(15)

May 1982

Report No. STAN-CS-82-913

Also numbered: HPP-82-10

AD A122351

20000731245

Learning and Inductive Inference

by

Thomas G. Dietterich, Bob London, Kenneth Clarkson, Geoff Dromey

A section of the Handbook of Artificial Intelligence
edited by Paul R. Cohen and Edward A. Feigenbaum

Department of Computer Science

Stanford University
Stanford, CA 94305

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1 REPORT NUMBER	2 GOVT ACCESSION NO: AD-A122351
	3 RECIPIENT'S CATALOG NUMBER
4 TITLE (and Subtitle): Learning and Inductive Inference	5 TYPE OF REPORT & PERIOD COVERED: technical, July 1982
	6 PERFORMING ORG. REPORT NUMBER
7 AUTHOR(s): Thomas G. Dietterich (edited by Paul R. Cohen and Edward A. Feigenbaum)	8 CONTRACT OR GRANT NUMBER(s): MDA 903-80-C-0107
9 PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science Stanford University Stanford, California 94305 U.S.A.	10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11 CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency Information Processing Techniques Office 1400 Wilson Avenue, Arlington, VA 22209	12 REPORT DATE: July 1982 13 NO. OF PAGES: 215
	15 SECURITY CLASS. (of this report): Unclassified
14 MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office): Mr. Robin Simpson, Resident Representative Office of Naval Research, Durand 165 Stanford University	15a DECLASSIFICATION/DOWNGRADING SCHEDULE
16 DISTRIBUTION STATEMENT (of this report): Reproduction in whole or in part is permitted for any purpose of the U.S. Government.	
17 DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)	
18 SUPPLEMENTARY NOTES	
19 KEY WORDS (Continue on reverse side if necessary and identify by block number)	
20 ABSTRACT (Continue on reverse side if necessary and identify by block number): (see reverse side)	

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

This technical report surveys Artificial Intelligence research in the area of learning and inductive inference. It was written as Chapter XIV of Volume III (edited by Paul R. Cohen and Edward A. Feigenbaum) of the Handbook of Artificial Intelligence. The main results of the report are: (a) a simple model that serves to generate a taxonomy of learning systems, (b) the explication and clarification of four methods of learning single concepts, (c) some understanding of the underlying causes of the credit-assignment problem (and possible solutions), and (d) some identification of open research problems and areas that have received little attention.

Briefly, the simple model posits the existence of a Learning Element whose task is to add or modify knowledge stored in a Knowledge Base so that the behavior of the Performance Element is improved. The key considerations that generate the taxonomy of learning systems include (a) the difference between the level at which the Learning Element receives new information and the level at which the Performance Element can use that information and (b) the complexity of the performance task as measured by the number of distinct concepts or rules needed and by the complexity of the inference process that employs those concepts and rules.

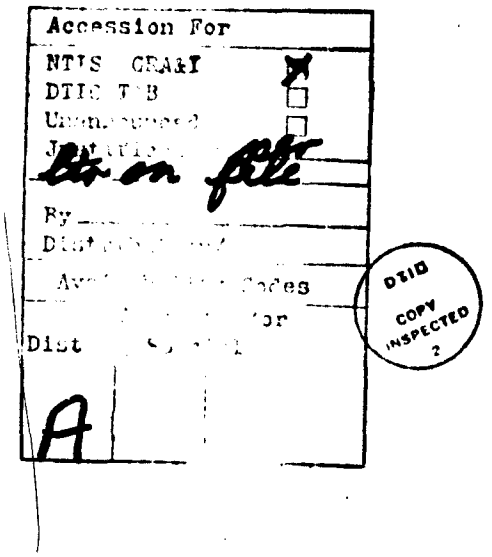
The credit-assignment problem is seen to arise in learning tasks where the Performance Element makes composite inferences, such as chaining together several production rules, but where only a global source of feedback is available. The classic case is where a game-playing program must make several moves before being able to evaluate how good those moves were. The credit-assignment problem is the problem of splitting up the global feedback to apportion credit or blame to the individual moves.

Areas in which further research is needed include (a) advice-taking; (b) learning from analogies; (c) experiment planning and instance selection in order to test a hypothesis or remove some ambiguity; (d) learning to perform complex performance tasks, which would require solving the credit-assignment problem in some non-trivial domain; and (e) learning without a fixed description language.

The report is structured as a set of articles. Seven of the articles present the main problems and issues in learning research, while the remaining fifteen articles describe particular learning systems that have been developed.

Chapter XIV

Learning and Inductive Inference



CHAPTER XIV: LEARNING AND INDUCTIVE INFERENCE

PREFACE

THIS TECHNICAL REPORT surveys Artificial Intelligence research in the area of learning and inductive inference. It was written as Chapter XIV of Volume III of the *Handbook of Artificial Intelligence*. Since AI learning research is still in its infancy, this chapter does not present many well-understood research results. Instead, we have attempted to provide a framework for viewing past research and a list of open problems for future research.

This survey is necessarily incomplete, and we apologize to those researchers whose work is not mentioned. In choosing which systems to include, we considered several different criteria, such as historical importance (e.g., Samuel, Waterman, Winston), performance (e.g., CLS/ID3, Meta-DENDRAL, Samuel), relevance to outstanding problems (e.g., LEX), and demonstration of unusual techniques (e.g., Lenat, Dietterich and Michalski, Langley). We attempted to select at least one representative program from each of the various learning methods and learning situations. In some cases, we have also taken liberties in recasting the terminology and representation of a system in order to improve the uniformity of the chapter (e.g., Hayes-Roth, Sussman).

This chapter was a group effort. Bob London helped to outline the chapter and wrote the articles on rote learning and advice-taking. Kenneth Clarkson contributed the article on grammatical inference, and Geoff Dromey wrote the article on adaptive learning. The remainder of the chapter was written by Tom Dietterich. Valuable criticisms were provided by our reviewers: James S. Bennett, Bruce G. Buchanan, Ryszard S. Michalski, Thomas M. Mitchell, Jack Mostow, David Shur, and Paul Utgoff. In addition, the volume editor, Paul R. Cohen, and the professional editor, Dianne Kanerva, helped immensely to improve the form and content of the chapter. Thanks also to Jose L. Gonzalez for assisting in the production of this technical report.

We hope that this chapter will serve both as a useful reference for students of learning and as a technical contribution to AI learning research.

Tom Dietterich, chapter editor

This research was supported by the Defense Advanced Research Projects Agency (ARPA Contract No. MDA 903-80-C-0107). The views and conclusions of this report should not be interpreted as necessarily representing the official policies, either express or implied of the Defense Advanced Research Projects Agency or the United States Government.

A. OVERVIEW

LEARNING is a very general term denoting the way in which people (and computers) increase their knowledge and improve their skills. From the very beginnings of AI, researchers have sought to understand the process of learning and to create computer programs that can learn.

There are two fundamental reasons for studying learning. One is to understand the process itself. By developing computer models of learning, psychologists have attempted to gain an understanding of the way humans learn. Philosophers since Plato have also been interested in learning research, because it may help them understand what knowledge is and how it grows.

The second reason for conducting learning research is to provide computers with the ability to learn. It has long been a goal of AI to develop computer systems that could be taught rather than programmed. Many other applications of computers, such as intelligent programs for assisting scientists, involve the acquisition of new knowledge. Thus, learning research has potential for extending the range of problems to which computers can be applied.

In this overview article, we first present a short history of AI research on learning. This is followed by a review of AI perspectives on learning, from which a simple model of learning is developed. This model allows us to discuss some of the major factors affecting the design of learning systems.

A Brief History of AI Research on Learning

AI research on learning has evolved through three stages. The first, and most optimistic, stage of work centered on self-organizing systems that modified themselves to adapt to their environments (see Yovits, Jacobi, and Goldstein, 1962). The hope was that if a system were given a set of stimuli, a source of feedback, and enough degrees of freedom to modify its own organization, it would adapt itself toward an optimum organization. Attempts were made, for example, to simulate evolution in the hope that intelligent programs would result from the processes of random mutation and natural selection (Friedberg, 1958; Friedberg, Dunham, and North, 1959; Fogel, Owens, and Walsh, 1966). Various computational analogues of neurons were developed and tested; foremost of these was the perceptron (Rosenblatt, 1957). Unfortunately, most of these attempts failed to produce systems of any complexity or intelligence (see Article XIV.D2 on adaptive learning).

Theoretical limitations were discovered that dampened the optimism of these early AI researchers (see Minsky and Papert, 1969). In the 1960s, attention moved away from learning toward knowledge-based problem solving and

natural-language understanding (Minsky, 1968). Those people who continued to work with adaptive systems ceased to consider themselves AI researchers; their research branched off to become a subarea of linear systems theory. Adaptive-systems techniques are presently applied to problems in pattern recognition and control theory.

The beginning of the 1970s saw a renewal of interest in learning with the publication of Winston's (1970) influential thesis. In this second stage of learning research, workers adopted the view that learning is a complex and difficult process and that, consequently, a learning system cannot be expected to learn high-level concepts by starting without any knowledge at all. This view has led researchers, on the one hand, to study simple learning problems in depth (such as learning single concepts) and, on the other, to incorporate large amounts of domain knowledge into learning systems (such as the Meta-DENDRAL and AM programs discussed in Articles XIV.D4b and XIV.D4c) so that they could discover high-level concepts.

A third stage of learning research, motivated by the need to acquire knowledge for expert systems, is now under way. Unlike the first two phases of learning research, which focused on rote learning and learning from examples, the current work looks at all forms of learning, including advice-taking and learning from analogies.

Four Perspectives on Learning

Herbert Simon (in press) defines learning as *any process by which a system improves its performance*. His definition assumes that the system has a task that it is attempting to perform. It may improve its performance by applying new methods and knowledge or by improving existing methods and knowledge to make them faster, more accurate, or more robust.

A more constrained view of learning, adopted by many people who work on expert systems, is that learning is *the acquisition of explicit knowledge*. Many expert systems represent their expertise as large collections of rules that need to be acquired, organized, and extended. This view emphasizes the importance of making the acquired knowledge explicit, so that it can be easily verified, modified, and explained. Researchers are presently working on knowledge-acquisition systems that discover new rules from examples or accept new rules from experts and integrate them into the knowledge base of the system.

A third view is that learning is *skill acquisition*. Psychologists have pointed out that long after people are told *how* to do a task, such as touch typing or computer programming, their performance on that task continues to improve through practice (Norman, 1980). It appears that although people can easily understand verbal instructions on how to perform a task, much work remains to be done to turn that verbal knowledge into efficient mental or muscular operations. Researchers in AI and cognitive psychology have sought

to understand the kinds of knowledge that are needed to perform skillfully. The processes by which people acquire this knowledge through practice are little understood.

The collective enterprise of science is usually considered to be one of the most effective ways that our culture learns about the world. Thus, a fourth view of learning is that it is *theory formation, hypothesis formation, and inductive inference*. Work on theory formation has centered on understanding how scientists build theories to describe and explain complex phenomena. A necessary part of theory formation is hypothesis formation—the activity of finding one or more plausible hypotheses to explain a particular set of data in the context of a more general theory. Another aspect of theory formation is inductive inference—the process of inferring general laws from particular examples.

A Simple Model of Learning and Its Implications for the Design of Learning Systems

Of these four views of learning, Simon's (in press) is perhaps the most encompassing. Taking his definition as a starting point, we have developed the simple model of learning systems shown in Figure A-1. Throughout this chapter, we use this simple model to organize our discussion of learning systems.

In the model, the circles denote declarative bodies of information (e.g., facts represented in predicate calculus or statements made by an expert), while the boxes denote procedures. The arrows show the predominant direction of data flow through the learning system. The environment supplies some information to the learning element, the learning element uses this information to make improvements in an explicit knowledge base, and the performance element uses the knowledge base to perform its task. Finally, information gained during attempts to perform the task can serve as feedback to the learning element. This model is primitive and omits many important functions. It is useful, however, in that it allows us to classify learning systems according to how they "fill" these four functional units. In any particular application, the environment, the knowledge base, and the performance task determine the nature of the particular learning problem and, hence, the particular functions that the learning element must fulfill. In the following three sections, we

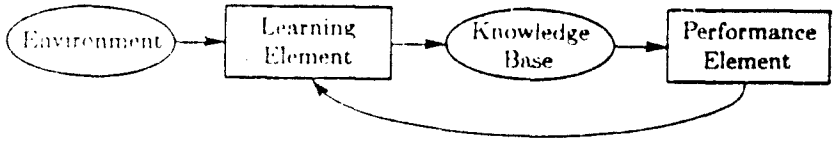


Figure A-1. A simple model of learning systems.

examine the role of each of these three functional units that surround the learning element.

The Environment

The most important factor affecting the design of learning systems is the kind of information supplied to the system by the environment—particularly the *level* and *quality* of this information.

The *level* of information refers to the degree of generality (or domain of applicability) of the information relative to the needs of the performance element. High-level information is abstract information that is relevant to a broad class of problems. Low-level information is detailed information that is relevant to a single problem. The task of the learning element can be viewed as the task of bridging the gap between the level at which the information is provided by the environment and the level at which the performance element can use the information to carry out its function. Thus, if the learning system is given very abstract (high-level) advice about its performance task, it must fill in the missing details, so that the performance element can interpret the information in particular situations. Correspondingly, if the system is given very specific (low-level) information about how to perform in particular situations, the learning element must generalize this information—by ignoring unimportant details—into a rule that can be used to guide the performance element in a broader class of situations.

Since its knowledge is imperfect, the learning element does not know in advance exactly how to fill in missing details or ignore unimportant details. Consequently, it must guess—that is, *form hypotheses*—about how the gap between the levels should be bridged. After guessing, the system must receive some feedback that allows it to evaluate its hypotheses and revise them if necessary. It is in this way that a learning system learns: by trial and error.

The level of the information provided by the environment determines the kinds of hypotheses that the system must generate. Four basic learning situations can be discerned:

1. *Rote learning*, in which the environment provides information exactly at the level of the performance task and, thus, no hypotheses are needed.
2. *Learning by being told* in which the information provided by the environment is too abstract or general and, thus, the learning element must hypothesize the missing details.
3. *Learning from examples*, in which the information provided by the environment is too specific and detailed and, thus, the learning element must hypothesize more general rules.
4. *Learning by analogy*, in which the information provided by the environment is relevant only to an analogous performance task and, thus, the

learning system must discover the analogy and hypothesize analogous rules for its present performance task.

Each of these learning situations is discussed in more detail below.

The *quality* of information can have a significant effect on the difficulty of the learning task. Induction is easiest, for example, when the training instances are selected by a cooperative teacher who chooses "clean" examples, classifies them, and presents them in good pedagogical order. Learning by induction is particularly difficult when the training instances are made up of noise-ridden, unclassified data that are "presented" by nature in an uncontrollable fashion. Similarly, in advice-taking systems, information is of little use if it is provided by an unreliable and inarticulate expert; rote learning cannot succeed with poor-quality, possibly contradictory data; and analogies are useless if they are cluttered with errors.

The Knowledge Base

The second factor affecting the design of learning systems is the knowledge base, its *form* and *content*. We discuss first the *form*, or representational system, in which the knowledge base is expressed; it is a particularly important design consideration (see Chap. III, in Vol. I, on representation of knowledge). Most work in learning has used one of two basic representational forms—feature vectors and predicate calculus – although other forms, such as production rules, grammars, LISP functions, numerical polynomials, semantic nets, and frames, have also been used. These representational forms vary along four important dimensions: expressiveness, ease of inference, modifiability, and extendability.

Expressiveness of the representation. In any AI system it is important to have a representation in which the relevant knowledge can be easily expressed. Feature vectors, for example, are useful for describing objects that lack internal structure. They describe objects in terms of a fixed set of features (such as color, shape, and size) that take on a finite set of values (such as red or green, circle or square, and small or large). Predicate calculus, on the other hand, is useful for describing structured objects and situations. A situation in which a red object is on top of a green one, for example, can be expressed as $\exists x, y : \mathrm{RED}(x) \wedge \mathrm{GREEN}(y) \wedge \mathrm{ONTOP}(x, y)$.

Ease of inference within the representation. The computational cost of performing inference is another important property of a representational system. One type of inference frequently required in learning systems is the comparison of two descriptions to determine whether they are equivalent. It is very easy to test two feature-vectors for equivalence. The comparison of two predicate-calculus expressions is more costly. Since many learning systems must search large spaces of possible descriptions, the cost of comparisons can severely limit the extent of these searches.

Modifiability of the knowledge base. A learning system must, by its very nature, modify some part of the knowledge base to store the knowledge it is gaining. Consequently, most learning systems have employed explicit, stylized representations (such as feature vectors, predicate calculus, and production rules) in which it is easy to add knowledge to the knowledge base. Very little attention has been given to the problem of adding to knowledge bases in which substantial revision and integration must be performed. These problems arise, for example, in systems that refer to time or state information (e.g., procedural representations) and in systems that make default assumptions that may later need to be retracted.

Extendability of the representation. For a learning program to manipulate explicitly its acquired knowledge, there must be a meta-level description within the program that tells how the representation is structured. This meta-level knowledge has usually been embodied in procedures that manipulate the data structures of the representation. Of recent interest in learning research, however, are representational systems in which this meta-knowledge is also made an explicit part of the knowledge base (see Davis, 1976). The purpose is to allow the program to examine and alter its own representation by adding vocabulary terms and representational structures. This ability in turn provides the possibility of developing learning systems that are open-ended—that is, that can learn successively more complex units of knowledge without limit. The outstanding example of an extendable representation is Lenat's (1976) AM program (see Article XIV.D4c), which allows new concepts to be defined in terms of old ones. Recent work on RLL (Greiner and Lenat, 1980; Greiner, 1980) has pushed this idea much further toward allowing a program to define new representations dynamically.

Now that we have examined issues relating to the *form* of the knowledge base, we turn our attention to its *content*. A learning system does not gain knowledge by starting "from scratch," that is, without any knowledge at all. Some knowledge must be employed by every learning system to understand the information provided by the environment, to form hypotheses, and to test and refine those hypotheses. Thus, it is more appropriate to view a learning system as extending and improving an existing body of knowledge. Unfortunately, in most learning systems, the knowledge employed is not explicit; it is built into the program by the designer. Throughout this chapter, we try to point out the ways in which domain-specific knowledge has entered into existing learning systems.

The Performance Element

The performance element is the focus of the whole learning system, since it is the actions of the performance element that the learning element is trying to improve. There are three important issues related to the performance element: complexity, feedback, and transparency.

First, the *complexity* of the task is important. Complex tasks require more knowledge than simple tasks. For instance, a simple task like binary classification, in which objects are classified into one of two groups, requires only a single classification rule. On the other hand, a program that can play a reasonable poker game (Waterman, 1970) needs about 20 rules, and a medical-diagnosis system like MYCIN (Shortliffe, 1976) employs several hundred rules.

In learning from examples, three classes of performance tasks can be distinguished according to their complexity. The simplest performance task is *classification* or *prediction* based on a *single concept* or *rule*. Indeed, the problem of learning single concepts from examples has received more study than any other problem in AI learning research. Slightly more complex are tasks involving multiple concepts. An example is the problem of predicting which bonds of an organic molecule will be broken in the mass spectrometer; the DENDRAL prediction program employs a set of cleavage rules to perform this task. The most complex tasks for which learning systems have been developed are small planning tasks in which a set of rules must be applied in *sequence*. Symbolic integration, for example, is a task that requires chaining together several integration rules to obtain a solution. The articles on learning from examples consider these three classes of performance tasks and their corresponding learning methods.

As the performance task becomes more complex and the knowledge base grows in size, the problems of *integrating new rules* and *diagnosing incorrect rules* become more complicated. The *integration problem* —that is, the problem of integrating a new rule into an existing set of rules—is difficult, because the learning system must consider possible interactions between the new rule and the previous rules. During the construction of the MYCIN system, for example, there were several cases in which a new rule caused existing rules to be applied incorrectly or to cease being applied altogether (see Article VIII.B1).

