative, which would make parsing much more complex. On the other hand, if students use their fingers, then the hypotheses and representation can remain intact, but the representation of the state of a multiplication problem would have to have a "hand" added to it. Either modification would require significant enough programming that I simply stopped SIERRA's traversal of the multiplication sequence at this lesson. It was the last one, anyhow.

The most significant task dependencies concerned notational syntax. This is not surprising, since the four tasks employ quite different notations. SIERRA's treatment of problem states and their syntax has not yet been discussed in this article (see [44]). The basic idea, however, is simple to present. Problem states are represented as letters, digits and lines situated on a Cartesian plane, and a two-dimensional context-free grammar is used to parse them. This technique failed in some cases. For instance, given the expression "$5 - x$," the minus sign must be viewed two ways: as a prefix for the term following it and as an infix operator that separates the two terms. So there are two parse trees for "$5 - x$," four for "$2 + (5 - x)$," and so on. SIERRA's context-free grammar technique is combinatorially explosive. A better solution would be to redesign the parser and pattern matcher so that they keep local ambiguities local. This might, in fact, be the first step toward an interesting theory of the interpretation of mathematical notation.

To sum up, there were two main difficulties in getting SIERRA to learn other skills than subtraction. (1) The dataflow architecture is incomplete. Some globally bound resource (e.g., fingers, short-term memory) is needed to do carrying without scratch marks. (2) The notational grammars are not quite expressive enough. People seem to view the same problem state several ways, a facility that SIERRA's grammar system does not adequately support.

9. Concluding Remarks

Until now, it seemed that to be successful, an inducer had to use either absolute or relative biases. To put it differently, successful inducers have either been partially blind or strongly prejudiced. SIERRA is a demonstration that there is a third way. An inducer can be successful if it receives a well-structured example sequence whose structure it understands. That is, the example sequence obeys certain manner constraints and thereby encodes information about the structure of the target generalization. The learner takes advantage of these constraints in order to recover the "message" that is encoded in the form of the example sequence. In human learning situations, if neither teacher nor learner are aware of the manner constraints, then they warrant the name *felicity conditions*. So a successful inducer is either partially blind, strongly prejudiced, and/or felicitiously taught.

9.1. Some speculations on manner constraints and felicity conditions

In a certain sense, manner constraints may be optimal strategies for knowledge communication. For instance, in order to solve the learner's disjunction problem, the teacher's optimal strategy would be to point to a node in the learner's knowledge structure and say "disjoin that node with the following subprocedure: . . . ." Clearly, this is impossible. So the teacher says the next best thing. "Disjoin *some* node with the following subprocedure: . . . ." The learner must figure out which node to disjoin because the teacher cannot point to it. But the learner now knows that some disjunction is necessary and that the examples following the teacher's command will determine its contents. If it were not for the exigencies of school scheduling, this would be perhaps the optimal information transmission strategy. However, lessons must be about an hour long. This means that only some of the lesson boundaries will correspond to the teacher's command to start a new disjunction. The other lessons will finish up a previous lesson. In short, the optimal, feasible manner constraint for disjunctive information transmission could well be one-disjunct-per-lesson.

There is a great deal of complaining about the so-called knowledge acquisition bottleneck in developing expert systems. It seems to be quite difficult to get human experts to formalize their expertise as, e.g., production rules. One often heard solution is to have the system learn the knowledge on its own, say, by discovery or by analogy (e.g., [23, 28]). I tend to agree with Simon [42],

who predicts that programming will always be the most effective way to "educate" a computer. However, if Simon and I are wrong, and machine learning does hold promise as a solution to the knowledge acquisition bottleneck, then examining how human experts acquire their knowledge is a good research heuristic. Even a cursory examination shows that most human experts didn't discover their knowledge or infer it, they learned it from a mentor, either in school or as an apprentice.[14] A good mentor is careful about selecting tasks for the student that are appropriate for the student's current state of knowledge. That is, the instruction is not a randomly ordered sequence of tasks, but a carefully structured one. Yet few researchers are trying to get an expert systems to learn from the kinds of structured instruction that human experts receive. Such a system would take advantage of the format that its mentor places on the instruction. The present research, in its explication of felicity conditions, should be helpful in building such a knowledge acquisition system. Such a system will be easier for human experts to educate than present systems because the experts, many of whom are experienced teachers, are more familiar with formatting their knowledge as lesson sequences than as production rules.