The problem of diagnosing incorrect rules—also known as the *credit-assignment problem* (Minsky, 1963)—can be very difficult in systems that perform a sequence of actions before receiving any feedback. Consider, for example, the problem of learning to play chess by first playing a complete game, then determining who won and lost, and finally updating the knowledge base accordingly. The credit-assignment problem is the problem of assigning credit or blame to the individual decisions that led to some overall result—in this case, the individual chess moves that contributed most to the win or loss.

The second important issue related to the performance task is the role of the performance element in providing *feedback* to the learning element. All learning systems must have some way of evaluating the hypotheses that have been proposed by the learning element. Some programs have a separate body of knowledge for such evaluation. The AM program, for example, has many heuristic rules that assess the interestingness of the new concepts developed by the learning element. A more frequently used technique, however, is to have the environment, often a teacher, provide an external *performance standard*.

Then, by observing how well the performance element is doing relative to this standard, the system can evaluate its current store of hypotheses.

In systems that learn a single concept from training instances, the performance standard is the correct classification of each training instance (as to whether it is, or is not, an instance of the concept to be learned). In most systems, the training instances are preclassified by a reliable teacher. In the META-DENDRAL system (see Article XIV.D1b), the performance standard is the actual mass spectrum produced when a molecule of known structure is placed in the mass spectrometer.

The third issue regarding the performance task is the *transparency* of the performance element. For the learning element to assign credit or blame to individual rules in the knowledge base, it is useful for the learning element to have access to the internal actions of the performance element. Consider again the *problem of learning how to play chess*. If the learning element is given a trace of all the moves that were *considered* by the performance element (rather than only those moves that were actually chosen), the credit-assignment problem is easier to solve.

Overview of the Chapter

In the previous section, we discussed the interaction between the information provided by the environment and the problems that are presented to the learning element. From this analysis, four learning situations could be discerned. In this section, we discuss these four situations in detail and give an example of a learning problem in each situation. The remainder of this chapter is organized around these four situations, with a separate set of articles devoted to each.

Rote learning. The simplest learning situation is one in which the environment supplies knowledge in a form that can be used directly by the performance element. The learning system does not need to do any processing to understand or interpret the information supplied by the environment. All it must do is memorize the incoming information for later use. This is a form of rote learning—if it is considered learning at all. Virtually every computer system can be said to do rote learning insofar as it stores instructions for performing a task.

An important AI study of rote learning was undertaken by Samuel (1959, 1967). He developed a checkers-playing program that was able to improve its performance by memorizing every board position that it evaluated. The program used a standard minimax look-ahead search (see Chap. 1, in Vol. I) that evaluated potential future board positions. A simple polynomial evaluation function measured board properties such as center control, fork threats, and possible exchanges. In terms of our primitive learning-system model, the look-ahead search portion of Samuel's program served as the "environment." It supplied the learning element with board positions and their backed-up

minimax values. The learning element simply stored these board positions and indexed them for rapid retrieval. Interestingly, the look-ahead search portion of Samuel's program also served as part of the performance element that played a game of checkers against an opponent. It used the previously memorized board positions to improve the speed and depth of its look-ahead search during subsequent games.

Learning by being told—Advice-taking. When a system is given vague, general-purpose knowledge or advice, it must transform this high-level knowledge into a form that can be used readily by the performance element. This transformation is called *operationalization*. The system must understand and interpret the high-level knowledge and relate it to what it already knows. Operationalization is an active process that can involve such activities as deducing the consequences of what it has been told, making assumptions and "filling in the details," and deciding when to ask for more advice. McCarthy's (1958) proposal for an "advice taker" was the first description of a system that could learn by being told. More recent work in the area of learning by being told includes the TEIRESIAS program (Davis, 1976) and Mostow's program FOO (Mostow and Hayes-Roth, 1979; Mostow, 1981).

FOO, for example, is told the rules of the game of Hearts and is given vague strategic advice such as "Avoid taking points." It operationalizes this advice into specific strategies such as "Play lower than the highest card so far in the suit led." This kind of operationalization is similar to the kind of processing performed by ordinary language compilers that convert unexecutable high-level languages into directly interpretable machine code. In the same trivial sense that every computer system can be said to do rote learning, every system can also be said to learn by being told: Advice in the form of a high-level language program is compiled and assembled into an executable object program.

Learning from examples—Induction. One way to teach a system how to perform a task is to present it with examples of how it should behave. The system must then generalize these examples to find higher level rules that can be applied to guide the performance element. Examples can be viewed as being pieces of very specific knowledge that cannot be used efficiently by the performance element. These are transformed into more general, higher level pieces of knowledge that can be used effectively.

For example, consider the problem of teaching a program to recognize poker hands that contain a *pair*. The program would be presented with sample hands that, it is told, contain pairs. Here is such a training instance:

4 of clubs, 4 of spades, 5 of diamonds, 6 of hearts, jack of diamonds.

This training example is a very specific piece of knowledge. If the program merely memorized it (by rote learning), it would now *know* that the hand

4 of clubs, 4 of spades, 5 of diamonds, 6 of hearts, jack of diamonds

contains a pair. It would *not* know that the hand

4 of clubs, 4 of spades, 5 of diamonds, 6 of hearts, 8 of diamonds

also contains a pair, since the program has not *generalized* its knowledge. To recognize all possible pair hands, the program needs to discover that the hand must contain two cards of the same rank and that the remaining cards are irrelevant. The generalization of knowledge to make it apply to a broader class of situations is the key inference process in learning from examples.

Learning by analogy. If a system has available to it a knowledge base for a related performance task, it may be able to improve its own performance by recognizing analogies and transferring the relevant knowledge from the other knowledge base. Thus far, however, very little work has been done in this area. Some of the open research questions are: What exactly is an analogy? How are analogies recognized? How is the relevant knowledge transferred from the analogous knowledge base and applied to accomplish the desired tasks?

Suppose, for example, that a program has available to it a knowledge base describing how to diagnose diseases in human beings and someone wants to use the same program to diagnose computer-system failures. By finding the proper analogies, the program can develop classes of computer failures ("diseases") and possible solutions ("therapies"). Diagnostic procedures can be transferred as the analogy is developed (e.g., x-rays can be analogized to core dumps).

We do not include in this chapter any articles discussing learning by analogy, since this area has not received much attention.

Conclusion

This introduction has surveyed AI research on learning and presented a simple model of AI learning systems. The model has been used to discuss the factors that bear upon the design of the learning element. These include the level and quality of the information provided by the environment, the form and content of the knowledge base, and the complexity and transparency of the performance element. Of these factors, the most important is the level of the information provided by the environment. This has been used to develop the simple taxonomy of four learning situations that provides an organization for the remainder of this chapter.

References

Buchanan et al. (1977) survey several systems and present a general model of learning systems. See also Lenat, Hayes-Roth, and Klahr (1979) and Dietterich and Michalski (1979).

B. ROTE LEARNING

B1. Issues

ROTE LEARNING is memorization; it is saving new knowledge so that when it is needed again, the only problem will be *retrieval*, rather than a repeated computation, inference, or query. Two extreme perspectives on rote learning are possible. One view says that memorization is such a basic necessity for any intelligent program that it cannot be considered a separate learning process at all. An alternate view regards memorization as a complex subject that is vital to any effective cognitive system and well worth study and modeling on its own. This article takes a less extreme perspective, partly because the former viewpoint leaves nothing to say about rote learning and the latter would require more than is appropriate here. (See Chap. XI for a discussion of AI investigations into human memory processes.)

Rote memorization can be seen as an elementary learning process, not powerful enough to accomplish intelligent learning on its own (because not everything that needs to be known in any nontrivial domain can be memorized), but an inherent and important part of any learning system. All learning systems must remember the knowledge that they have acquired so that it can be applied in the future. In a rote-learning system, the knowledge has already been gained by some method and is in a directly usable form. Other, more sophisticated learning systems first acquire the knowledge from examples or from advice and then memorize it. Thus, all learning systems are built on a rote-learning process that stores, maintains, and retrieves knowledge in a knowledge base.

Rote learning works by taking problems that the performance element has solved and memorizing the problem and its solution. Viewed abstractly, the performance element can be thought of as some function, f, that takes an *input pattern* $(X_1, \ldots, X_n)$ and computes an *output value* $(Y_1, \ldots, Y_p)$. A rote memory for f simply stores the *associated pair* $[(X_1, \ldots, X_n), (Y_1, \ldots, Y_p)]$ in memory. During subsequent computations of $f(X_1, \ldots, X_n)$, the performance element can simply retrieve $(Y_1, \ldots, Y_p)$ from memory rather than recomputing it. This simple model of rote learning is depicted in Figure B1-1.

Consider, for example, an automobile insurance program that determines the cost of repairs for damaged automobiles. The input pattern is a description of the damaged automobile, including make and year, and a list of the damaged portions of the car. The output value is the estimated cost of the repairs. The system has only a rote memory. To estimate the cost of repairs, it looks in its memory for a previous automobile of the same make, model,

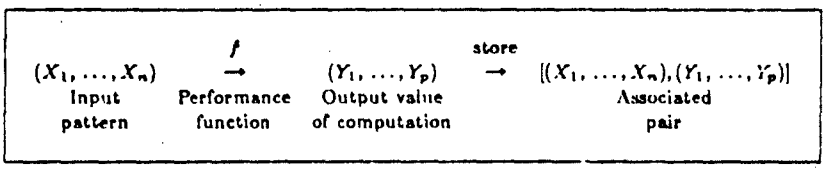


Figure B1-1. Simple model of rote learning.

and damage description and retrieves the corresponding cost. If it cannot find such an automobile, it uses a set of rules (published by a consortium of insurance companies) to guess the cost of the repairs and then saves its estimate for future use. This computed estimate, along with the description of the damaged automobile, forms the associated pair that is memorized.

Lenat, Hayes-Roth, and Klahr (1979) provide an interesting perspective on rote learning. They point out that rote learning (or "caching") can be viewed as the lowest level of a hierarchy of *data reductions*. The reductions are analogous to *computer language compilation:* The purpose is to refine the original information down to the essentials for performance. In rote learning, we generally attempt to save the input/output details of some calculation and so bypass a future need for the intermediate computation process. Thus, a *calculation* task, if valuable and stable enough to be remembered, is reduced to an *access* task (see Fig. B1-2, below).

Just as calculavions can be reduced to retrievals by caching, so can other inferential processes be reduced to simpler tasks. For instance, *deductions* can be reduced to *calculations*. The first time we are asked to solve a quadratic equation, for example, we must follow lengthy deductive chains to find the quadratic formula. Subsequently, we can simply compute the roots of a quadratic equation directly from the formula. We have distilled the results of a deductive search and summarized them as an efficient algorithm. Going one step further, the process of induction can convert a huge body of training instances into a single heuristic rule. Once again, the primary gain is in efficiency: It is no longer necessary to consult a huge body of examples to find out how to behave in a new situation.

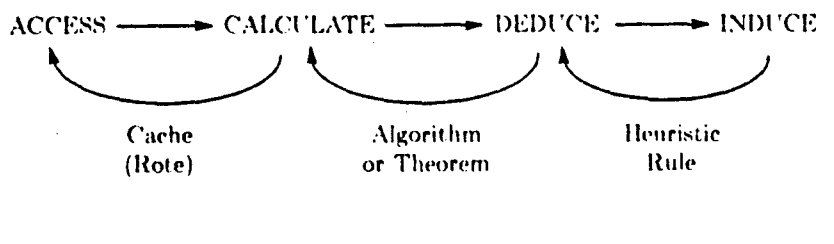


Figure B1-2. Spectrum of data reductions (from Lenat et al., 1979).

Issues in the Design of Rote-learning Systems

There are three important issues relevant to rote-learning systems: memory organization, stability, and the store-versus-compute trade-off.

Memory organization. Rote learning is useful only if it takes less time to retrieve the desired item than it does to recompute it. Retrieval can be made very rapid by properly organizing memory. Consequently, indexing, sorting, and hashing techniques have been thoroughly studied in the computer science subfields of data structures (Aho, Hopcroft, and Ullman, 1974) and database systems (Wiederhold, 1977; Date, 1977; Ullman, 1980).

Stability of the environment and the frame problem. Rote learning is not very helpful or effective in a rapidly changing environment. One important assumption underlying rote learning is that information stored at one time will still be valid later. If, however, the information changes frequently, this assumption can be violated. Consider, for example, information gathered about automobile repair costs during the early 1950s. Such information would be of little value for estimating automobile repair costs in the 1980s because the world has changed in critical ways: The makes and models of cars presently manufactured did not exist in the 1950s; furthermore, inflation has made the direct comparison of dollar costs impossible. A rote-learning system must be able to detect when the world has changed in such a way as to make stored information invalid. This is an instance of the *frame problem* (see Chap. III, in Vol. I).

Some solutions to this problem have been developed. One approach is to monitor every change to the world and keep the stored information always up to date. Thus, when an old model of automobile is discontinued, all information about that model could be removed from the knowledge base. This approach requires that the relevant aspects of the world be continually monitored.

A second approach to solving the frame problem is to check, when the information is retrieved for use, that it is still valid. Typically, this requires storing, along with the information itself, some additional data about the state of the world at the time the information was memorized. When the information is retrieved, the stored state can be compared to the current state, and the system can determine whether or not the information is still valid. This approach requires that the relevant aspects of the world (such as the current value of the dollar) be anticipated and stored with the data.

Many other approaches are possible. If the system can determine *how* the world has changed (e.g., by knowing the inflation rate), it may be able to make appropriate modifications to restore the validity of the memorized information (e.g., by converting the 1950 prices into 1980 equivalents).

Store-versus-compute trade-off. Since the primary goal of rote learning is to improve the overall performance of the system, it is important that the rote-learning process itself does not decrease the efficiency of the system.

It is conceivable, for instance, that the cost of storing and retrieving the memorized information is greater than the cost of recomputing it. This is certainly the case with the multiplication of two numbers; virtually all computers recompute the product of two numbers rather than store a large multiplication table.

There are two basic approaches to resolving the store-versus-compute trade-off. One is to decide at the time the information is first available whether or not it should be stored for later use. A cost-benefit analysis can be performed that weighs the amount of storage space consumed by the information and the cost of recomputing it against the likelihood that the information will be needed in the future. A second approach is to go ahead and store the information and later decide whether or not to forget it. This procedure, called *selective forgetting*, allows the system to determine empirically which items of information are most frequently reused.

One of the most common selective-forgetting techniques is called the *least recently used* (LRU) replacement algorithm. Each item stored in memory is tagged with the time when it was last retrieved. Every time an item is retrieved, its "time of last use" is updated. When a new item is to be memorized, the least recently used item is forgotten and replaced by the new one. Variations on this scheme take into consideration the amount of storage required for the item, the cost of recomputing the item, and so on.

References

Lenat, Hayes-Roth, and Klahr (1979) provide an excellent discussion of various learning methods, including rote learning. Samuel (1959) remains the best example of research into rote processes.

B2. Rote Learning in Samuel's Checkers Player

SAMUEL conducted a series of studies (1959, 1967) on how to get a computer to learn to play checkers. Among the earliest investigations of machine learning, they remain some of the most successful both in terms of improved performance (i.e., demonstrated improvements in the performance element) and in terms of lessons for AI. His experiments with three different learning methods—rote learning, polynomial evaluation functions, and signature tables—showed that significant improvement in playing checkers could be obtained. This article focuses on his thorough analysis of the question of how much rote learning alone can contribute to expertise and improved performance. Other aspects of Samuel's work are discussed later in Article XIV.D4a.

The Game of Checkers as a Performance Task

Checkers is a difficult game to play well. It is estimated that a full exploration of all possible moves in checkers would require roughly 10^{40} moves. Samuel's program was provided with procedures for playing the game *correctly;* that is, the rules of checkers were incorporated into the program. He sought to have the program learn to play *well* by having it memorize and recall board positions that it had encountered in previous games.

At each turn, Samuel's program chose its move by conducting a *minimax game-tree search* (see Articles II.B3 and II.C5, in Vol. I). In principle, of course, a program could try all possible moves and all possible consequences of each move and thereby search the entire checkers game-tree. Such a calculation—which is equivalent to playing every possible game of checkers—is not feasible because the search space is too large. Every potential move by one player generally leads to many possible countermoves, each of which has still more possible responses. The resulting combinatorial explosion (see Article II.A, in Vol. I) prevents any program from searching the whole tree.

Consequently, the standard approach to conducting a game-tree search is to search only a few moves (and countermoves) into the future and then apply a *static evaluation function* to estimate which side is winning. The program then chooses the move that leads to the best estimated position.

Suppose, for example, that at some board position, A, it is the program's turn to move (see Fig. B2-1). The program searches ahead three moves by considering first all possible moves that it could make, then all possible countermoves available to its opponent, and finally all possible replies to those countermoves. At this point, the program applies a static evaluation function to estimate its net advantage at each of the board positions shown on the right in the figure. These values are then "backed up" by assuming that

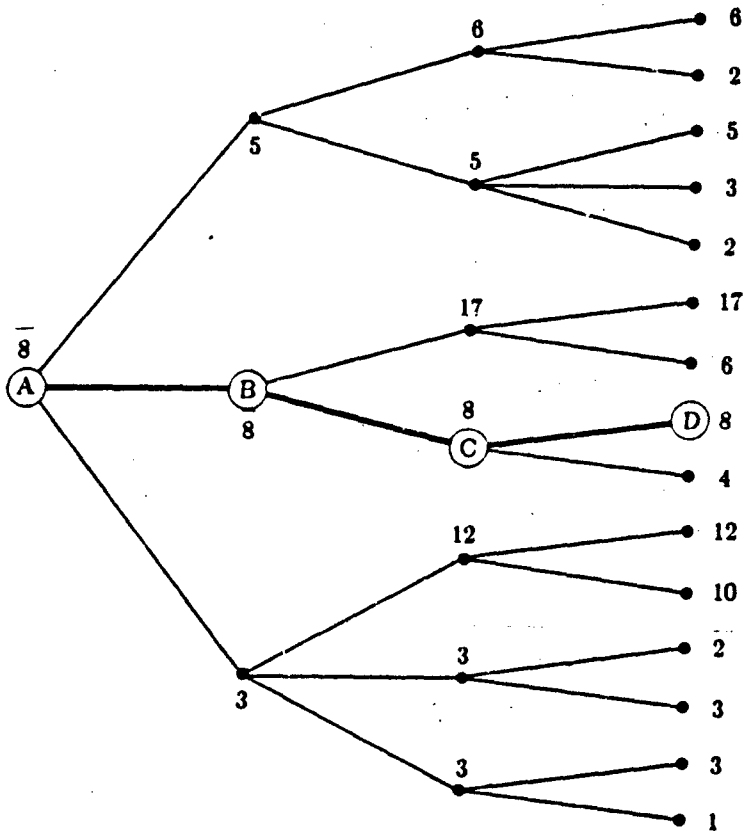


Figure B2-1. An example of a minimax game-tree search.

the opponent will always take the move that is worst for the computer (and vice versa). Thus, the best move for the program is the one that leads to position *B*. The program expects that the opponent will countermove to *C*, to which the program can reply with *D*. The static evaluation function has estimated the value of *D* to be 8, so this is the backed-up value of position *A*.

Improving the Performance of the Checkers Player

There are two basic ways to improve the performance of a game-tree search. One method is to search farther into the future and thus better approximate a full search of the tree. This is known as improving the *look-ahead power* of the program. The other method is to improve the static

evaluation function, so that the estimated value of each board position is more accurate. Samuel's rote-learning studies aimed at improving the look-ahead power by memorizing the backed-up values of board positions. The techniques discussed in Article XIV.D5a address the problem of improving the evaluation function.

The rote-learning approach employed by Samuel saved every board position encountered during play, along with its backed-up value. In the situation shown in Figure B2-1, for instance, Samuel's program would memorize the description of board position A and its backed-up value of 8 as an associated pair, $[A, 8]$. When position A is encountered in subsequent games, its evaluation score is retrieved from memory rather than recomputed. This makes the program more efficient, because it does not have to compute the value for A with the static evalution function.

There is a more important benefit of retrieving the backed-up value of A from memory, however. The memorized value of A is more accurate than the static value of A, because it is based on a look-ahead search. Thus, the look-ahead power of the program is improved. Figure B2-2 shows an example of this improvement. The program is considering which move to make at position E. It searches ahead three moves and then applies the static evaluation function. For position A, however, the program is able to retrieve the memorized value based on the previous search to position D.

This approach improves the effective search depth for E. As more and more positions are memorized, the effective search depth improves from its

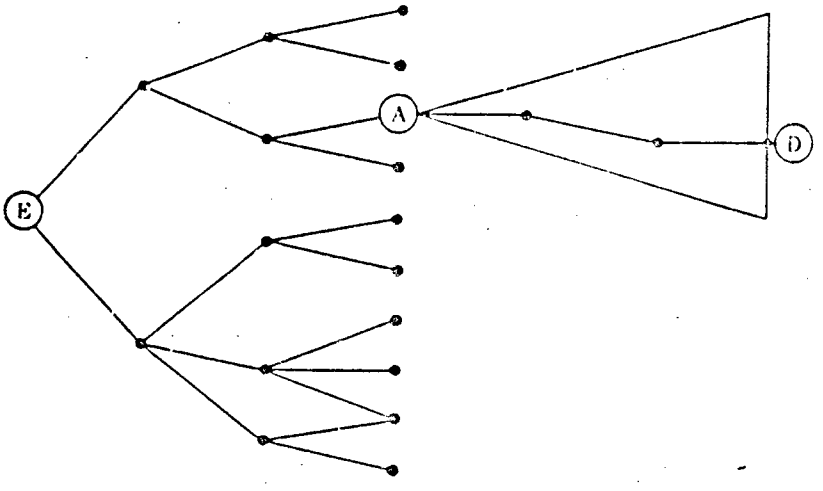


Figure B2-2. Improving look-ahead power by rote learning.

original value of 3 moves, up to 6, then to 9, and so on. Rote learning is thus used in Samuel's program to save the results of previous partial game-tree searches, so that they can gradually be extended and deepened. Rote learning converts a computation (tree search) into a retrieval from memory.

Memory Organization

Samuel employed several clever techniques to store the evaluated board positions, so that they took up little space and could be retrieved rapidly. To store the positions compactly, Samuel took advantage of several symmetries (e.g., positions in which it was Red's turn to move were converted into the corresponding Black-to-move positions; king positions are symmetric in two ways). Efficient retrieval was accomplished by indexing the boards according to many different characteristics (including the number of pieces on the board, presence or absence of kings, and piece advantage) and writing them onto a tape in the order they would most likely be needed during a game. The use of magnetic tape was necessary because the program was running on a relatively small IBM 704 computer, and only a few board positions could be kept in the computer's core memory. During rote learning, the program would accumulate a number of board positions before reading, sorting, and rewriting them onto the memory tape.

Samuel resolved the store-versus-compute trade-off with a variation of *least recently used* (LRU) replacement. Each board position was given an age. Whenever a position was retrieved from memory, its age was divided by 2. When the memory tape was rewritten, the ages of all stored positions were increased by 1, and very old positions were forgotten—that is, not written back onto tape.

Results

The program was trained in several ways: by playing against itself, by playing against people (including some checkers masters), and by following published games between master players (so-called book games). After training, the memory tape contained roughly 53,000 positions. As the program learned more, it improved slowly but steadily, becoming, in Samuel's words, a "rather better-than-average novice, but definitely not ... an expert" (Samuel, 1959, p. 218). Success in learning varied markedly depending on the phase of the game. The program became capable of playing a very good opening game, since the number of board variations is relatively small near the start of the game. Performance during the midgame, with its far greater range of possible configurations, did not greatly improve with rote learning. During the end game, the program became able to recognize winning and losing positions well in advance, but it needed some improvement before it was able to force the game to a successful conclusion (see below).

On the whole, Samuel's experiments demonstrated that significant and measurable learning can result from rote processes alone, but that on its own, rote learning is limited in several ways. The first and most obvious limitation is in storage space and retrieval. One question that interested Samuel is the following: If rote learning produces steady improvement of performance as it gathers new positions (up to a limit determined by available space and the efficiency of indexing algorithms), could it ever reach a performance level considered expert before exceeding the storage and indexing limits? If so, how much data would it need to remember, and how long would it take to gather the data?

Samuel estimated that his program would need to memorize about one million positions to approximate a master level of checkers play. Unfortunately, even a system with sufficient storage capacity and rapid retrieval methods would require an impractical amount of machine playing in order to gather a million useful positions. However, Samuel suggests that even this long acquisition period would be shorter than the time taken by humans to improve from complete beginners to masters.

The inability of the program actually to effect a win once it had a winning position was a curious problem. It was caused by the *mesa effect* (Minsky, 1963)—that is, once the program has found a winning position, all moves look equally good, and the program tends to wander aimlessly. Samuel solved the problem by storing, along with each board position and value, the length of the search path that was used to compute the board value. The move-selection procedure was modified to select the best move that also had the shortest associated search distance. This change gave the program a sense of direction, so that it was able to press forward to win the game (or stall as much as possible to avoid losing a game).

Another interesting problem arose when Samuel attempted to combine rote learning with learning techniques that modified the static evaluation function. Unfortunately, changes to the evaluation function tended to invalidate previously memorized positions (see Article XIV.B1, on the frame problem). Samuel's solution was to avoid this problem by postponing rote learning until the evaluation function had been effectively learned.

Conclusion

Besides showing that real improvement of performance could be gained by the conceptually simplest form of learning—rote memorization—Samuel identified and elaborated several issues that need to be handled if rote is to offer significant gains. In general, the value of rote learning is to gain problem-solving power in the form of speed. By retrieving the stored results of extensive computations, the program can proceed deeper in its reasoning. The price is storage space, access time, and effort in organizing the stored knowledge.

Samuel found that for rote learning to be effective, knowledge had to be carefully organized for efficient retrieval, stabilized to avoid using values whose meanings had changed, augmented with search-depth information, and selectively forgotten so that only the most useful information would tend to be saved. In the case of Samuel's checkers player, rote learning may have had enough power on its own to lead eventually to expert performance, but the time and space required for that much improvement were beyond the available resources.

References

Samuel (1959) describes the rote-learning research in detail.

C. LEARNING BY TAKING ADVICE

C1. Issues

IN ONE of the earliest AI papers on learning, McCarthy (1958) proposed the creation of an advice-taking system that could accept advice and make use of it to plan and execute actions in the world. Until the late 1970s, however, there were very few attempts to write programs that could *learn by taking advice*. The recent emphasis in AI on expert systems has focused new attention on the problem of converting expert advice into expert performance (see Barr, Bennett, and Clancey, 1979).

Research on advice-taking systems has followed two major paths. One approach has been to develop systems that accept abstract, high-level advice and convert it into rules that can effectively guide the performance element. This research seeks to automate all phases of the advice-taking process. The other approach has been to develop sophisticated tools—such as knowledge-base editing and debugging aids--that make it easier for the expert to transform his own abstract expertise into detailed rules. In this second approach, the expert is an integral part of the learning system, detecting and diagnosing bugs and repairing and refining the knowledge base. The former approach shows promise of eventually developing completely instructable systems, while the latter approach has proved invaluable for creating knowledge-based expert systems. This article describes both of these research paths. We will discuss the more highly automated approach first and return later to the research on knowledge-base editing and debugging aids.

Steps for Automatic Advice-taking

Hayes-Roth, Klahr, and Mostow (1980, 1981) provide an outline of the processes required to convert expert advice into program performance. This outline can be summarized as follows:

1. *Request*—request advice from expert,
2. *Interpret*—assimilate into internal representation,
3. *Operationalize* - convert into usable form,
4. *Integrate* -integrate into knowledge base,
5. *Evaluate*—evaluate resulting actions of performance element.

Request. The first step is for the program to request advice from the expert. The request can be simple—just asking the expert to give some

general advice --or it can be sophisticated--identifying a shortcoming in the knowledge base and asking the expert how to repair it. Some systems are completely passive and simply wait for the expert to interrupt them and provide advice, while others are very careful to focus the attention of the expert on a particular problem.

Interpret. The next step in advice-taking is to accept the advice and represent it internally. McCarthy (1958) points out that in order for a program to accept advice, the program must have an *epistemologically adequate* representation for the advice (see Article III.C1, in Vol. II, that is, a representation that is capable of expressing the advice without losing any information. This interpretation step can be very difficult if the advice is given in a natural language. The program must understand the natural language sufficiently well to convert it into an unambiguous internal representation. See Chapter IV, in Volume I, for a detailed survey of AI research into natural-language understanding.

Operationalize. Once the advice has been accepted and interpreted into an unambiguous representation, it still may not be directly executable by the performance element. The third step--*operationalization* --seeks to bridge the gap between the level at which the advice is provided and the level at which the performance element can apply it.

Mostow's (1981) program FOO, for example, accepts advice about how to play the card game of Hearts. English-language advice, such as "Avoid taking points," is interpreted by FOO's human user and given to the program as the lambda-calculus statement (AVOID (TAKE-POINTS ME) (CURRENT TRICK)). However, even though this advice has been interpreted into an unambiguous internal representation, it is still not *operational* since FOO has no procedures or methods to avoid taking points. FOO *does* have methods for selecting and playing cards, however. Thus, the advice must be converted into a form, such as [ACHIEVE (LOW (CARD OF ME))] (i.e., "Play a low card"), that requires only these operations.

FOO accomplishes this task by applying many different operationalization methods (see Article XIV.C2). It tries to re-express the advice, using known relationships, until it can recognize that one of its operationalization methods is applicable. These methods then allow it to develop a procedure for carrying out all or part of the advice. The steps of reformulating the advice and applying operationalization methods are repeated until the advice is completely executable.

This process is similar to the approach taken by automatic-programming systems that convert high-level program specifications into efficient implementations (see Chap. X, in Vol. I). However, unlike those systems, which seek to create provably correct programs, FOO is not foolproof. The gap between the advice and the performance element is usually too wide, and the operationalization methods are usually too weak, to permit error-free operationalization.

For example, it is often necessary for FOO to make assumptions and approximations in order to transform the advice. FOO cannot always successfully "avoid taking points" in Hearts, since it is impossible for the program to know the contents of its opponents' hands. Instead, FOO applies heuristic methods to reduce the likelihood that points will be taken. Its strategy of playing low cards is, consequently, a tentative *hypothesis* about how to avoid taking points. The tentative hypotheses developed by operationalization must be tested and debugged before they can be accepted.

Integrate. When knowledge is added to the knowledge base, care must be taken to see that it is properly integrated (see Article XIV.A). New advice can result in new mistakes if it takes precedence over previous knowledge in situations in which the old knowledge is still correct. Yet the new advice must take precedence in the intended situations. The learning program must know enough about how the performance element applies the knowledge to be able to anticipate and avoid any bad side-effects that could result from adding the knowledge to the knowledge base.

Two common problems of integration are (a) overlapping applicability and (b) contradictory recommendations. Consider an expert system, such as MYCIN, whose knowledge base is represented as a set of production rules. When a new rule is added, its left-hand side (or condition part) may be overly general, causing it to trigger in situations in which some other rule is properly applicable. One solution to this problem is to specialize the rules, so that this overlap of applicability no longer occurs. Another approach—the *meta-rule approach*—is to add ordering rules (*meta-rules*) that explicitly indicate which regular rules should be applied before others.

When the right-hand sides (or action parts) of two production rules recommend inconsistent actions in the same situation, the problem of contradictory recommendations arises. Again, either the right-hand sides can be modified to remove the contradiction or a meta-rule can be added to indicate which action should take precedence. There are many other integration problems aside from these two typical ones.

Evaluate. Since the new knowledge received from the expert is only tentative—that is, it is the result of interpretation, operationalization, and integration—it must be evaluated somehow. The learning system may be able to recognize some errors and inconsistencies in the advice when it integrates the advice into the knowledge base. More frequently, however, it is necessary to test the advice empirically by actually employing it to perform some task and then assessing whether the system is working properly.

Evaluation requires some performance standard against which the actual behavior of the system can be compared. In some domains, the performance standard can be built into the program. Game-playing programs, for example, can tell if the system is doing well by whether or not the system wins the game. In other domains, however, the system needs to set up detailed expectations

about how the new knowledge will affect the performance of the system. These expectations allow the program to detect and locate bugs in the knowledge base.

Evaluation can naturally feed back into the *request* step (the first of these five steps). When the program detects that the performance element is not functioning properly, it can announce this to the expert and request additional advice. A more sophisticated approach is for the program to do *credit assignment*—that is, to determine which parts of the knowledge base are incorrect. Once the bug has been located, the advice-taking system can ask the expert to tell it how to repair the particular piece of knowledge that is incorrect.

Now that we have discussed the five basic steps in an advice-taking system, we describe some systems that have been developed as aids for creating, modifying, and debugging large knowledge bases.

Aids for Knowledge-base Maintenance

Instead of fully automating these five steps, many researchers working on expert systems have built tools for assisting in the development and maintenance of expert knowledge bases. EMYCIN (van Melle, 1980; Davis, 1976), AGE (Nii and Aiello, 1979), and KAS (Reboh, 1981), for example, all provide certain functions to assist a domain expert or knowledge engineer in carrying out these five steps. Particular assistance has been provided for integrating new knowledge into the knowledge base (intelligent editors, flexible representation languages) and for evaluating and debugging the knowledge base (explanation and tracing facilities). This semiautomated approach to advice-taking places the knowledge engineer in the role of requesting, interpreting, and operationalizing the expert's advice.

To assist the knowledge engineer, these systems must be able to communicate effectively. It is particularly important for the engineer to get good feedback from the system during testing and debugging. Thus, a great deal of effort has been expended on the development of tracing and explanation facilities for expert systems (see Article VII.B, in Vol. II; Davis, 1976).

Conclusion

Research on advice-taking systems is still in its infancy, although important ideas and methods are available from the related areas of natural-language understanding and automatic programming. Present research is advancing along two paths: the theoretical path of automatic operationalization of expert advice and the practical path of providing aids to help knowledge engineers build and debug expert systems. The development of fully automatic systems remains an active research area.

A few AI systems have been developed that perform some kind of advice-taking. Mostow's FOO system is described in Article XIV.C2. The reader is also directed to the articles on TEIRESIAS (Article VII.B, in Vol. II) and on Waterman's poker player (Article XIV.D5b) for other examples of advice-taking systems.

References

Davis's work (1976, 1978) describes pioneering efforts in interactive advice-taking. Hayes-Roth, Klahr, and Mostow (1981) and Mostow and Hayes-Roth (1979) present the most comprehensive analyses of advice-taking as a whole.

C2. Mostow's Operationalizer

A GROUP of researchers at the Rand Corporation, Carnegie-Mellon University, and Stanford University has recently been developing the *machine-aided heuristic programming* methodology in which a computer would be instructed to perform a new task in much the same way that a person is taught (see Hayes-Roth, Klahr, Burge, and Mostow, 1978; Hayes-Roth, Klahr, and Mostow, 1981). A central effort in this project is understanding the problem of *operationalization* (see Article XIV.C1). Mostow's program FOO (First Operational Operationalizer) is one of the first results of this work. It investigates principles, problems, and methods involved in converting high-level advice into effective, executable procedures.

Accepting Advice About the Game of Hearts

Mostow, in his research with FOO, has dealt primarily with operationalization problems taken from the card game of Hearts. The game is played as a sequence of tricks. In each trick, one player—who is said to *have the lead*—starts the trick by playing a card and each of the other players continues the trick by playing a card during his (or her) turn. If he can, each player must follow suit, that is, play a card of the same suit as the suit led. The player who played the highest valued card in the suit led takes the trick and any point cards contained in it. Every heart counts as one point, and the queen of spades is worth 13 points. The goal of the game is to avoid taking points. Hayes-Roth et al. (1978) provide a more complete explanation of the game.

Hearts is a game of partial information, with no known algorithm for winning. Although the possible situations in the game are extremely numerous, beginning players often hear general advice such as "Avoid taking points," "Don't lead a high card in a suit in which an opponent is void," and "If an opponent has the queen of spades, try to flush it." The task of the FOO program is to take such general advice and render it directly applicable by a performance program. This task can be viewed as a kind of *planning* task. A piece of advice, such as "Avoid taking points," can be viewed as a goal. The operationalization program must develop an executable plan for achieving that goal. What makes this advice difficult to operationalize, however, is that the goal can be ill-defined and unattainable. It is impossible, for example, always to avoid taking points. Instead, the program must develop approximate strategies. The advice-giver intends the goal to *suggest*, but not *specify*, the desired behavior.

FOO is not able to accomplish this advice-taking task unaided. First, it does not perform the interpretation step at all but, instead, relies on the

user to translate the English form of the advice into an unambiguous lambda-calculus representation. Second, FOO cannot perform the operationalization step without human assistance. Although FOO has a large knowledge base of transformation rules and an interpreter for applying those rules, it must be told by the user which rules to apply. The user must operate FOO by repeatedly selecting an appropriate rule and indicating which expression or subexpression should be transformed. Finally, FOO does not integrate the operational knowledge it develops into a knowledge base that could drive a Hearts-playing program. No performance element has been developed that could provide an empirical test of the operationalized knowledge. Despite these shortcomings, Mostow's work on FOO provides an in-depth analysis of the techniques required to perform operationalization.

The primary way in which advice is operationalized in FOO is by applying *operationalization methods*, such as heuristic search, the pigeonhole principle, and finding necessary or sufficient conditions. Mostow claims that this is precisely what knowledge engineers and AI researchers do when they are faced with a new problem to solve: They look in their bag of tricks for a method, such as worst-case analysis, that allows them to construct an effective, but inefficient, program. This program can then be further refined by applying other knowledge and advice. Mostow's work can thus be viewed as formalizing the knowledge and techniques used by AI researchers to do heuristic programming.

The most sophisticated of FOO's operationalization methods is the *heuristic-search method*. When FOO needs to evaluate a predicate, such as (TAKE-POINTS ME), over a sequence, such as the sequence of cards in a trick, it is able to reformulate this problem as a heuristic search of the space of all possible tricks. FOO starts with a basic generate-and-test algorithm (discussed in Article II.A, in Vol. I) and refines it into a heuristic search by improving the ways the algorithm (a) selects the next node to expand, (b) selects possible expansions of the node to apply, (c) prunes nodes from the search tree, and (d) prunes possible expansions prior to applying them. The overall effect of these refinements is to move constraints from the *test* portion of the algorithm, that is, the step that checks to see whether the goal has been achieved, into the *generate* portion of the algorithm, that is, the step that chooses which nodes to expand and how they should be expanded. Some refinements actually move constraints out of the search altogether by precompiling them into tables or by modifying the algorithm to search a smaller space.

In the "Avoid taking points" problem, for example, FOO starts with a simple generate-and-test algorithm that generates all possible tricks and tests to see if ME (FOO's performance persona) takes any points. This is gradually converted into a heuristic search in which the only tricks considered are those in which ME plays a card higher than any card played so far in the suit led. Additional heuristics, such as generating tricks that contain points first and pruning tricks in which the opponents play cards higher than ME, are

extracted from the test and applied earlier in the search to order and prune the search tree.

Underlying all of FOO's operationalization methods is its basic ability to reformulate an expression in many different ways. For example, in order to evaluate (VOID P1 S1) (i.e., player P_1 is void in suit S_1), FOO must reformulate VOID in terms of observable variables such as the number of cards already played in the suit S_1. In order for FOO to *recognize* that an operationalization method is applicable, it must often do some reformulations. Then, in order actually to *apply* the method, FOO may need to do some further reformulations. The heuristic search method, for instance, is applicable only to a problem that is expressed as a search through some space. Consequently, in order to use heuristic search to operationalize the "Avoid taking points" advice, FOO must first reformulate the advice as a predicate over the search space of all possible tricks. The heuristic search can then search this space for those tricks that do not contain points.

The reformulation and operationalization process is accomplished by approximately 200 transformation rules (Mostow, in press). These rules employ analysis techniques and domain knowledge to successively reformulate the advice into an operational form. In this article, we trace a portion of FOO's operationalization of the "Avoid taking points" advice to show how these reformulation techniques are applied. Before doing this, however, we describe the knowledge that FOO has initially and how it is represented.