9.2. Summary

Putting speculation aside, I'll review the techniques that have been presented. The formost is one-disjunct-per-lesson. Because of it, SIERRA's design is quite simple. The procedure induction problem is reduced to a series of subprocedure induction problems, one per lesson. Subprocedure induction reduces to three subproblems: skeleton induction, pattern induction and function induction.

Skeleton induction is performed by parsing the action sequences top-down as far as possible and bottom-up as far as possible. In neither case will parsing yield a complete parse tree. To complete the parse tree, a new piece of tree structure must be built to connect the top-down parse to the bottom-up parse. Any such structure is a candidate skeleton. This dual-parser calculation is done on each example in the lesson, yielding one set of skeletons per example. These sets are intersected, yielding the skeleton(s) that are consistent with all the examples in the lesson.

Pattern induction is performed by a standard technique, Mitchell's version space algorithm [29].

Function induction can employ a brute-force generate-and-test algorithm because there is a manner constraint that simplifies the problem. The show-

[14] Expert programmers, particularly older ones, are notorious exceptions. Many of them acquired their expertise without instruction. Perhaps this accounts for the widespread myth in the knowledge engineering community that domain experts became experts without instruction.

work constraint says that examples must "show all the work" when introducing a new subprocedure, That is, intermediate results must be written on the example where the learner can "see" them. Because a composition (nest) of two primitive functions has an intermediate result that is not written down, composite functions cannot be introduced during a normal lesson. Consequently, the learner only has to consider primitive functions and not compositions of functions when it does function induction. The show-work constraint makes function induction almost trivial.

9.3. Discussion

The "per lesson" part of one-disjunct-per-lesson is slightly misleading. It turns out that SIERRA could get along just fine without lessons. The skeleton induction algorithm is the one component that controls the introduction of disjunctions. The algorithm fails only if an example introduces two or more disjuncts. In particular, just by omitting skeleton intersection, SIERRA could get along fine without lesson boundaries. Its two manner constraints would become (1) at most one disjunct *per example*, and (2) an *example* is induced without invisible object designators unless it is marked as a hide-work example. A per-example learner might be more appropriate for some knowledge acquisition tasks than the current per-lesson version.

It bears reiterating that the dual-parser technique that performs skeleton induction is simple and efficient because the procedure representation language is applicative. If the language allowed side-effects, such as storage of information in global buffers or variables, then parsing would be much more difficult.

As mentioned earlier, SIERRA consists of three induction algorithms, for, respectively, skeletons, patterns and functions. This three-way decomposition may apply to other learning tasks than procedure learning. The application of the induction algorithms is limited only by the topology of the knowledge representations, and not, of course, by what those knowledge representations denote. I would expect these algorithms to apply, for instance, to other learning-by-explanation tasks. Explanations of stories often feature hierarchical structures similar to the calling structures of procedures. Such explanations are generated by instantiating and composing schemata. These schemata are analogous to subprocedures. Schemata often contain restrictions on slots that are equivalent to patterns. If these equivalences hold, then learning new schemata could be accomplished by the same techniques that are used here to learn subprocedures. To take a second example, electronic circuits and other engineered devices are often designed to have a hierarchy of modules. This hierarchy corresponds to the calling hierarchy of procedures. Patterns and functions may have analogs in device designs as well. If so, then the techniques presented here might suffice to learn how to analyze devices. In short, the three-way decomposition of the induction problem into induction of hierar-

chies, induction of pattern-like constructs and induction compositions may be quite generally applicable.

SIERRA's skeleton induction is a form of context-free grammar induction, since the skeleton of a whole AOG is precisely a context-free grammar. Skeleton induction can be used to induce grammars, as long as the learner's example sequence conforms to one-disjunct-per-example. On the other hand, if one-disjunct-per-example is not an appropriate manner constraint for some domain, then some other grammar induction algorithm may be employed to perform skeleton induction, while the other two induction algorithms can remain relatively unchanged (for reviews of grammar induction, see [9, 15]).

A beneficial consequence of one-disjunct-per-lesson is that rule patterns are pure conjunctions. This means that a nonheuristic, complete induction algorithm, based on Mitchell's version space technique [29], can be employed to induce the conditions. However, it turns out to be infeasible to use just the version space technique. For empirical reasons, SIERRA must use nontoy patterns. A single pattern may have 50 variables and 200 relations. For patterns of such sizes, induction is just not practical without further constraints on patterns. Fortunately, there are several well-motivated constraints available in this domain. The two mentioned at the end of Section 6 seem likely to be useful outside the present domain.