FOO's Initial Knowledge Base

FOO's performance knowledge is made up of *domain concepts*, plus *rules* and *heuristics* that are composed in terms of these concepts The advice offered to the program likewise consists of domain concepts, plus *compositions* of concepts. So as long as these compositions of basic concepts can be described in general ways, both the performance knowledge and the advice for building and improving it can be used and manipulated by domain-independent methods (see Hayes-Roth et al., 1981, for further discussion).

For example, in the domain of the card game Hearts, *basic concepts* include:

> deck, hand, card, suit, spades, deal, round, trick, avoid, point, player, play, take, lead, win, follow suit.

Examples of advice in the form of behavioral *constraints* include:

> The lead of the first trick is by the player with the 2C.
> Each player must follow suit if possible.
> The player of the highest card in the suit led wins the trick.
> The winner of a trick leads the next trick.

Advice in the form of *heuristics* includes:

If the queen of spades has not been played, then flush it out.
Take all the points in a round.
If you can't take all the points in a round, then take as few as possible.
If necessary, take a point to prevent someone else from taking them all.

A constraint such as "The lead of the first trick is by the player with the 2C" is represented as a composition, using domain-independent concepts like *first* and *with* and domain-dependent concepts like *lead*, *trick*, *player*, and *2C*.

An Example: Operationalizing "Avoid Taking Points"

After advice has been *interpreted* into an internal representation that is precise and unambiguous, it might be in an operational form, for example, "Play a low card." On the other hand, it may be far more general: "Avoid taking points." Experienced Hearts players will recognize that the first, specific piece of advice is a possible strategy for carrying out the latter, general advice. But it is a rather simplistic strategy, more appropriate for the later stages of a game than for the beginning. Furthermore, repeated attempts to play low cards will sometimes conflict with other advice. For purposes of illustration, however, operationalizing even a quite simple goal can require a wide range of knowledge and methods (see Mostow, 1981; Hayes-Roth et al., 1981). For the remainder of this article, several of the methods and problems of operationalization will be illustrated by showing how advice such as this can be converted into directly executable procedures.

First, consider how a person might handle advice such as "Avoid taking points." He might apply it to a specific situation by reasoning as follows:

1. To avoid taking points in general, I should avoid taking any points in the current *trick* (a single round in which one card is played by each player).
2. Thus, if the trick contains *points* (either a heart or the queen of spades), I should try not to win it.
3. I can do this by trying not to play the winning card.
4. That can be done by my playing a card lower than some other card played in the suit led.

Each step above is an attempt to implement the previous statement as closely as possible by restatement in successively more specific, operational terms. Some restatements may fully preserve the truth or accuracy of the previous one, while others may be very suppositional (i.e., valid given certain assumptions) or more restrictive (i.e., valid only in certain situations). The final statement above is not a very sophisticated plan, but it is at least a reasonable operationalization of the initial advice, and it represents a kind of process that seems very common in human learning. A *problem-reduction* strategy is employed until the advice can be applied directly in the given situation.

Now that we have a sense of how a person might operationalize "Avoid taking points," we trace the methods applied by FOO to accomplish this task. The following example is based on Derivation 6 in Mostow (1981) in which he guided FOO to reformulate "Avoid taking points" as "Play a low card." This particular trace shows the use of several simple operationalization and reformulation methods but does not show the application of the heuristic-search method discussed above.

To begin with, the advice must be *interpreted* into a tractable representational form, such as:

```
(avoid (take-points me) (trick))
```

That is, "Avoid the event in which ME takes points during the current trick." In FOO, this is done manually by the advice-giver.

A useful beginning in operationalization is to *elaborate* the original advice by expanding definitions (first of "avoid" and then of "trick"). The point is to unfold high-level terms so that the expression can be more easily manipulated. The results are

```
[achieve (not (during (trick) (take-points me)))]
```

and

```
(achieve (not (during [scenario
                       (each p (players)(play-card p))
                       (take-trick (trick-winner))]
                      (take-points me)))).
```

The advice in this form is still not operational, since it depends on the outcome of the trick, which is not generally knowable at the time ME needs to choose an action in accordance with the advice. Therefore, a *case analysis* is done on the subexpression (during...). The idea is to reformulate a single concept as several disjoint expressions that can be evaluated separately. To this end, the single (during...) expression is split into two expressions that depend on alternative assumptions. Here, taking points during the two-part "scenario" above can be considered as either of two possible cases: that taking points occurs during (a) the playing of cards or (b) the taking of the trick. The transformation results in:

```
(achieve (not (or [during (each p (players) (play-card p))
                          (take-points me)]
                  [during (take-trick (trick-winner))
                          (take-points me)]))).
```

The next transformation eliminates impossible cases. When expressions cannot be achieved because of impossible conditions, the learner should recognize this and drop them from consideration. Here, the first case can be ignored because there is no way to take points during the play of the cards (it is possible only after all players have played, when the trick is taken). FOO recognizes this by an *intersection search*. It searches through the knowledge

base of defined concepts for a common subevent of the two events (each p (players) (play-card p)) and (take-points me). Since no common subevent is found for these two, FOO concludes that the situation is an impossible one and eliminates it. (For the second case, take-trick and take-points have a common sub-event, take.) The advice now is:

```
(achieve (not [during (take-trick (trick-winner))
                      (take-points me)])).
```

The advice is still far from operational. One difficulty is that neither take-trick nor trick-winner is immediately evaluable at the time a card must be chosen for play. At this point, the problem can be reduced by reexpressing different concepts in *common terms*. This is possible here by again elaborating definitions and restructuring the subexpressions. Since take-points occurs during take-trick, the expression can be reformulated as:

```
(achieve (not [exists c1 (cards-played)
                  (exists c2 (point-cards)
                      (during (take (trick-winner) c1)
                          (take me c2)))])).
```

This says, "Make sure the situation does not happen where ME takes a point card (c2) during the time that the winner of the trick takes the cards played."

A process of *partial matching* recognizes that the two events in the during subexpression are closely related and thus are candidates for simplification, depending on the constraints of the during predicate. Using domain knowledge of relationships among the concepts, the terms can be combined and the subexpression made less complex. Instead of the complicated relation during, the events become joined by the far simpler predicates = and and. We now have:

```
(achieve (not (exists c1 (cards-played)
                  (exists c2 (point-cards)
                      [and (= (trick-winner) me)(= c1 c2)])))).
```

Further analysis at this point shows that *simplification* of some forms is possible. The central purpose of searching for simplifications is to *restructure* expressions to make them more amenable to further analysis. Examples of simplifying methods are deleting null clauses from a disjunction, transforming an expression into a constant (by evaluation), applying logical transformations (such as De Morgan's laws), or removing quantifiers when possible. The last of these methods is appropriate here, since c1 and c2 denote the same object: a point card. Thus with some reformulation employing domain knowledge, one variable can be replaced by the other, and the condition that they be equal can be dropped. The expression is transformed into:

```
(achieve (not [and (= (trick-winner) me)
                   (exists c1 (cards-played)
                       (in c1 (point-cards)))])).
```

Another kind of pattern-matching can accomplish another kind of simplification: By looking for canonical constructions, the operationalizer can *recognize known concepts*. If the form of a lower level expression fits the definition of a higher level concept, the former can be replaced by its simpler equivalent. (Note that this is the inverse of the first transformation mentioned above: expanding definitions.) In this case, the last two lines of the above expression match the definition of `trick-has-points`. This is analogous to the psychological process of *chunking*. In addition to all the analytical advantages gained by simplification, the recognition of known concepts can also enable the application of previously learned knowledge about them (e.g., ways to predict the likelihood that a trick will have points in it). Our expression is now reduced to not winning a trick that has points:

```
(achieve (not (and (= (trick-winner) me) [trick-has-points]))).
```

The expression is still not operational, since `trick-winner` is not generally knowable at the time of choosing which card to play. The concept of `trick-winner` is further analyzed, and, in fact, it takes about 20 further transformations to reformulate the above expression, "Try not to win a trick that has points," into "If you're following suit in a trick with points, try to play lower than some other card played in the suit led." Symbolically, this looks like:

```
(achieve (=> [and (in-suit-led (card-of me))
                  (trick-has-points)]
             [lower (card-of me)
                  (find-element (cards-played-in-suit-led))])).
```

But this still is not operational, since in general the set `cards-played-in-suit-led` is not fully known at the time that ME must choose a card. Since Hearts is a game of imperfect information, this set cannot generally be known, but the data available (cards already played) can be used to *approximate* the result. Here, the binary relation `lower` is approximated by the unary predicate `low`. In other words, in the absence of complete information for evaluating a comparative predicate (`lower x1 x2`), use instead an estimating function (`low x'`) that may not be exact but can produce a result from the available data. The approximation is:

```
(achieve (=> (and (in-suit-led (card-of me))
                  (trick-has-points))
             [low (card-of me)])).
```

This is now very close to being operational. `Low` is an imprecise term but can be treated as a *fuzzy* predicate (see Zadeh, 1979)—that is, it could be used to order potential candidates for the choice variable, `card-of me`.

The only remaining barrier to full operationality is the predicate (`trick-has-points`). This also is not always knowable at the time of choosing a card to play. However, further analysis leads to application of a rule that formulates an assertion as `possible` (effectively assuming it to be true) in the

absence of any knowledge to the contrary. Even when a predicate p is not evaluable, (possible p) will be.

Thus, the fully operational (though approximate) reformulation of the original "Avoid taking points" is "If following suit in a trick that may have points, play a low card." Again, the result may not always be the most effective action and may be in conflict with other advice. These are issues to be decided by the evaluating module of the learning element and by the performance element of the program. The symbolic form of the operationalized advice is:

```
(achieve (=> [and (in-suit-led (card-of me))
                  [possible (trick-has-points)]]
             [low (card-of me)])).
```

Conclusion

The example given above is a useful one because of the diversity of its reformulations, not because of any completeness. Among the most useful contributions of this research has been an introduction to the considerable complexity of operationalizing advice. Of the 13 examples of operationalized advice given in Mostow's thesis (1981), a couple required only a handful of transformations (a minimum of 8), but several required over 100. About 10 domain-independent transformational rules were mentioned in the example above, but over 200 such rules have been formulated and included in the system. Mostow (1981) gives a taxonomy of operationalization methods according to their *purpose*, *scope*, and *accuracy*. This taxonomy is outlined in Figure C2-1; each category is illustrated by one or more methods.

The greatest shortcoming of the work on FOO is the lack of a control structure that could apply these operationalization methods automatically. The development of such a control regime may be quite difficult. Mostow suggests using means-ends analysis (see Article II.D2, in Vol. I) and describes how his execution of rules often conformed to the following pattern:

1. *Reformulate* an expression until it is possible to
2. *recognize* that the method is applicable and decide to apply it, so
3. *reformulate* the expression to match the method problem statement and
4. *fill in* additional information required by the method; then
5. *refine* the instantiated method by applying additional domain knowledge.

A second shortcoming of FOO is that its methods are quite specific to the game of Hearts and similar tasks. The development of a general-purpose operationalization program will require the explication of many more operationalization methods. Still, these first steps in operationalization should prove valuable either for the overall project of *machine-aided heuristic programming* (see the beginning of this article) or for future efforts at implementing advice-taking systems.

1. **Methods for evaluating an expression**
   a. Procedures that always produce a result (assuming their inputs are available)
      "Pigeonhole principle"
      "Historical reasoning"
      "Heuristic search"
   b. Procedures that sometimes produce a result
      "Check a necessary or sufficient condition"
      "Make a simplifying assumption that restricts the scope of applicability"
   c. Procedures that produce an approximate result
      "Apply formula for probability that randomly chosen subsets overlap"
      "Characterize a quantity as an increasing or decreasing function of some variable"
      "Use an untested simplifying assumption"
      "Predict others' choices pessimistically"
2. **Methods for achieving a goal**
   a. Sound methods (introduce no errors)—execution of plan (when feasible) will achieve goal
      "To empty a set, remove one element at a time"
      "Find a sufficient condition and achieve it"
      "Restrict a choice to satisfy the goal"
      "Modify a plan for one goal to achieve an additional goal"
      "To achieve a goal with a future deadline, satisfy it now and then avoid violating it"
   b. Heuristic methods—execution of plan may not always achieve goal
      "Simplify the goal by arbitrarily choosing a value for one of its variables"
      "Find a necessary condition and achieve it"
      "Order choice set with respect to goal"

Figure C2-1. Taxonomy of operationalization methods.

References

Mostow (1981) is the most comprehensive description of FOO. The articles by Hayes-Roth, Klahr, and Mostow (1980, 1981) and by Hayes-Roth, Klahr, Burge, and Mostow (1978) provide a good overview of the idea of machine-aided heuristic programming. Mostow (in press) describes the work on heuristic search.

D. LEARNING FROM EXAMPLES

D1. Issues

THE PROSPECT of creating a program that can learn from examples has attracted the attention of AI researchers since the 1950s. McCarthy (1958, p. 78) said, "Our ultimate objective is to make programs that learn from their experience as effectively as humans do." Of course, the attainment of this goal still lies in the distant future. The area of learning from examples is, however, the best understood aspect of learning.

A program that learns from examples must reason from specific instances to general rules that can be used to guide the actions of the performance element. The learning element is presented with very low level information, in the form of a specific situation and the appropriate behavior for the performance element in that situation, and it is expected to *generalize* this information to obtain general rules of behavior.

Consider, for example, a program that is learning to play checkers. One way to train the program is to present it with particular checkers-board situations and tell it what the best moves are. The program must generalize from these particular moves to discover strategies for good play. Similarly, if we are teaching a program the concept of a *dog*, for example, we might present the program with various animals (and other things) and tell it whether or not they are dogs. The program must develop general rules for recognizing dogs and distinguishing them from everything else in the world.

Simon and Lea (1974), in an important early paper on induction, describe the problem of learning from examples as the problem of using training instances, selected from some space of possible instances, to guide a search for general rules. They call the space of possible training instances the *instance space* and the space of possible general rules the *rule space*. Furthermore, Simon and Lea point out that an intelligent program might select its own training instances by actively searching the instance space in order to resolve some ambiguity about the rules in the rule space. Thus, if the program were unsure whether all dogs have four legs, it might search the instance space for animals with different numbers of legs to see which ones are dogs. Simon and Lea view a learning system as moving back and forth between an instance space and a rule space until it has converged on the desired rule.

This *two-space view* of learning from examples as a simultaneous, cooperative search of the instance space and the rule space is a good perspective for organizing this article. We will use the terms *instance space* and *rule space* even in situations where the rule space does not contain rules but, instead,

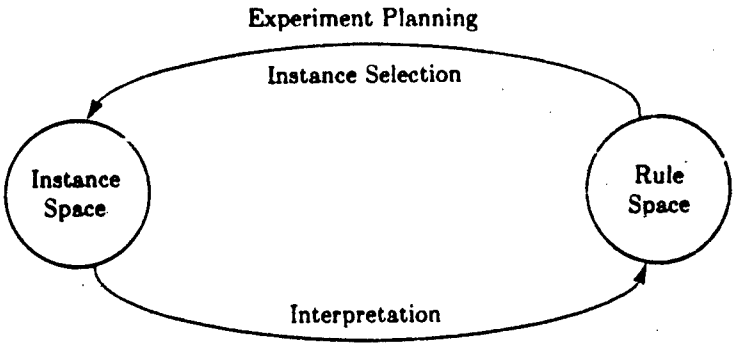


Figure D1–1. The two-space model of learning from examples.

contains some other high-level descriptions of the knowledge needed by the performance element.

Figure D1–1 shows a schematic diagram of the two-space model of learning from examples. In addition to the instance space and the rule space, the processes of *interpretation* and *experiment planning* are depicted. In some learning situations, the training instances are provided in a form far removed from the form of the rules in the rule space. As a result, when the program moves from the instance space to the rule space, special processes are needed to interpret the raw training instances so that they can guide the search of the rule space. Similarly, when the program needs to gather some new training instances, special experiment-planning routines are needed so that the current high-level hypotheses can guide the search of the instance space.

As an example of the two-space model, consider the problem of teaching a computer program the concept of a *flush* in poker (i.e., a hand in which all five cards have the same suit). The instance space in this learning problem is the space of all possible poker hands. We can represent an individual point in this space as a set of five ordered pairs, for example,

$$\{(2,clubs), (3,clubs), (5,clubs), (jack,clubs), (king,clubs)\}.$$

Each ordered pair specifies the rank and suit of one of the cards in the hand. The entire instance space is the space of all such five-card sets.

The rule space in this problem could be the space of all predicate calculus expressions composed of the predicates SUIT and RANK; the variables c_1, c_2, c_3, c_4, c_5 for the cards; any necessary free variables; the constant values of *clubs, diamonds, hearts, spades, ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, jack, queen,* and *king;* the conjunction operator ($\wedge$); and the existential quantifier ($\exists$). This rule space includes concepts such as ***contains at least three cards of the same rank:***

$$\exists\, c_1, c_2, c_3 : \text{RANK}(c_1, x) \wedge \text{RANK}(c_2, x) \wedge \text{RANK}(c_3, x),$$

and also the desired concept of a *flush:*

$$\exists\, c_1, c_2, c_3, c_4, c_5 : \text{SUIT}(c_1, x) \wedge \text{SUIT}(c_2, x) \wedge \text{SUIT}(c_3, x) \wedge \text{SUIT}(c_4, x) \wedge \text{SUIT}(c_5, x).$$

Note that this rule space does *not* contain the concept of a *straight.*

A learning program for searching these two spaces might operate as follows. First, the program selects a training instance from the instance space and asks the teacher whether it is an instance of the desired concept. This information (the instance and its classification) is converted by the interpretation procedures into a form that can help guide the search of the rule space. When some plausible candidate concepts are found in the rule space, experiment-planning routines decide which training instances should be examined next. If the learning program works properly, it will eventually choose, as its best candidate concept, the flush concept shown above.

Learning systems that employ the two-space approach are making use of the *closed-world assumption*, that is, the assumption that the rule space contains the desired concept. The closed-world assumption allows programs to locate the desired concept by progressively excluding candidate concepts that are known to be incorrect.

This two-space view of learning from examples helps to elucidate many of the design issues for learning systems. In this article, we follow this two-space model full circle. We examine, in turn, the issues concerning the instance space, the interpretation process, the rule space, and the experiment-planning process.

Instance Space

The first issue involving the instance space is the *quality* of the training instances. High-quality training instances are unambiguous and thus provide reliable guidance to the search of the rule space. Low-quality training instances invite multiple, conflicting interpretations and, consequently, provide only tentative guidance to the rule-space search.

Consider again the problem of teaching a program the concept of a *flush.* There are several sources of ambiguity that could make it difficult for the program to discover the concept from training instances.

First, the instances may contain errors. If the descriptions of the instances are incorrect, for example, if a 2 of clubs is incorrectly observed to be a 2 of spades, the error is a *measurement error.* If, on the other hand, the classification of the hand (as being a flush or not being a flush) is incorrect, the error is a *classification error.* Two kinds of classification errors can occur. The program can be told that a sample hand is a flush when in fact it is

not—a *false positive* instance—or that it is not a flush when in fact it is—a *false negative* instance.

A second source of ambiguity arises if the program must learn from *unclassified training instances*. In these so-called *unsupervised learning situations*, the program is given heuristic information that it must use to classify the training instances itself. If this heuristic knowledge is weak and imperfect, the rule-space search must treat the resulting classifications as being potentially incorrect.

A third factor relating to the quality of the training instances is the *order in which they are presented*. A good training sequence systematically varies the relevant features to determine which features are important. When a program is selecting training instances, it attempts to construct a good training sequence for itself. The task of learning is made much easier if there is a teacher who can be counted on to perform this function. In such cases, a program can reason about a puzzling instance by trying to infer "what the teacher was getting at" in presenting the example.

The main point, then, is that high-quality training instances are unambiguous. Under such favorable conditions, the program can be designed to embody a whole set of constraining assumptions about the examples that permit it to locate rapidly the appropriate high-level rules in the rule space. Low-quality instances, again, are ambiguous, because the program must consider a much larger space of hypotheses. Thus, if it is possible that the training instances contain errors, the program must consider the hypothesis that any given instance is incorrect due to either measurement error or classification error. In general, the more constraints a program can assume about the data, the more easily it can learn from them.

The second design issue concerning the instance space is the question of how it should be searched. This issue has not received much attention in AI research, since most work has assumed either that the instances are presented all at once or else that the program has no control over their selection. (See, however, Rissland and Soloway, 1980, for recent work on instance selection.) Programs that can update their hypotheses as additional training instances are selected (or are made available by the environment) are said to perform *incremental learning*. Programs that explicitly search the instance space are said to perform *active instance selection*.

Most methods of searching the instance space make use of a set, H, of hypotheses in the rule space that are currently believed by the program to be most plausible. One approach is to try to *discriminate* as much as possible among the alternatives within H. A training instance can be chosen that "splits H in half," so that half of the hypotheses can be ruled out when the new instance is obtained. Another approach is to choose the most likely hypothesis in H and try to *confirm* it by checking additional training instances (particularly instances with extreme characteristics). Using a confirmatory strategy, the learning system can determine the limits of applicability of the

hypothesis under consideration. A third approach, called *expectation-based filtering*, selects training instances that *contradict* the hypotheses in H (see Lenat, Hayes-Roth, and Klahr, 1979). The hypotheses in H are used to filter out those instances that are expected to be true (i.e., those that are consistent with H), so that the learning program can focus its attention on those instances in which its current hypotheses break down. Finally, an important consideration may be the size of H, or other computational costs associated with the learning process. In such cases, new instances may be selected to *minimize* these computational costs. For example, the program might try to rule out only one factor at a time in order to reduce the cost of comparing a drastically different training instance with each hypothesis in H.

Interpretation Processes

Once the training instances have been selected, they may need to be transformed before they can be used to guide the search of the rule space. This transformation process can be quite difficult, especially in perceptual learning tasks. Suppose, for example, that we wish to train a computer to recognize the concept of an arch constructed from toy blocks. The program will be presented with a line drawing of a scene involving a structure of blocks and told whether or not the scene contains an arch. Winston's (1970) program that solves this learning task (see Article XIV.D3a) makes extensive use of "blocks-world knowledge" to interpret the line drawing and extract a relational graph structure that indicates which blocks are resting on top of which other blocks, which blocks are touching, and so forth. These are the relations needed to express the concept of an arch.

Another learning program that performs extensive interpretation of the training instances is Soloway's (1978) BASEBALL system. The raw training instances are roughly 2,000 noise-free "snapshots" of a baseball game. The snapshots give the locations of the nine players on the two teams (e.g., `(AT P1 FIRST-BASE)`), the location of the ball, and the state of the scoreboard. The program is composed of a sequence of nine steps that employ various kinds of knowledge to interpret and generalize the training instances. The first three steps apply general knowledge about games to filter out periods of inactivity and focus on cycles of high activity. The next three steps apply knowledge about physics and about competition and cooperation to interpret these cycles of activity as competitive or cooperative episodes. To identify these episodes, the program must assign goals to the different players (e.g., `(WANT-TO-EXECUTE (AT P1 FIRST-BASE))`). It also guesses that the overall goal of an episode is that of the last action taken by a player. The final three steps search the rule space to discover generalized episodes and episode goals such as *hit* and *out*. These concepts are far removed from the original training instances, but because the previous steps have properly interpreted the data in terms of goals and actions, this rule-space search is easily accomplished.

The basic purpose of interpreting the training instances is to extract information that is useful for guiding the search of the rule space. This usually involves converting the raw training instances into a representational form that allows syntactic generalization to be easily accomplished (see below).

Rule Space

Two main issues are related to the rule space of high-level knowledge: What is the space, and how can it be searched? The rule space is usually defined by specifying the kinds of operators and terms that can be used to represent a rule. The designer of a learning system seeks to choose a rule space that is easy to search and that contains the desired rule or rules. In the sections that follow, we first discuss two factors that influence the choice of a representation language for the rule space: the kinds of inference supported by the representation and the single-representation trick. Then we survey the four methods for searching the rule space. We conclude the discussion of rule-space issues by examining problems that arise when the representation is found to be inadequate for expressing the desired rule or rules.

Syntactic rules of inference. Both the expressiveness of a representation and the ease of searching the rule space depend on the kind and complexity of the inferences supported by the representation. The most common inference process needed for learning from examples is *generalization*. We say that one description, A, is *more general than* another description, B, if A applies in all of the situations in which B applies and then some more. Thus, the set of situations in which A is relevant is a superset of the set of situations in which B is relevant. For example, the rule that *All ravens are black* is more general than the rule that *All one-eyed ravens are black*, since the set of all ravens strictly includes the set of one-eyed ravens. Often, a description A is more general than a description B because A places fewer constraints on any relevant situations. The *all ravens* rule omits the *one-eyed* constraint and, hence, is more general.

It is important to choose a representation for the rule space in which generalization can be accomplished by inexpensive syntactic operations. Predicate calculus, for example, is quite amenable to certain kinds of syntactic generalization. Below are some examples of syntactic rules of inference that accomplish generalization in predicate calculus. Some recent work in learning (Larson, 1977; Larson and Michalski, 1977; Michalski, 1980) has sought to identify rules of inference that are particularly useful in learning systems. It is important to note that these rules of inference do not preserve truth—the rules are inductive.

1. **Turning constants to variables.** Suppose we want a program to discover the concept of a flush in poker. We might give some training instances of the form:

Instance 1. $\text{SUIT}(c_1, \textit{clubs}) \wedge \text{SUIT}(c_2, \textit{clubs}) \wedge$
$\text{SUIT}(c_3, \textit{clubs}) \wedge \text{SUIT}(c_4, \textit{clubs}) \wedge$
$\text{SUIT}(c_5, \textit{clubs}) \Rightarrow \text{FLUSH}(c_1, c_2, c_3, c_4, c_5)$.

Instance 2. $\text{SUIT}(c_1, \textit{spades}) \wedge \text{SUIT}(c_2, \textit{spades}) \wedge$
$\text{SUIT}(c_3, \textit{spades}) \wedge \text{SUIT}(c_4, \textit{spades}) \wedge$
$\text{SUIT}(c_5, \textit{spades}) \Rightarrow \text{FLUSH}(c_1, c_2, c_3, c_4, c_5)$.

From these, the program could hypothesize the rule

Rule 1. $\text{SUIT}(c_1, x) \wedge \text{SUIT}(c_2, x) \wedge \text{SUIT}(c_3, x) \wedge \text{SUIT}(c_4, x) \wedge$
$\text{SUIT}(c_5, x) \Rightarrow \text{FLUSH}(c_1, c_2, c_3, c_4, c_5)$.

by replacing the atomic constants of clubs and spades by the variable x (where x stands for any suit).

2. **Dropping conditions.** Suppose again that we are teaching a program the concept of a flush, but now we present instances of the form:

Instance 1. $\text{SUIT}(c_1, \textit{clubs}) \wedge \text{RANK}(c_1, 3) \wedge$
$\text{SUIT}(c_2, \textit{clubs}) \wedge \text{RANK}(c_2, 5) \wedge$
$\text{SUIT}(c_3, \textit{clubs}) \wedge \text{RANK}(c_3, 7) \wedge$
$\text{SUIT}(c_4, \textit{clubs}) \wedge \text{RANK}(c_4, 10) \wedge$
$\text{SUIT}(c_5, \textit{clubs}) \wedge \text{RANK}(c_5, \textit{king})$
$\Rightarrow \text{FLUSH}(c_1, c_2, c_3, c_4, c_5)$.

In order to discover rule 1, the program must not only turn constants into variables, but it must also "forget" all of the RANK predicates, since rank is irrelevant. This can be accomplished by *dropping conditions*. Any conjunction can be generalized by dropping one of its conditions. We can view a conjunctive condition as a constraint on the set of possible instances that could satisfy the description. By dropping a condition, we are removing a constraint and generalizing the rule.

3. **Adding options.** A further way to generalize a rule is to add another option to the rule so that more instances may conceivably satisfy it. Suppose we are trying to teach a program the concept of a *face card* (i.e., jack, queen, or king). We might give examples of the form:

Instance 1. $\text{RANK}(c_1, \textit{jack}) \Rightarrow \text{FACE}(c_1)$.
Instance 2. $\text{RANK}(c_1, \textit{queen}) \Rightarrow \text{FACE}(c_1)$.
Instance 3. $\text{RANK}(c_1, \textit{king}) \Rightarrow \text{FACE}(c_1)$.

The program can discover the rule by forming the disjunction of the possibilities:

Rule 2. $\text{RANK}(c_1, \textit{jack}) \vee \text{RANK}(c_1, \textit{queen}) \vee \text{RANK}(c_1, \textit{king})$
$\Rightarrow \text{FACE}(c_1)$.

Notice that this decision to add options is a less drastic generalization than that of turning the jack, queen, and king constants into a single variable to get

Rule 3 (wrong). $\text{RANK}(c_1, y) \Rightarrow \text{FACE}(c_1)$.

An alternative to ordinary disjunction is what Michalski (1980) terms an *internal disjunction*. If we allow sets and set membership in our representation, we can express our instances as

Instance 1′. $\text{RANK}(c_1) \in \{jack\} \Rightarrow \text{FACE}(c_1)$.
Instance 2′. $\text{RANK}(c_1) \in \{queen\} \Rightarrow \text{FACE}(c_1)$.
Instance 3′. $\text{RANK}(c_1) \in \{king\} \Rightarrow \text{FACE}(c_1)$.

The generalization can then be expressed as

Rule 2′. $\text{RANK}(c_1) \in \{jack, queen, king\} \Rightarrow \text{FACE}(c_1)$.

This latter representation is more compact.

Similar rules of generalization can be defined for numerical representations that use a linear combination of features, as follows:

4. **Curve fitting.** Suppose a program is attempting to discover how the output, z, of a system is related to two inputs, x and y. The program is provided with training instances in the form of (x, y, z) triples that show the output of the system for particular values of the inputs:

Instance 1. $(0, 2, 7)$.
Instance 2. $(6, -1, 10)$.
Instance 3. $(-1, -5, -16)$.

By a curve-fitting technique, such as least-squares regression, the program fits the line

Rule 1. $z = 2x + 3y + 1$,

or, alternately, the ordered triple $(x, y, 2x + 3y + 1)$, to these data. This generalizes the relationship, so that it holds for many more (x, y, z) triples than just the three training instances. The program can now predict the z output for any values of the x and y inputs. This process is analogous to the turning-constants-into-variables generalization rule.

5. **Zeroing a coefficient.** The program can further generalize this relationship by zeroing the y coefficient and fitting a plane to the three training instances. In this case, it obtains

Rule 2. $z = 2.59x - 3.99$.

Alternately, the ordered triple is $(x, y, 2.59x - 3.99)$. (The y coordinate can be anything.) By giving y the coefficient of zero, the program has dropped it as a condition and reduced the dimensionality of the function $z = F(x, y)$ to make it $z = G(x)$. The program has decided that y is irrelevant to the value of z. The relationship now holds for an even larger set of (x, y, z) triples. This rule is analogous to the dropping-condition rule of generalization.

Notice that these rules of inference correspond to particular features of the representation language. For example, the method of turning constants

into variables makes use of free variables, the method of adding options uses the disjunction operator, and the coefficient-zeroing technique makes use of the multiplication operator. To the extent that the representation language has fewer of these features, fewer inference rules will be applicable and, consequently, the search of the rule space will be easier to accomplish. But since each of these language features contributes to the expressiveness of the representation, the designer of a learning system faces a trade-off between the increased expressiveness of the representation and the increased difficulty of searching the rule space.

The single-representation trick. Another factor relating to the difficulty of searching the rule space (and the instance space) is the difference between the representation used for rules and the representation used for the training instances. If the representations for the rule space and the instance space are far removed from each other, then the searches of the two spaces must be coordinated by complex interpretation and experiment-planning procedures. One trick commonly used to avoid this problem is to choose the same representation for both spaces. Training instances are viewed literally as highly specific pieces of acquired knowledge. Suppose, for example, that we are trying to teach a program the concept of a *pair* in poker. We want the program to learn the rule

Rule 4. $\exists\ card_1, card_2 : \text{RANK}(card_1, x) \wedge \text{RANK}(card_2, x) \Rightarrow \text{PAIR}.$

(This is only an approximate definition of PAIR. An exact definition would require a more complex representation involving equality.)

As was shown above, specific hands could be represented "naturally" as sets of five ordered pairs—the rank and suit of each of the cards. With such a representation for the hand made up of the 2 of clubs, 3 of diamonds, 2 of hearts, 6 of spades, and king of hearts, we would obtain

Instance 1. $\{(2, clubs), (3, diamonds), (2, hearts), (6, spades), (king, hearts)\} \Rightarrow \text{PAIR}.$

But this representation makes it difficult to discover the concept of a pair in poker with the syntactic rules of inference described above. A less natural, but more useful, representation would describe the hand in predicate calculus—the same representation that we will eventually need for the acquired concept (rule 4). Thus, we would say of our hand

Instance 1′. $$\begin{aligned}\exists\ c_1, c_2, c_3, c_4, c_5 : \ & \text{RANK}(c_1, 2) \wedge \text{SUIT}(c_1, clubs) \wedge \\ & \text{RANK}(c_2, 3) \wedge \text{SUIT}(c_2, diamonds) \wedge \\ & \text{RANK}(c_3, 2) \wedge \text{SUIT}(c_3, hearts) \wedge \\ & \text{RANK}(c_4, 6) \wedge \text{SUIT}(c_4, spades) \wedge \\ & \text{RANK}(c_5, \text{K}) \wedge \text{SUIT}(c_5, hearts) \Rightarrow \text{PAIR}.\end{aligned}$$

Now the process of generalization merely involves dropping the SUIT conditions and replacing the constant 2 by a variable x. Of course, there are many other possible generalizations of instance 1′, and the search of the rule space

would still be nontrivial. The advantage of using the single-representation trick is that we have chosen a representation that allows this search to be accomplished by simple syntactic processes.

The problems of interpretation and experiment planning are eased when the single-representation trick is used. Many learning programs sidestep these problems completely by assuming that the training instances are provided by the environment in the same representation as used for the rule space. In more practical situations, the interpretation and experiment-planning routines serve to translate between the *raw* instances (as they are received from the environment) and the *derived* instances (after they have been interpreted as specific points in the rule space).

Methods of searching the rule space. Now that we have discussed the issue of how to represent the rule space, we can turn our attention to the four main methods that have been used to search the rule space. All of these methods maintain a set, H, of the *currently most plausible rules*. They differ primarily in how they refine the set H so that it eventually includes the desired points in the rule space. A useful classification of search methods distinguishes methods in which the presentation of the training instances drives the search (so-called *data-driven methods*) from those methods in which an a priori model guides the search (so-called *model-driven methods*).

The first data-driven method is the *version-space method* (and several related techniques). This approach uses the single-representation trick to represent training instances as very specific points in the rule space. The set H is initialized to contain all hypotheses consistent with the first positive training instance. New training instances are examined one at a time and pattern-matched against H to determine whether the hypotheses in H should be generalized or specialized.

The second method, also a data-driven method, does not use the single-representation trick. Instead, special procedures (or production rules) examine the set of training instances and decide how to refine the current set, H, of hypotheses. The program can be viewed as having a set of *hypothesis-refinement operators*. In each cycle, it uses the data to choose one of these operators and then applies it. Lenat's (1976) AM system is an example of this approach.

The third approach is *model-driven generate and test*. This method repeatedly generates and tests hypotheses from the rule space against the training instances. Model-based knowledge is used to constrain the hypothesis generator to generate only plausible hypotheses. The Meta-DENDRAL program is the best example of this approach (see Buchanan and Mitchell, 1978).

Finally, the fourth approach is *model-driven schema instantiation*. It uses a set of *rule schemas* to provide general constraints on the form of plausible rules. The method attempts to instantiate these schemas from the current set of training instances. The instantiated schema that best fits the training instances is considered the most plausible rule. Dietterich's SPARC program

(Dietterich, 1979; Dietterich and Michalski, in press), which discovers secret rules in the card game Eleusis, applies the schema-instantiation method.

Data-driven techniques generally have the advantage of supporting incremental learning. A feature of the version space method, in particular, is that the H set can easily be modified to account for new training instances without any backtracking by the learning program. In contrast, model-driven methods, which test and reject hypotheses based on an examination of the whole body of data, are difficult to use in incremental learning situations. When new training instances become available, model-driven methods must either backtrack or search the rule space again, because the criteria by which hypotheses were originally tested (or schemas instantiated) have changed.

A strength of model-driven methods, on the other hand, is that they tend to have good noise immunity. When a set of hypotheses, H, is tested against noisy training instances, the model-driven methods need not reject a hypothesis on the basis of one or two counterexamples. Since the whole set of training instances is available, the program can use statistical measures of how well a proposed hypothesis accounts for the data. In data-driven methods, H is revised each time on the basis of the current training instance. Consequently, a single erroneous instance can cause a large perturbation in H (from which it may never recover). One approach that allows data-driven methods to handle noise is to make very slight, conservative changes in H in response to each training instance. This minimizes the effect of any erroneous training instances, but it causes the learning system to learn much more slowly.

The problem of new terms. In some learning problems, the program can assume that the desired rule or rules exist somewhere in the rule space. Consequently, the search has a well-defined goal. In many situations, however, there is no such guarantee, and the learning program must confront the possibility that its representation of the rule space is inadequate and should be expanded. This is called the problem of *new terms*.

One approach to expanding the rule space is to add new terms to the representation. Consider again the problem of teaching a program the concept of a *pair* in poker. In the section above, the program was able to represent the *pair* concept by using a predicate-calculus representation with the suit and rank terms. Such a representation would not permit the program to discover the concept of a *straight*, however. One way to represent the *straight* concept would be to create a new term called SUCC(x, y), which is true if and only if $x = y + 1$. Now the *straight* concept can be represented as:

$$\text{RANK}(c_1, r_1) \wedge \text{RANK}(c_2, r_2) \wedge \text{RANK}(c_3, r_3) \wedge \text{RANK}(c_4, r_4) \wedge \text{RANK}(c_5, r_5) \wedge \text{SUCC}(r_1, r_2) \wedge \text{SUCC}(r_2, r_3) \wedge \text{SUCC}(r_3, r_4) \wedge \text{SUCC}(r_4, r_5).$$

The problem of defining new terms is quite difficult to solve. An advantage of the hypothesis-refinement operator approach to searching the rule space is that it is fairly easy to incorporate operators that create new terms. The

BACON (Langley, 1980) and AM programs both have operators that create new terms by combining and refining existing terms.

Experiment Planning

Once the learning element has searched the rule space and developed a set, H, of plausible hypotheses, the program may need to gather more training instances to test and refine them. When the instance space and the rule space are represented in very different ways, the process of determining which training instances are needed and how they can be obtained can be quite involved. Suppose, for example, that a genetics learning program is attempting to discover which portions of DNA are important. To test a high-level hypothesis (or several hypotheses), it may be necessary to plan a very involved experiment to synthesize a particular strand of DNA and insert it into the appropriate bacterial cells to observe the resulting behavior of the cells.

The AM program is an example of an AI learning program that performs some experiment planning. After one of AM's refinement operators creates a new concept, AM must gather examples of that concept to evaluate and refine it. Several techniques are used to generate good training instances, for example, by symbolically instantiating the concept definition or by inheriting examples from more general or more specific concepts. AM has a special body of heuristics for locating positive and negative boundary examples (i.e., examples that barely succeed, or barely fail, to be instances of the concept).

Taxonomy of Work in Learning from Examples

Now that we have described the two-space model, we present a rough taxonomy of work in the area of learning from examples. Several subareas of research have developed within this area, ranging from philosophically oriented inductive learning to highly engineering-oriented pattern-classification work. These different areas can be characterized by two components of the simple learning model presented in Article XIV.A: the representation used in the knowledge base and the task that the performance element carries out. In the remainder of this chapter, a separate article is devoted to each of these subareas.