The invisible objects problem was somewhat of a surprise. It took a long time to figure out that function induction had an inherent problem (i.e., that SIERRA wasn't a victim of incomplete sets of examples). It took even longer to become convinced that there is no least-commitment, incremental algorithm to solve it, such as the version space algorithm. If an inducer cannot ban invisible object designators, there seem to be only two ways to get around the invisible objects problem: BACON3's nonincremental induction (see Section 7) or SIERRA's show-work manner constraint. Perhaps more research will find other techniques. The problem of inducing invisible object designators has received little attention from machine learning.

ACKNOWLEDGMENT

I am deeply grateful to John Seely Brown for the ideas and encouragement he has lent to this project. I would like to thank the readers whose thoughtful comments have helped this document along: Agustin Araya, Tom Dietterich, John Laird, Stan Lanning, Steve Minton, Paul Rosenbloom, and Dave Wilkins. This research was supported by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under Contract No. N00014-82C-0067, Contract Authority Identification No. NR667-477.

REFERENCES

1. Amarel, S., Representations and modelling in problems of program formation, in: B. Meltzer and D. Michie (Eds.), *Machine Intelligence* **6** (Elsevier, New York, 1971).
2. Anderson, J.R., Tuning of search of the problem space for geometry proofs, Carnegie-Mellon University, Pittsburgh, PA, 1982.

3. Anzai, Y. and Simon, H.A., The theory of learning by doing, *Psychol. Rev.* **86** (1979) 124–140.
4. Austin, J.L., *How to Do Things with Words* (Oxford University Press, New York, 1962).
5. Badre, N.A., Computer learning from English text, ERL-M372, University of California at Berkeley, Electronic Research Laboratory, Berkeley, CA, 1972.
6. Bauer, M.A., A basis for the acquisition of procedures from protocols, in: *Proceedings IJCAI-75*, Tblisi, USSR, (1975) 226–231.
7. Biermann, A.W., On the inference of Turing machines from sample computations, *Artificial Intelligence* **3** (1972) 181–198.
8. Biermann, A.W., The inference of regular LISP programs from examples, *IEEE Trans. Syst. Man Cybern.* **8** (1978) 585–600.
9. Biermann, A.W. and Feldman, J.A., A survey of results in grammatical inference, in: S. Watanabe, (Ed.), *Frontiers of Pattern Recognition* (Academic Press, New York, 1972).
10. DeJong, G., Generalizations based on explanations, in: *Proceedings IJCAI-81*, Vancouver, BC, 1981.
11. DeJong, G., A brief overview of explanatory schema acquisition, in: *Proceedings Third Machine Learning Workshop*, 1985.
12. Dietterich, T.G. and Michalski, R.S., A comparative review of selected methods for learning from examples, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell, (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga Press, Palo Alto, CA, 1986).
13. Dilley, C.A., Rucker, W.E. and Jackson, A.E., *Heath Elementary Mathematics* (Heath, Lexington, MA, 1975).
14. Ellman, T., Explanation-based learning in logic circuit design, in: *Proceedings Third Machine Learning Workshop*, 1985.
15. Fu; K. and Booth, T., Grammatical inference: Introduction and survey, *IEEE Trans. Syst. Man Cybern.* **5** (1975) 95–111.
16. Genesereth, M.R., The role of plans in intelligent teaching systems, in: J.S. Brown and D. Sleeman (Eds.), *Intelligent Tutoring Systems* (Academic Press, New York, 1982).
17. Gold, E.M., Language identification in the limit, *Inf. Control* **10** (1967) 447–474.
18. Hedrick, C.L., Learning production systems from examples, *Artificial Intelligence* **7** (1976) 21–49.
19. Hempel, C.G., Studies in the logic of confirmation, *Mind* **54** (1945) 1–26, 97–121.
20. Johnson, L. and Soloway, E., Intention-based diagnosis of programming errors, in: *Proceedings AAAI-84*, Austin, TX (1984) 162–168.
21. Kaplan, R.M., A general syntactic processor, in: R. Rustin (Ed.), *Natural Language Processing* (Algorithmics Press, New York, 1973).
22. Knuth, D.E., Semantics of context-free languages, *Math. Syst. Theory* **2** (1968) 127–145.
23. Koster, C.H.A., Affix grammars, in: J.E. Peck (Ed.), *ALGOL 68 Implementation* (North-Holland, Amsterdam, 1971).
24. Langley, P., Rediscovering physics with Bacon3, in: *Proceedings IJCAI-79*, Tokyo, Japan, 1979.
25. Langley, P., Ohlsson, S. and Sage, S., A machine learning approach to student modeling, CMU-RI-TR-84-7, Carnegie-Mellon University, Pittsburgh, PA, 1984.
26. McDermott, J. and Forgy, C.L., Production system conflict resolution strategies, in: D.A. Waterman and F. Hayes-Roth (Eds.), *Pattern-directed Inference Systems* (Academic Press, New York, 1978).
27. Miller, M.L. and Goldstein, I.P., Overview of a Linguistic theory of design, Memo 383A, MIT, Artificial Intelligence Laboratory, Cambridge, MA, 1977.
28. Mitchell, T.M., The need for biases in learning generalizations, CBM-TR-117, Rutgers University Computer Science Department, New Brunswick, NJ, 1980.
29. Mitchell, T.M., Generalization as search, *Artificial Intelligence* **18** (1982) 203–226.
30. Mitchell, T.M., Mahadevan, S. and Steinberg, L., A learning apprentice system for VLSI design, in: *Proceedings Third Machine Learning Workshop*, 1985.