Systems that use numerical representations. Researchers in electrical engineering and systems theory have developed learning methods that represent acquired knowledge in the form of polynomials and matrices. The performance elements of these learning systems, which are usually called ***adaptive systems***, typically perform tasks such as pattern classification, adaptive control, and adaptive filtering. The strengths of these adaptive methods are that they can be used in noisy environments, in environments whose properties

are changing rapidly, and in situations where analytic solutions based on classical systems theory are unavailable. We include an article on this subject because of its historical relationship to AI and because of the possibility that useful hybrid systems may be constructed in the future.

Systems that use symbolic representations. Most AI work on learning has used symbolic representations such as feature vectors, first-order predicate calculus, and production rules to represent the knowledge acquired by the learning element. It is useful to classify this work according to the complexity of the task being performed by the learning system:

1. *Learning single concepts.* The simplest performance task is to classify new instances according to whether they are instances of a single concept. The problem of learning single concepts has received a lot of attention and is probably the best understood learning task in AI.

2. *Learning multiple concepts.* Many performance tasks involve the use of a *set* of concepts that operate independently. Disease diagnosis, for example, is a task in which the program seeks to assign one or more disease classes to a patient. The problem of learning a set of concepts has received some attention in AI. The Meta-DENDRAL and AM systems, for example, discover many concepts in order to describe their training instances and guide the performance element.

3. *Learning to perform multiple-step tasks.* The most complex performance tasks for which learning techniques have been developed are relatively simple planning tasks that require the performance element to apply a *sequence* of operators to perform the task. Unlike the multiple, but independent, concepts used in Meta-DENDRAL and AM, the rules in these systems must be chained together into a sequence. Consequently, many difficult problems of integration and credit-assignment arise.

References

Simon and Lea (1974) describe the two-space model of rule induction. Dietterich and Michalski (1981) provide some perspectives on systems that learn from examples. See also Buchanan, Mitchell, Smith, and Johnson (1977).

D2. Learning in Control and Pattern Recognition Systems

THERE ARE many applications in engineering and science for which learning systems have been developed. These systems, usually called *adaptive systems*, are useful when classical systems techniques cannot be applied because of insufficient knowledge about the underlying system. Such situations often arise in extremely noisy and rapidly changing environments.

Classical systems theory addresses itself to problems in the design and analysis of *systems*, where a system is viewed abstractly as an operator that maps a vector of inputs, $\mathbf{x}$, to a vector of outputs, $\mathbf{y}$. Two important engineering problems for which learning systems have been developed are *control* and *pattern recognition*.

Consider the control problem shown in Figure D2-1. The system is an automobile engine. The inputs—in this case, control inputs—are the amount of gasoline and the setting of the spark-plug advance. The single output is the speed of the engine. The control problem is to determine the settings of the inputs over time, so that the output follows a particular curve. We want the speed of the engine to track the desired speed as commanded by the driver of the automobile. If we have a mathematical model of the engine—say, as a set of differential equations relating x_1 and x_2 to y—we can often solve this control problem. To obtain the model, we can usually inspect the system directly and apply the laws of physics. But in complex, time-varying systems, such an approach may be impossible. Instead, it may be necessary to *identify* the system—that is, construct a model by observing the system in operation and finding an empirical relationship between the inputs and the outputs.

Pattern recognition—the other task for which adaptive learning is useful—also can be viewed as a system-identification problem. The pattern-classification system shown in Figure D2-2 takes an input object—represented as a vector, $\mathbf{x}$, of features—and maps it into one of m pattern classes. The

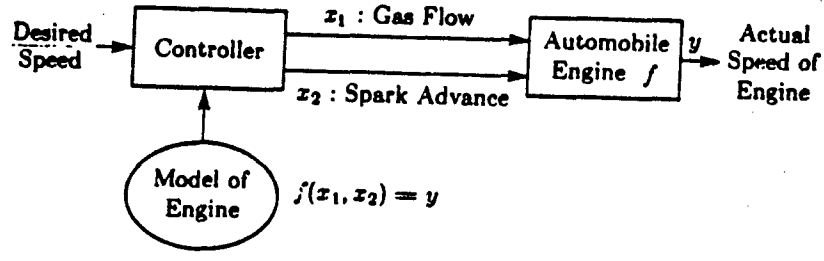


Figure D2-1. A simple control problem.

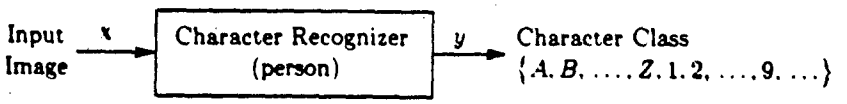


Figure D2-2. A simple pattern-classification problem.

archetypal pattern-classification problem is optical character recognition, in which the inputs are images of handwritten or printed characters and the output is a classification of each image as one of the letters, numerals, or punctuation symbols. Suppose we want to build a computer system that can recognize characters. We have available an unknown system—in this case, a person—that can perform the task reliably. If we can identify the system, we will then have a computer model that can recognize handwritten characters.

Figure D2-3 illustrates the general setup for adaptive system identification. The unknown system and the model are configured in parallel. Their outputs—the true output, $\mathbf{y}$, and the estimated output, $\tilde{\mathbf{y}}$—are compared, and the error, e, is fed back to the learning element, which then modifies the model appropriately. In the terminology of our simple learning-system model, the unknown system is the environment. It provides training instances, in the form of $(\mathbf{x}, \mathbf{y})$ pairs, to the learning element. The learning element modifies certain parts of the model (i.e., the knowledge base), so that the model system (i.e., the performance element) more accurately models the unknown system.

Conceptually, therefore, adaptive system identification, adaptive control, and pattern recognition are all problems of learning from examples. The

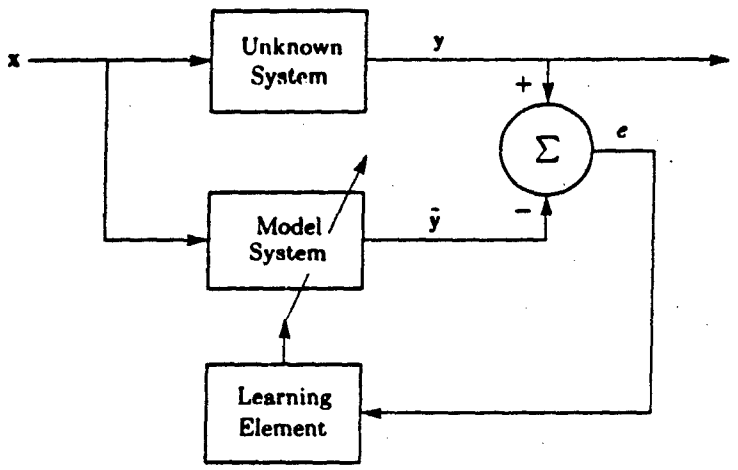


Figure D2-3. Adaptive system identification.

unknown system provides the training instances and the performance standard (i.e., the true y-values).

In this article, we discuss the methods that have been used to accomplish this learning. We have divided the methods into four groups according to the representations that are used to model the unknown system:

1. *Statistical algorithms*, which employ probability density functions to create a Bayesian decision procedure;
2. *Parameter learning*, which uses a vector of parameters and a linear model;
3. *Automata learning*, which uses stochastic and fuzzy automata (discussed below) to model the unknown system; and
4. *Structural learning*, which uses pattern grammars and graphs to represent classes of objects for pattern classification.

Statistical Learning Algorithms

In pattern recognition (and sometimes in control), it is possible to view the unknown system as making a decision to assign the input, $\mathbf{x}$, to one class, y, out of m classes. By defining a loss function that penalizes incorrect decisions (i.e., decisions in which $\bar{y}$ differs from y), a minimum-average-loss Bayes classifier can be used to model the unknown system. The problem of identifying the unknown system then reduces to the problem of estimating a set of parameters for certain probability density functions. These parameters, such as the mean vector and the variance-covariance matrix, can be estimated from the training instances in a fairly straightforward fashion (see Duda and Hart, 1973).

In the terminology of Simon and Lea (1974), the set of all possible $\mathbf{x}$ vectors forms the instance space, and the set of possible values for the parameters of the probability distributions forms the rule space. The rule space is searched by direct calculation from the training instances. The instance space is not actively searched.

Unfortunately, these methods rely on assuming a particular form (e.g., multivariate normal) for the probability distributions in the model. These assumptions frequently do not hold in real-world problems. Furthermore, the computational costs of the estimation may be very high when there are many features.

Parameter Learning

In parameter learning, a fixed functional form is assumed for the unknown system. This functional form has a vector of parameters, $\mathbf{w}$, that must be determined from the training instances. Unlike the statistical methods, there is little or no probabilistic interpretation for the unknown parameters and,

consequently, probability theory provides no guidance for estimating them from the data. Instead, some sort of criterion, usually the squared error $(y - \tilde{y})^2$ averaged over all training instances, is minimized. The rule space is thus a space of possible parameter vectors, and it is searched by hill-climbing (also called *gradient descent*) to find the point that minimizes the error between the model and the unknown system.

The most popular form assumed for the unknown system is a linear functional:

$$y = \mathbf{wx} = \sum_i w_i x_i \, .$$

The output is assumed to be a linear combination of the input feature vector, $\mathbf{x}$, with a weight vector, $\mathbf{w}$. The elements of the weight vector are the unknown parameters. The rule space is thus the space of all possible weight vectors, known as the *weight space*.

An important special case arises when the unknown system is a *binary* pattern classification system similar to the system shown earlier in Figure D2-2. In binary pattern classification, the classifier must indicate in which of the two pattern classes the input pattern, $\mathbf{x}$, belongs. This is typically accomplished by taking the output, y, of a linear functional and comparing it to a threshold, b:

If $y > b$, then $\mathbf{x}$ is in class 1.
If $y < b$, then $\mathbf{x}$ is in class 2.

Usually, the instance space is normalized, so that the threshold b is zero. This *linear-discriminant function* can be thought of as a hyperplane that splits the instance space into two regions (class 1 and class 2). For example, if $\mathbf{x} = (x_1, x_2)$ is a two-dimensional feature vector and $\mathbf{w} = (-1, 2)$, the instance space is split as shown in Figure D2-4.

The learning problem of finding $\mathbf{w}$ can thus be viewed as the problem of finding a hyperplane that separates training instances of class 1 from training instances in class 2. When it is possible to find such a hyperplane, the training instances are said to be *linearly separable*. Often, however, the training instances are not linearly separable. In such cases, we must either use a more complex functional form, such as a quadratic function, or else settle for the hyperplane that makes the fewest errors on the average.

How can the desired hyperplane, or, equivalently, the desired weight vector, be found? We describe three basic algorithms for computing the weight vector. The first two algorithms are hill-climbing methods that process the training instances one at a time. After each training instance, $\mathbf{x}_k$, the weight vector, $\mathbf{w}_k$, is updated to give $\mathbf{w}_{k+1}$.

The first algorithm, called the *fixed-increment perceptron algorithm*, seeks to minimize the classification errors made by the model. If $\mathbf{x}_k$ is an instance of class 1 and $\tilde{y} = \mathbf{w}_k \mathbf{x}_k$ is less than 0, instead of greater than 0, an error

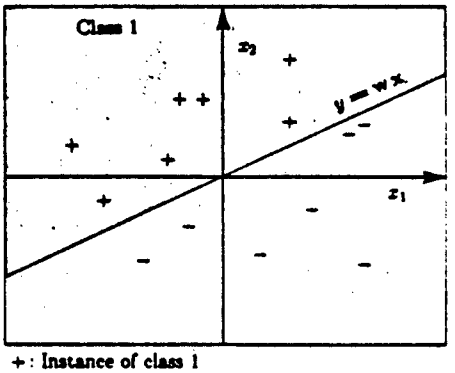


Figure D2–4. An example of a linear-discriminant function.

has been made. The magnitude of this error is $e = 0 - \mathbf{w}_k\mathbf{x}_k$, that is, the difference between the desired value for the output of the system ($y = 0$) and the value computed by the model ($\bar{y} = \mathbf{w}_k\mathbf{x}_k$). This is usually written as the perceptron criterion,

$$J_p = -\mathbf{w}_k\mathbf{x}_k ,$$

and the goal of learning is to minimize J_p. The fixed-increment algorithm updates $\mathbf{w}_k$ whenever $J_p > 0$ according to

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \mathbf{x}_k . \tag{1}$$

We can think of J_p as a surface over the *weight space*, the space of possible values for the weight vector w (see Fig. D2–5). Mathematical analysis shows that x can be viewed as a vector in this weight space (as well as in instance space) pointing in the direction of steepest descent for J_p. Thus, this algorithm takes a fixed-size step in the direction of steepest descent.

Similarly, if $\mathbf{x}_k$ is in class 2 and $\mathbf{w}_k\mathbf{x}_k > 0$, an error has been made. The solution is to adjust w as

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{x}_k .$$

Equivalently, all training instances in class 2 can be replaced by their negatives, and all instances can be processed as though they were in class 1. Equation (1) can then be used to perform the entire learning process.

The fixed-increment algorithm converges in a finite number of steps if the training instances are linearly separable. It has been shown for the two-class case that the number of training instances should be at least twice the number of features in the instance space (see Nilsson, 1965).

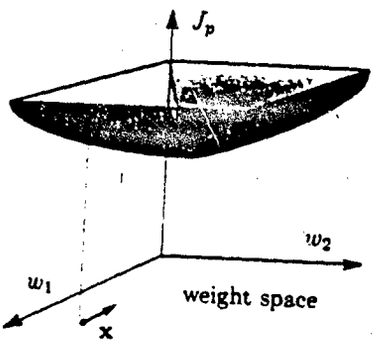


Figure D2-5. A schematic diagram of the perceptron algorithm.

Historically, the fixed-increment algorithm is associated with Rosenblatt's (1957, 1962) perceptron, which was developed within the study of bionics and neural mechanisms. The simplest perceptron, shown in Figure D2-6, is a device that assigns patterns to one of two classes. It consists of an array of sensory units connected in a random way to an array of unmodifiable threshold units, each of which computes some desired feature of the sensory array and produces a +1 or −1 output, depending on whether the feature is present or absent. The outputs of these feature-extraction units are then connected to a modifiable unit that weights each input and sums the result (i.e., computes wx). The resulting value is compared with a threshold, and the perceptron produces an output of +1 if wx is greater than the threshold and −1 otherwise. Thus, the simplest perceptron implements a linear-discriminant function. The original publication of the perceptron model sparked a large

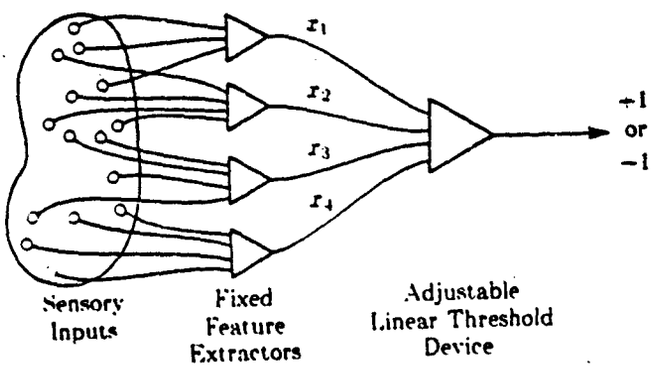


Figure D2-6. The simplest form of perceptron.

amount of research, and a fair amount of speculation, concerning the potential for building intelligent machines from perceptrons. Minsky and Papert (1969) attempted to quiet this speculation by proving several theorems about the limits of perceptron-based learning. The introduction to their book provides several criticisms of AI learning research that remain valid today.

The fixed-increment perceptron algorithm can be improved in several ways by choosing how far in the direction of the gradient to go at each step. The LMS (least-mean-square) algorithm (Widrow and Hoff, 1960), for example, updates $\mathbf{w}$ according to

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \rho e_k \mathbf{x}_k ,$$

where ρ is a positive value and e_k is the magnitude of the error, that is, $-\mathbf{w}_k\mathbf{x}_k$. This algorithm tends to minimize the mean-squared error

$$J_s = \sum_k (\mathbf{w}_k \mathbf{x}_k)^2$$

even when the classes are not linearly separable. The algorithm is also very easy to implement.

More robust, but harder to compute, algorithms are based on traditional linear-regression and linear-programming techniques (see Duda and Hart, 1973). Given a set of training instances, linear regression can be used to minimize J_s. The weight vector is computed from the data as

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} ,$$

where $\mathbf{y}$ is the true output of the unknown system and $\mathbf{X}$ is a matrix of training instances, one instance in each row. Unfortunately, this method requires computing the pseudo-inverse $(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$ of $\mathbf{X}$, which is an expensive step. Less costly recursive algorithms have been developed that can compute $\mathbf{w}$ incrementally as the training instances become available, rather than collecting all of the instances and computing $\mathbf{w}$ once and for all (Goodwin and Payne, 1977).

Linear-programming techniques can be used to minimize the perceptron criterion, J_p. These methods also conduct a hill-climbing search of the weight space. Further details are available in Duda and Hart (1973).

Some of these linear-discriminant algorithms can be modified slightly to put them on sound statistical foundations. The regression techniques, for example, can be adjusted to converge in the limit to an optimum Bayes classifier. Their rate of convergence is slower than the unmodified algorithms. Consequently, the simpler, faster algorithms shown above are often chosen in favor of the statistically more rigorous methods.

All of these methods for finding discriminant functions can be generalized to handle classification problems for more than two classes. Typically,

a separate discrimination function is learned for each of m classes, and $\mathbf{x}$ is classified to that class i for which the value of the discriminant function $f_i(\mathbf{x})$ is largest. Another approach to multiple-class problems is to perform a multistage classification in which $\mathbf{x}$ is first classified into one of a few classes and then each of these is in turn split into subclasses until $\mathbf{x}$ is properly classified. By decomposing the classification problem into subproblems, other a priori knowledge about different classes—and the features relevant to those classes—can be incorporated into the system. Most large, multicategory problems do not lend themselves to straightforward general solutions. Instead, the structure and organization of the classification strategy are usually highly dependent on the particular problem and domain-specific knowledge. Consequently, many of these classification problems overlap problems in AI.

Learning Automata

An alternate representation for an unknown system is as a finite-state automaton (Fu, 1970b). The goal is to find a finite-state automaton whose behavior imitates that of the unknown system. Two quite similar approaches have been pursued. One models the unknown system as a deterministic finite-state machine with randomly perturbed inputs. The learning program is given an initial state transition probability matrix, M, which tells overall for each state, q_i, what the probability is that the next state will be q_j. From M, an equivalent deterministic machine can be derived, and the probability distribution of the input symbols can be determined. This approach requires that the internal states of the unknown system can be precisely observed and measured.

A second approach models the unknown system as a stochastic machine with a random transition matrix for each possible input symbol. Reinforcement techniques are applied to adjust the transition probabilities. Unfortunately, this requires a large amount of training information in order to exercise all possible transitions. As with the first approach, assumptions about the observability of all internal states must be made.

Fuzzy automata based on Zadeh's *fuzzy set* concept provide an alternate, but similar, approach to that used with stochastic automata (Wee and Fu, 1969). Set-membership criteria are applied, rather than probabilistic constraints, in the selection of transitions and outputs. Fuzzy automata are also able to make higher order transitions than stochastic automata and, consequently, they can usually learn faster.

The basic ideas of automata learning have been extended to take into account the interactions of a number of automata operating in the same environment. Such automata may interact in either cooperative or competitive modes. This has led to the formulation and study of automata games (Fu, 1970b).

Automata methods have the advantage over parameter-learning methods in that they do not require that there be a performance criterion with a unique minimum point. Furthermore, automata provide a more expressive representation for describing the unknown system. The principal disadvantage of automata learning methods is that they are relatively slow compared to parameter learning techniques. In addition, they are usually suitable only for application in stationary (i.e., non-time-varying) environments. Consequently, automata methods have not yet seen much practical application.

Structural Learning

Structural learning techniques have been used primarily in situations in which the objects to be classified have important substructure (Fu, 1974). The parametric linear-discriminant approaches described above can represent only the global features of objects. By employing pattern graphs and grammars, important substructures, such as the pen strokes that make up a character and the phonemes that make up a spoken word, can be represented along with their interrelationships. A first step in setting up a structural learning scheme involves identifying a set of primitive structural elements associated with the problem. These primitives may be thought of as the alphabet for describing all possible patterns associated with the application. They need to be higher level objects than simple scalar measurements (e.g., characters, shapes, and phonemes instead of height, width, and curvature). Legal and recognizable patterns are formed from combinations of the primitives according to certain syntactic rules.