31. Mitchell, T.M., Utgoff, P.E. and Banerji, R.B., Learning problem-solving heuristics by experimentation, in: R.S. Michalski, T.M. Mitchell and J.G. Carbonell (Eds.), *Machine Learning* (Tioga Press, Palo Alto, CA, 1983).
32. Mooney, R., Generalizing explanations of narratives into schemata, in: *Proceedings Third Machine Learning Workshop*, 1985.
33. Neves, D.M., Learning procedures from examples, Ph.D. Thesis, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA, 1981.
34. Newell, A., The knowledge level, *Artificial Intelligence* **18** (1982) 87–127.
35. Osherson, D.N., Stob, M. and Weinstein, S., Ideal learning machines, *Cognitive Sci.* **6** (1982), 277–290.
36. Rich, C. and Shrobe, H., Initial report on a Lisp programmer's apprentice, AI-TR-354, MIT, Artificial Intelligence Laboratory, Cambridge, MA, 1976.
37. Searle, J., *Speech Acts: An Essay in the Philosophy of Language* (Cambridge University Press, Cambridge, U.K. 1969).
38. Segre, A.M., Explanation-based manipulator learning, in: *Proceedings Third Machine Learning Workshop*, 1985.
39. Shavlik, J., Learning classical physics, in: *Proceedings Third Machine Learning Workshop*, 1985.
40. Shaw, D.E., Swartout, W.R. and Green, C.C., Inferring Lisp programs from examples, in: *Proceedings IJCAI-75*, Tblisi, USSR, 1975.
41. Siklossy, L. and Sykes, D.A., Automatic program synthesis from example problems, in: *Proceedings IJCAI-75*, Tblisi, USSR, 1975.
42. *Simon, H.A., Why should machines learn? in: R.S. Michalski, T.M. Mitchell and J.G. Carbonell (Eds.), Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983).
43. Smith, D.E., Focuser: A strategic interaction paradigm for language acquisition, Tech. Rept. LCSR-TR-36, Laboratory for Computer Science Research, Rutgers University, New Brunswick, NJ, 1982.
44. VanLehn, K., Felicity conditions for human skill acquisition: Validating an AI-based theory, Tech. Rept. CIS-21, Xerox Parc, Palo Alto, CA, 1983.
45. VanLehn, K., Representation of procedures in repair theory, in: H.P. Ginsberg, (Ed.), *The Development of Mathematical Thinking* (Academic Press, New York, 1983).
46. Watt, D.A., The parsing problem for affix grammars, *Acta Inf.* **8** (1977) 1–20.
47. Winston, P.H., Learning structural descriptions from examples, AI-TR-231, MIT, Artificial Intelligence Laboratory, Cambridge, MA, 1970.
48. Winston, P.H., Learning structural descriptions from examples, in: P.H. Winston (Ed.), *The Psychology of Computer Vision* (McGraw-Hill, New York, 1975).
49. Winston, P.H., Learning by creating and justifying transfer frames, *Artificial Intelligence* **10** (1978) 147–172.
50. Winston, P.H., Learning new principles from precedents and exercises, AIM 632, MIT, Artificial Intelligence Laboratory, Cambridge, MA, 1981.
51. Woods, W.A., Kaplan, R. and Nash-Webber, B., The lunar sciences natural language information system, BBN Rept. 2378, Bolt, Beranek and Newman, Cambridge, MA, 1972.

Received September 1984; revised version received April 1986