Formal language theory provides a theoretical framework that accommodates the structural or descriptive formulation of pattern recognition. Here, the alphabet corresponds to the set of structural primitives. A number of formalisms have been used to express structural descriptions. In linguistic terms, a pattern may be thought of as a string or sentence, and a grammar may be associated with each pattern class. The grammar controls the structure of the language in such a way that the sentences (patterns) produced belong exclusively to a particular pattern class; a grammar is therefore needed for each pattern class. Parsing techniques can help determine whether a sentence (pattern) is grammatically correct for a given language. Both deterministic and stochastic grammars have been employed in pattern classification. (See Article XIII.E3 for a discussion of grammatical approaches to image understanding.)

Stochastic grammars (see Article XIV.D5e) have been used in an attempt to accommodate the possibilities of ambiguity and error in pattern description. These grammars make it possible for probabilistic assignments to be made. Before such a grammar can be used for classification, the production probabilities must be determined, for example, by "learning" them from a set of training examples.

There are still several difficulties associated with the structural approach to pattern classification. In contrast to the statistical and parameter learning methods, very few practical structural training algorithms have presently been proposed. The problem of learning a grammar from training instances is called *grammatical inference*. Article XIV.D5e describes the current state of work in that area. In addition to the problem of learning the grammar, the steps of segmentation into primitives and formation of structural descriptions are only partly solved.

Relevance for Artificial Intelligence

This survey of learning systems in engineering shows that many of the problems addressed are analogous to those encountered in the design of AI learning systems. Engineering systems are particularly adept at handling noisy training instances—a problem that few AI systems have addressed. It has also been possible to develop detailed analyses of these learning algorithms, including convergence proofs and investigations of their statistical foundations.

The primary drawback of these methods is their reliance on simple feature-vector representations. Although there are many practical applications for which these representations suffice, most problems of interest to AI researchers require more expressive representations. The more recent attempts to use automata and pattern-grammar representations are much more relevant to AI research.

Some aspects of the work in engineering may be important for AI researchers. In addition to work on the problem of noise, some progress has been made on solving the problem of choosing a good set of features with which to perform the learning process. One approach is to estimate the discriminatory ability of each feature given choices of the other features. Dynamic-programming techniques can help determine a good ordering of the features (from most relevant to least relevant). A second interesting approach—called *dimensionality reduction*—is to take a large set of features and compute a new, smaller set by forming linear combinations of the old features. The Karhunen-Loève expansion can be used to create such derived features (see Fu, 1970a, and Article XIII.C5).

References

A very readable introduction to linear-discriminant functions can be found in Nilsson (1965). Duda and Hart (1973) provide an excellent survey of pattern recognition techniques. Tsypkin (1973) develops a formal, unified treatment of learning methods in engineering.

D3. Learning Single Concepts

MANY PROGRAMS have been developed that are able to learn a single concept from training instances. This article describes the single-concept learning problem and discusses a few, selected learning programs that give a sense of the techniques that have been applied to this problem.

What does it mean to learn a concept from training instances? The term *concept* is used quite loosely in the AI literature. In this article, we take a concept to be a predicate, expressed in some description language, that is TRUE when applied to a positive instance and FALSE when applied to a negative instance of the concept. A concept is thus a predicate that partitions the instance space into positive and negative subsets. For example, the concept of *straight* can be thought of as a predicate that indicates, for any poker hand, whether or not that hand is a straight.

The single-concept learning problem is the problem of discovering such a concept predicate from training instances—that is, from a sample of positive and negative instances in the instance space. The standard solution to this problem is to provide the learning program with a space of possible concept descriptions that the learning program searches to find the desired concept description (see Article XIV.D1).

Formally, the single-concept learning problem can be stated as follows:

Given: (1) A representation language for concepts. This implicitly defines the rule space: the space of all concepts representable in the language.

(2) A set of positive (and usually negative) training instances. In most work to date, these training instances are noise free and classified in advance by the teacher.

Find: The unique concept in the rule space that best covers all of the positive and none of the negative instances. Most work to date assumes that if enough instances are presented, exactly one concept exists that is consistent with the training instances.

To gain insight into the origin of the single-concept learning problem, it is useful to examine the performance tasks that make use of the concept once it is learned. The standard performance task is *classification;* the system is presented with new unknowns and is asked to classify them as positive or negative instances of a concept. Another common task is *prediction;* if the training instances are successive elements of a sequence, the system is asked to predict future elements in the sequence. A third task is *data compression;* the system is given all possible instances (the full instance space) and is asked to

find a concept that compactly describes them. The concept-classification and sequence-prediction tasks both arose as laboratory paradigms within cognitive psychology (see Hunt, Marin, and Stone, 1966). Sequence extrapolation is also a paradigm example of induction as discussed by philosophers (Carnap, 1950). Data compression is of practical value for storage and classification.

The two key assumptions made in all of this work are (a) that the training instances are all examples (or counterexamples) of a single concept and (b) that that concept can be represented by a point in the given rule space. When the first assumption is violated, it is necessary to find a *set* of concepts that account for the training instances. The systems described in the article on multiple concepts (Article XIV.D4) address this problem. When the second assumption is violated, it is necessary to alter the rule space so that it *does* contain the desired concept. Very little attention has been given to this problem in single-concept learning. The BACON program employs some simple methods to alter the rule space by adding new terms to the representation language (see Article XIV.D3b).

Approaches to Solving the Single-concept Learning Problem

In Article XIV.D1, we described four basic techniques—version spaces, refinement operators, generate and test, and schema instantiation—that are used to search the rule space. Each of these search methods has been applied to the single-concept learning problem. The remainder of this article is divided into four subarticles—one devoted to each method. The first two subarticles describe data-driven methods. Mitchell's version-space method is discussed first. It provides a useful framework for describing several related systems developed by Hayes-Roth, Vere, and Winston. Then two refinement-operator systems, BACON and CLS/ID3, are presented. The second pair of subarticles describes model-driven methods: a generate-and-test method developed by Dietterich and Michalski (1981) and a schema-instantiation method, SPARC, that plays the card game Eleusis.

References

See Mitchell (1978, 1979).

D3a. Version Space

RECENT WORK by Mitchell (1977, 1979) provides a unified framework for describing systems that use a data-driven, single-representation approach to concept learning. Mitchell has noted that, in all representation languages, the sentences can be placed in a partial order according to the *generality* of each sentence. Figure D3a-1 illustrates this general-to-specific ordering with a few sentences in predicate calculus containing the predicates RED and BLACK. The concept $\exists\ c_1 : \text{RED}(c_1)$, for example, describes the set S of all poker hands that contain at least one red card. This concept is more general than the concept $\exists\ c_1\, c_2 : \text{RED}(c_1) \wedge \text{RED}(c_2)$ that describes the set T of all poker hands containing at least two red cards, since the set S strictly contains the set T. The set of cards described by $\exists\ c_1\, c_2\, c_3 : \text{RED}(c_1) \wedge \text{RED}(c_2) \wedge \text{BLACK}(c_3)$ is smaller still and, thus, is even more specific than the $\exists\ c_1\, c_2 : \text{RED}(c_1) \wedge \text{RED}(c_2)$ concept.

It should be evident that the syntactic rules of generalization described in Article XIV.D1 can be used to generate this partial ordering. In this example, the *dropping-conditions* rule of generalization was applied to the three most specific concepts to generate the others. In general, any rule space can be partially ordered according to the general-to-specific ordering.

The most general point in the rule space is usually the null description (in which *all* conditions have been dropped), which places no constraints on the training instances and thus describes anything. The most specific points in the rule space correspond to the training instances themselves—represented in the same representation language as that used for the rule space (see Fig. D3a-2).

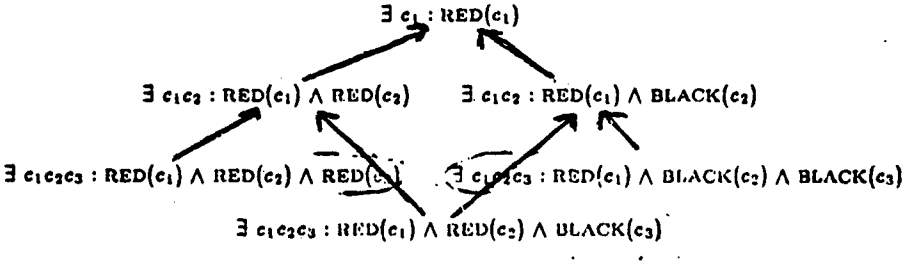
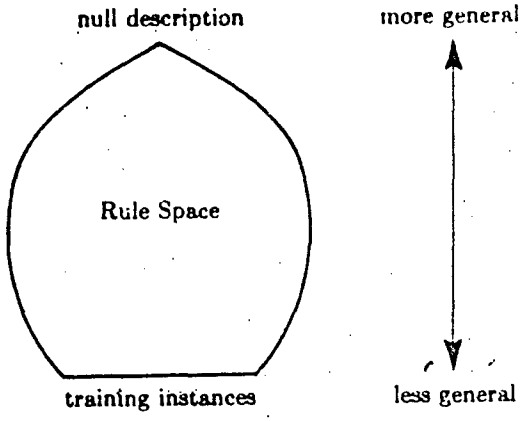


Figure D3a-1. A small rule space and its general-to-specific ordering.



Mitchell has pointed out that programs can take advantage of this partial ordering to represent the set H of plausible hypotheses very compactly. A set of points in a partially ordered set can be represented by its *most general* and *most specific* elements. Thus, as shown in Figure D3a-3, the set H of plausible hypotheses can be represented by two subsets: the set of most general elements in H (called the G set) and the set of most specific elements in H (called the S set). Once H has been represented in this manner, the rules of generalization must be used to fill in the subspace between the G set and the S set whenever the full H set is needed.

The Candidate-elimination Learning Algorithm

Mitchell's learning algorithm, called the *candidate-elimination algorithm*, takes advantage of the boundary-set representation for the set H of plausible

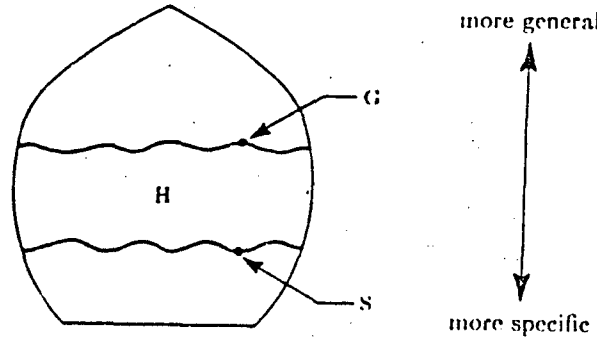


Figure D3a-3. Using the boundary sets to represent a subspace of the rule space.

hypotheses. Mitchell defines a plausible hypothesis as *any* hypothesis that has not yet been ruled out by the data. The set H of all plausible hypotheses is called the *version space*. Thus, the version space, H, is the set of all concept descriptions that are consistent with all of the training instances seen so far.

Initially, the version space is the complete rule space of possible concepts. Then, as training instances are presented to the program, candidate concepts are eliminated from the version space. When it contains only one candidate concept, the desired concept has been found. The candidate-elimination algorithm is a *least-commitment* algorithm, since it does not modify the set H until it is *forced to do so* by the training information. Positive instances force the program to generalize—thus, very specific concept descriptions are removed from the H set. Conversely, negative instances force the program to specialize, so very general concept descriptions are removed from the H set. The version space gradually shrinks in this manner until only the desired concept description remains.

To see how training instances force the version space to shrink, consider once again the problem of teaching a program the *flush* concept in poker. Suppose the program has already seen the positive training instance

$$\{(2, clubs), (5, clubs), (7, clubs), (jack, clubs), (queen, clubs)\} \Rightarrow \text{FLUSH}.$$

Since the candidate-elimination algorithm is a least-commitment algorithm, it makes the most specific possible assumption about the *flush* concept. Namely, it sets up the S set to contain

$$\begin{aligned} S = \{&\text{SUIT}(c_1, clubs) \wedge \text{RANK}(c_1, 2) \wedge \\ &\text{SUIT}(c_2, clubs) \wedge \text{RANK}(c_2, 5) \wedge \\ &\text{SUIT}(c_3, clubs) \wedge \text{RANK}(c_3, 7) \wedge \\ &\text{SUIT}(c_4, clubs) \wedge \text{RANK}(c_4, jack) \wedge \\ &\text{SUIT}(c_5, clubs) \wedge \text{RANK}(c_5, queen)\}. \end{aligned}$$

This hypothesis is very specific indeed. It says that there is only one hand that could possibly be a flush. At the same time, however, the candidate-elimination algorithm makes the most general possible assumption, namely, that *every* possible hand is a *flush*. The G set contains the null description. This means that the version space—the H set—of all plausible hypotheses contains S, G, and every hypothesis in between.

Now, suppose the positive training instance

$$\{(3, clubs), (8, clubs), (10, clubs), (king, clubs), (ace, clubs)\} \Rightarrow \text{FLUSH}$$

is presented. The candidate-elimination algorithm realizes that its initial assumption for the S set was too specific—there *are* other hands that can be

flushes. Thus, it is forced to generalize S to contain, among other hypotheses, the rule

$$S = \{\text{SUIT}(c_1, clubs) \wedge \text{SUIT}(c_2, clubs) \wedge \text{SUIT}(c_3, clubs) \wedge \text{SUIT}(c_4, clubs) \wedge \text{SUIT}(c_5, clubs)\}.$$

The G set does not change. Suppose, however, that a negative training instance

$$\{(3, spades), (8, clubs), (10, clubs), (king, clubs), (ace, clubs)\} \Rightarrow \neg\text{FLUSH}$$

is presented. This forces the candidate-elimination algorithm to realize that its assumption for the G set, that *any* hand could be a flush, was wrong. It must specialize the G set in some way, so that it does not wrongly classify this hand as a *flush*.

In full detail, the candidate-elimination algorithm proceeds as follows:

Step 1. Initialize H to be the whole space. Thus, the G set contains only the null description, and the S set contains all of the most specific concepts in the space. (In practice, this is not actually done due to the huge size of S. Instead, the S set is initialized to contain only the first positive example. Conceptually, however, H starts out as the whole space.)

Step 2. Accept a new training instance. If the instance is a positive example, first remove from G all concepts that do not cover the new example. Then update S to contain all of the maximally specific common generalizations of the new instance and the previous elements in S. In other words, generalize the elements in S *as little as possible*, so that they will cover this new positive example. This is called the Update-S routine.

If the instance is a negative example, first remove from S all concepts that cover this counterexample. Then update the G set to contain all of the maximally general, common specializations of the new instance and the previous elements in G. In other words, specialize the elements in G *as little as possible* so that they will *not* cover this new negative example. This is called the Update-G routine.

Step 3. Repeat step 2 until $G = S$ and this is a singleton set. When this occurs, H has collapsed to include only a single concept.

Step 4. Output H (i.e., either G or S).

Here is an example of a complete run of the candidate-elimination algorithm. Suppose we have the following feature-vector representation language: The instance space is a set of objects, each object having two features—*size* and *shape*. The size of an object can be *small* or *large*, and the shape of an

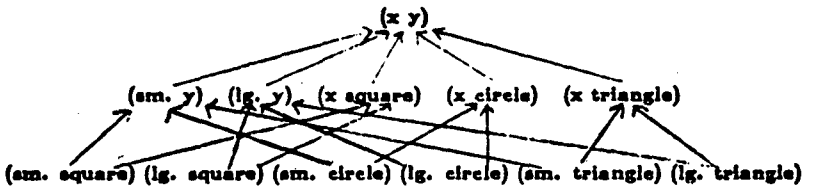


Figure D3a–4. The initial version space and the general-to-specific partial order.

object can be *circle*, *square*, or *triangle*. Figure D3a–4 shows the entire rule space for this representation language.

Each point in the rule space specifies either a variable or a value for both of the features. If a feature is specified by a variable, then *any* value of that feature can be applied.

Suppose we want to teach the program the concept of a *circle*. This is represented as `(x circle)` where x represents any size. First we initialize the H set to be the entire rule space. This means that the G set is

$$G = \{\texttt{(x y)}\},$$

representing the most general possible concept, and the S set is

$$S = \{\texttt{(small square) (large square) (small circle) (large circle) (small triangle) (large triangle)}\}.$$

Now we present the first training instance: a positive example of the concept, a small circle. The Update-S algorithm is applied in step 2 to yield:

$$G = \{\texttt{(x y)}\}$$
$$S = \{\texttt{(small circle)}\}.$$

Figure D3a–5 shows the resulting version space. Solid lines connect concepts that are still *in* the version space. In practical implementations of the candidate-elimination algorithm, the version space is usually initialized at this point rather than explicitly listing the entire instance space as in the step above.

The second training instance is `(large triangle)`—a negative example of the concept. This forces the G set to be specialized. Update-G is applied to produce

$$G = \{\texttt{(x circle) (small y)}\}$$
$$S = \{\texttt{(small circle)}\}.$$

Figure D3a–6 shows the resulting version space.

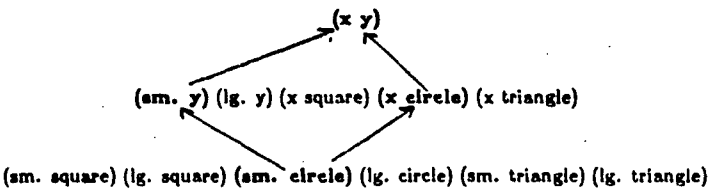


Figure D3a–5. The version space after the first training instance.

Notice how the (x y) description was specialized in two distinct ways, so that it no longer covered the negative example (large triangle). A third possible specialization (x square) is not considered, since it was removed from the version space during the previous training instance. Of course, further specializations such as (small circle) are not considered because the Update-G algorithm specializes as little as possible.

In this case, the G set grew larger as a result of the specialization. The Update-G and Update-S algorithms often expand the size of the G and S sets. It is the size of these sets that limits the practical application of this algorithm.

Finally, we present the algorithm with another positive example: (large circle). Update-S first prunes G to eliminate (small y), since it does not cover (large circle). Then S is generalized as necessary:

$$G = \{(\texttt{x circle})\}$$
$$S = \{(\texttt{x circle})\}.$$

Since $G = S$, the algorithm halts and prints (x circle) as the concept.

It is possible to give intuitive interpretations of the G and S sets. The set S is the set of sufficient conditions for a new example to be an instance

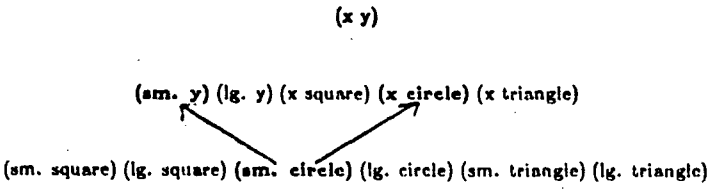


Figure D3a–6. The version space after two training instances.

of the concept. Thus, after the second training instance, we know that if the new example is a `(small circle)`, it is an instance of the concept; `(small circle)` is a sufficient condition for positive classification. The set G is the set of necessary conditions. After the second training instance, we know that an object either must be a circle or must be small in order to be an instance of the concept. Neither of these conditions is sufficient. The algorithm terminates when the necessary conditions are equal to the sufficient conditions—that is, the algorithm has found a necessary and sufficient condition.

It is important to note that the candidate-elimination algorithm conducts an exhaustive, breadth-first search of the given rule space, guided only by the training instances. This makes the algorithm infeasibly slow for large rule spaces. The efficiency of the algorithm can be improved (at the cost of possibly failing to find the desired concept) by employing heuristics to prune the S and G sets. We postpone further discussion of the strengths and weaknesses of the candidate-elimination algorithm until after we have discussed the related methods developed by Hayes-Roth, Vere, and Winston.

Methods Related to the Version-space Approach

Two learning methods similar to the Update-S procedure of the version-space algorithm were developed prior to it. One method, termed *interference matching*, was developed by Hayes-Roth and McDermott (1977, 1978). The other method, the *maximal unifying generalization* method, was developed by Vere (1975, 1978). These methods can both be viewed as implementations of the Update-S procedure with respect to slightly different representation languages in that they learn from positive training instances only.

Interference matching was developed to discover concepts expressed in Hayes-Roth's Parameterized Structural Representation (PSR), which is roughly equivalent to an existentially quantified conjunctive statement in predicate calculus. Recall that Update-S seeks to generalize the descriptions in S as little as possible in order to cover each new positive training instance. When the descriptions are represented as predicate calculus expressions, this is equivalent to finding the largest common subexpressions, because the largest common subexpression is that subexpression for which the *fewest* conjunctive conditions need to be dropped. As an example, suppose that the set S contains the description

$$S = \{\text{BLOCK}(x) \wedge \text{BLOCK}(y) \wedge \text{RECTANGLE}(x) \wedge \text{ONTOP}(x, y) \wedge \text{SQUARE}(y)\}$$

and the next positive training instance (I_1) is

$$I_1 = \text{BLOCK}(w) \wedge \text{BLOCK}(v) \wedge \text{SQUARE}(w) \wedge \text{ONTOP}(w, v) \wedge \text{RECTANGLE}(v).$$

Update-S will produce the following common subexpressions:

$$S' = \{s_1, s_2\},$$

where s_1 = BLOCK(a) ∧ BLOCK(b) ∧ SQUARE(a) ∧ RECTANGLE(b), and s_2 = BLOCK(c) ∧ BLOCK(d) ∧ ONTOP(c, d).

The s_1 description corresponds to the hypothesis that the ONTOP relation is irrelevant to the concept. The s_2 description, on the other hand, corresponds to the hypothesis that the shapes of the objects involved are irrelevant. Notice that there is no consistent way to match I_1 to S that preserves a one-to-one correspondence of the variables x and y with u and v; either the rectangle and square predicates conflict (e.g., when x is matched with w) or else the order of the arguments to ONTOP conflict (e.g., when x is matched to v).

The interference-matching algorithm starts out as a breadth-first search of all possible matchings of one PSR with another. The search proceeds by "growing" common subexpressions until a space limit is reached. Unpromising matches are then pruned with a heuristic utility function, and the growing process continues in a more depth-first fashion. The utility of a partial match is equal to the number of predicates matched less the number of variables matched. If the space limit is approximately the same as the largest common subexpression, the algorithm becomes truly depth-first, since only one subexpression "fits" within the space limit. Thus, the interference-matching algorithm tends to find one good common subexpression rather than finding all maximal common subexpressions (as in the Update-S algorithm).

Vere's algorithm for finding the maximal unifying generalization of two first-order predicate-calculus descriptions is very similar to the interference-matching algorithm. The representation language used by Vere, however, permits a many-to-one binding of parameters during the matching process (Vere, 1975). Vere's method also conducts a breadth-first search of possible matchings but does not do any pruning of this search.

Winston's Work on Learning Structural Descriptions from Examples

Winston's (1970) influential work on structural learning served as a precursor to the other learning methods described above. The method has the same basic data-driven approach as in the version-space and related algorithms: Training instances are accepted one at a time and matched against the concept descriptions in the set H. Unlike those breadth-first algorithms (e.g., Update-S and Update-G), however, Winston's system conducts a *depth-first* search of the concept space. Instead of maintaining a *set* of plausible hypotheses, Winston's program uses the training instances to update a *single* current concept description. This description contains all of the program's knowledge about the concept being learned.

The task of the program is to learn concept descriptions that characterize simple toy-block constructions. The toy-block assemblies are initially presented to the computer as line drawings. A knowledge-based interpretation program converts these line drawings into a semantic-network description.

Winston also uses this semantic-network representation to describe the current concept and some background knowledge about toy blocks.

Figure D3a–7 shows a line drawing of an arch and the corresponding semantic network. The network is roughly equivalent to the predicate-calculus expression

$$\begin{aligned}&\text{ONE-PART-IS}(\textit{arch}, a) \wedge \text{ONE-PART-IS}(\textit{arch}, b) \wedge \\ &\text{ONE-PART-IS}(\textit{arch}, c) \wedge \text{HAS-PROPERTY-OF}(a, \textit{lying}) \wedge \\ &\text{A-KIND-OF}(a, \textit{object}) \wedge \text{MUST-BE-SUPPORTED-BY}(a, b) \wedge \\ &\text{MUST-BE-SUPPORTED-BY}(a, c) \wedge \text{MUST-NOT-ABUT}(b, c) \wedge \\ &\text{MUST-NOT-ABUT}(c, b) \wedge \text{LEFT-OF}(b, c) \wedge \text{RIGHT-OF}(c, b) \wedge \\ &\text{HAS-PROPERTY-OF}(b, \textit{standing}) \wedge \text{HAS-PROPERTY-OF}(c, \textit{standing}) \wedge \\ &\text{A-KIND-OF}(b, \textit{brick}) \wedge \text{A-KIND-OF}(c, \textit{brick}),\end{aligned}$$

along with statements of blocks-world knowledge such as

$$\begin{aligned}&\text{A-KIND-OF}(\textit{brick}, \textit{object}) \\ &\text{A-KIND-OF}(\textit{standing}, \textit{property})\end{aligned}$$

and statements relating different predicates in the representation language, such as

$$\begin{aligned}&\text{OPPOSITES(MUST-ABUT, MUST-NOT-ABUT)} \\ &\text{MUST-FORM-OF(IS-SUPPORTED-BY, MUST-BE-SUPPORTED-BY)}.\end{aligned}$$

A distinctive aspect of Winston's concept representation is that it allows *necessary conditions* to be represented explicitly. For example, the condition that in an arch the posts *must not touch* can be directly represented by a MUST-NOT-ABUT link. This allows Winston's program to express necessary and sufficient conditions in one combined network structure.

Winston's learning algorithm works as follows:

Step 1. Initialize the current concept description, H, to be the network corresponding to the first positive training instance.

Step 2. Accept a new line drawing and convert it into a semantic-network representation.

Step 3. Match the training instance with H (using a graph-matching algorithm) to obtain the common skeleton. The skeleton is a maximal common subgraph of the two graphs. Annotate the skeleton by attaching comments indicating those nodes and links that did *not* match.

Step 4. Use the annotated skeleton to decide how to modify the current concept description H.

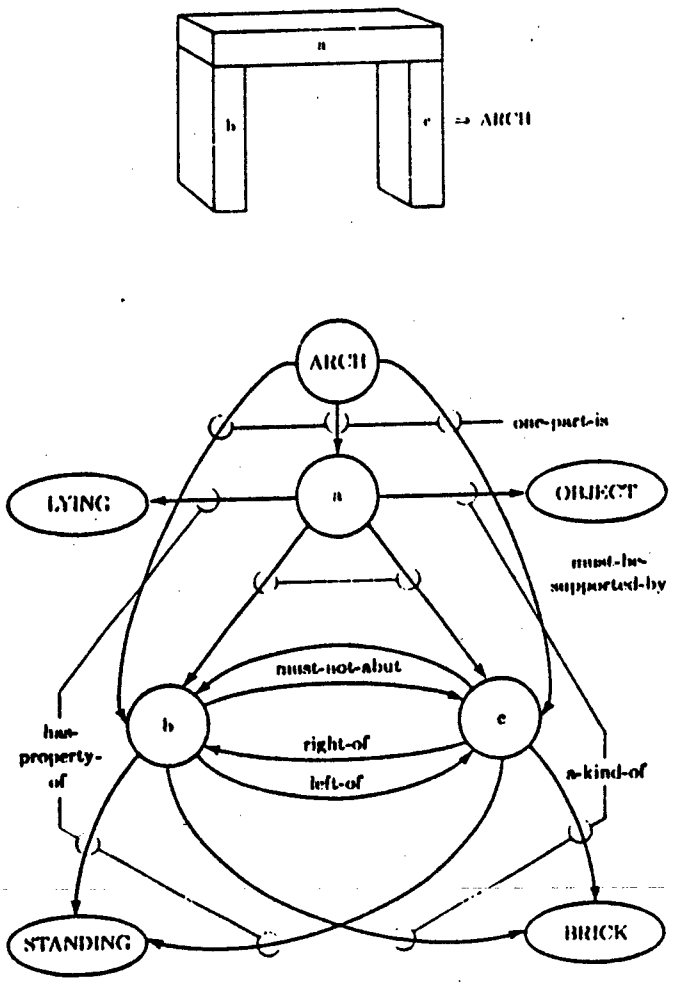


Figure D3a–7. A training instance and its internal representation.

If the new instance is a positive example of the concept, then generalize H as necessary. The algorithm generalizes either by dropping nodes and links or by replacing one node (e.g., `cube`) by a more general node (e.g., `brick`). In some cases, the algorithm must choose between these two generalization techniques. The program chooses the less drastic method (node replacement) and places the other choice on a backtrack list.

If the new instance is a negative example of the concept, a necessary condition (represented by a `must`-link) is added to H. If there are several differences between the negative training instance and H, the algorithm applies some ad hoc rules to choose one difference to "blame" for causing the instance to be a negative instance. This difference is converted into a necessary condition. The other differences are ignored.

Repeat steps 2, 3, and 4 until the teacher halts the program.

Since the algorithm searches in depth-first fashion, it is possible for contradictions to arise in step 4. For example, after seeing a negative training instance such as shown in Figure D3a–8, the algorithm might assume in step 4 that the reason this is not an arch is the triangular lintel rather than the fact that the posts are touching. Subsequently, when the program sees the positive instance shown in Figure D3a–9, a contradiction arises. When this happens, the system backtracks to the last point at which a choice was made, and the algorithm makes a new choice.

This learning algorithm is somewhat weak and ad hoc, since it does not concern itself either with the possibility that the training instance matches H in multiple ways or with the problem that there are multiple ways of generalizing or specializing H. Winston makes two important assumptions that allow this algorithm to ignore these problems. First, it is assumed that the training instances are presented in good pedagogical order, so that contradictions and choice-points are unlikely to arise; the teacher is assumed to have chosen the examples so as to vary only one aspect of the concept in each example. The second assumption is that the negative training instances

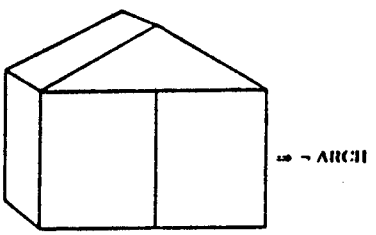


Figure D3a–8. A near-miss negative example of an ARCH.

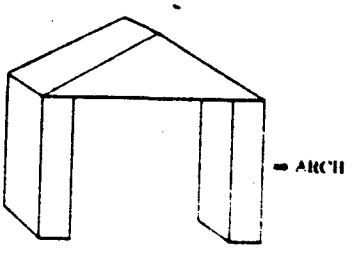


Figure D3a-9. A positive example of an ARCH.

are all *near misses*, that is, instances that just barely fail to be examples of the concept in question. These two assumptions permit the learning system to perform fairly well in the domain of toy-block concepts.

Weaknesses of the Version-space Approach (and Related Approaches)

There are several weaknesses in these methods that limit their practical application. This section discusses these problems and examines some proposed solutions.

Noisy training instances. As with all data-driven algorithms, these methods have difficulty with noisy training instances. Since these algorithms seek to find a concept description that is consistent with *all* of the training instances, any single bad instance (i.e., a false positive or false negative instance) can have a big effect. When the candidate elimination algorithm is given a false positive instance, for example, the S set becomes overly generalized. Similarly, a false negative instance causes the G set to become overly specialized. Eventually, noisy training instances can lead to a situation in which there are no concept descriptions that are consistent with all of the training instances. In such cases, the G set "passes" the S set, and the version space of consistent concept descriptions becomes empty. The methods of Hayes-Roth, Vere, and Winston also overgeneralize in the presence of false positive training instances.

In order to learn in the presence of noise, it is necessary to relax the condition that the concept descriptions be consistent with *all* of the training instances. One solution, proposed by Mitchell (1978), is to maintain several S and G sets of varying consistency. The set S_0, for example, is consistent with *all* of the positive examples, and the set S_1 is consistent with *all but one* of the positive examples. In general, each description in the set S_i is consistent with all but i of the positive training instances. Similarly, each description in the set G_i is consistent with all but i of the negative training instances. Figure D3a-10 gives a schematic diagram of these sets. Mitchell provides a fairly efficient algorithm for updating these multiple boundary sets.

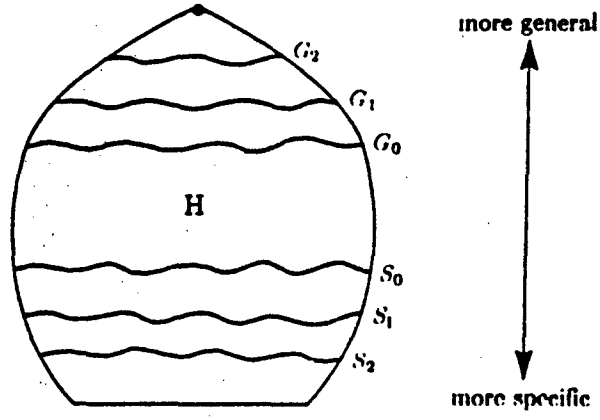


Figure D3a–10. The multiple-boundary set technique.

When G_0 crosses S_0, the algorithm can conclude that no concept in the rule space is consistent with *all* of the training instances. The algorithm can recover and try to find a concept that is consistent with *all but one* of the training instances. If that fails, it can look for a concept consistent with all but two instances, and so forth. This approach to error recovery works for learning problems containing a few erroneous training instances, but it requires a large amount of memory to store all of the S and G boundary sets.

Disjunctive concepts. A second, important weakness of these data-driven algorithms is their inability to discover disjunctive concepts. Many concepts have a disjunctive form. For instance, an *uncle* is either the brother of a parent or the spouse of a sister of a parent:

$$\text{UNCLE}(x) = \text{BROTHER}(\text{PARENT}(x)) \vee$$
$$\text{UNCLE}(x) = \text{SPOUSE}(\text{SISTER}(\text{PARENT}(x))).$$

Parent itself might be expressed disjunctively as $\text{PARENT}(x) = \text{FATHER}(x) \vee \text{PARENT}(x) = \text{MOTHER}(x)$. However, if disjunctions of arbitrary length are permitted in the representation language, the data-driven algorithms described above never generalize. In the candidate-elimination algorithm, for example, the S set will always contain a single disjunction of all of the positive training instances seen so far. This is because the least generalization of a new training instance and the current S set is simply the disjunction of the new instance with the S set. Similarly, the G set will contain the disjunction of the negation of each of the negative training instances. Unlimited disjunction allows the partially ordered rule space to become infinitely "branchy."

The basic difficulty is that all of these algorithms are least-commitment algorithms that generalize only when they are forced to. Disjunction provides a way of avoiding any generalization at all—so the algorithms are never forced to generalize. In order to develop a useful technique for learning disjunctive concepts, some method must be found for controlling the introduction of disjunctions. The learning algorithms must be guided toward generalizing in certain ways to exclude the *trivial disjunction*.

One solution (proposed in different forms by Michalski, 1969, and by Mitchell, 1978) is to employ a representation language that does not contain a disjunction operator and to perform repeated candidate-elimination runs to find several conjunctive descriptions that together cover all of the training instances. We repeatedly find a conjunctive concept description that is consistent with *some* of the positive training instances and *all* of the negative training instances. The positive instances that have been accounted for are removed from further consideration, and the process is repeated until all positive instances have been covered:

Step 1. Initialize the S set to contain one positive training instance. G is initialized to the null description—the most general concept.

Step 2. For each negative training instance, apply the Update-G algorithm to G.

Step 3. Choose a description g from G as one conjunction for the solution set. Since Update-G has been applied using *all* of the negative instances, g covers *no* negative instances. However, g may cover several of the positive instances. Remove from further consideration all positive training instances that are more specific than g.

Step 4. Repeat steps 1 through 3 until all positive training instances are covered.

This process builds a disjunction of descriptions that covers all of the data. It tends to find a disjunction containing only a few conjunctive terms. Figure D3a-11 is a schematic diagram of how this process works.

The point s_1 is the first positive training instance selected in step 1. After all of the negative instances have been processed with Update-G, g_1 is selected from the G set in step 3. Notice that g_1 covers several positive instances in addition to s_1, but that not all positive instances are yet covered. The point s_2 is then chosen and g_2 is developed. Similarly, s_3 is chosen and g_3 is developed. As the figure shows, the conjunctive concepts, g_i, need not be disjoint. Also, the set of concepts g_i that is obtained by this procedure varies depending on the order in which the positive training instances are selected in step 1.

An algorithm very similar to this, called the AQ algorithm, was developed by Michalski (1969, 1975) for use with an extended propositional calculus representation. The AQ algorithm makes use of an additional heuristic in

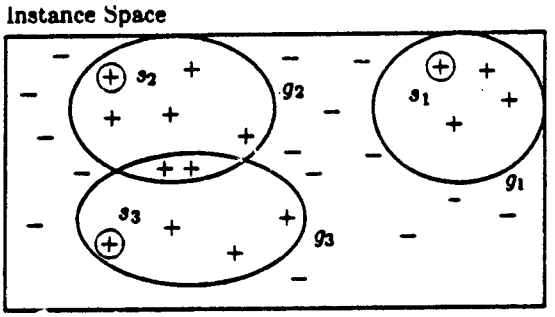


Figure D3a–11. Schematic diagram of an iterative version-space algorithm for finding disjunctive concepts.

step 1. It selects as a "seed" positive training instance one that has not been covered by *any* description in *any* previous G set. This has the effect of choosing training instances that are "far apart" in the instance space. Larson (1977) elaborated A^q to apply it to an extended predicate-calculus representation.

The effect of this iterative version-space approach is to find a description with virtually the fewest number of disjunctive terms. Finding such a description is not always desirable. Programs searching for symmetrical descriptions, for example, may hypothesize a disjunctive term for which there is, as yet, no evidence. Consider how a program would learn the direction of wind rotation about a weather system. After seeing the following two training instances

Instance 1. HEMISPHERE = *north* ∧ PRESSURE = *high*
⇒ ROTATION = *clockwise*

Instance 2. HEMISPHERE = *south* ∧ PRESSURE = *high*
⇒ ROTATION = *counterclockwise*,

the program might hypothesize that

HEMISPHERE = *north* ∧ PRESSURE = *high* ∨
HEMISPHERE = *south* ∧ PRESSURE = *low*
⇒ ROTATION = *clockwise*,

even though the *simplest* hypothesis would be

HEMISPHERE = *north* ⇒ ROTATION = *clockwise*.

The problem of learning disjunctive concepts is still largely unexamined by AI researchers.

References

Mitchell (1977, 1979) provides good descriptions of the version-space approach. Hayes-Roth and McDermott (1978), Vere (1975), and Winston (1970) present detailed descriptions of their methods. See Dietterich and Michalski (1981) for a critical comparison of these methods.

D3b. Data-driven Rule-space Operators

THE SECOND FAMILY of data-driven methods does not employ partial matching to search the rule space. Instead, these methods develop a set of hypotheses in a rule space that is separate from the instance space (i.e., the single-representation trick is not used). The hypotheses are modified by *refinement operators*, which are selected by heuristics that inspect the training instances. The following is a general outline of these operator-based algorithms:

Step 1. Gather some training instances.

Step 2. Analyze the instances to decide which rule-space operator to apply.

Step 3. Apply the operator to make some change in the current set, H, of hypotheses.

Repeat steps 1 through 3 until satisfactory hypotheses are obtained.

In this article, two systems are described that use this technique: BACON and CLS.

BACON

BACON is a set of concept-learning programs developed by Pat Langley (1977, 1980). These programs solve a variety of single-concept learning tasks, including "rediscovering" such classical scientific laws as Ohm's law, Newton's law of universal gravitation, and Kepler's law. The programs are also capable of using the learned concepts to predict future training instances.

The idea underlying BACON is simple: The program repeatedly examines the data and applies its refinement operators to create new terms. This continues until it finds that one of these terms is always constant. A single concept is thus represented in the form *term* = *constant value*.

BACON uses a feature-vector representation to describe each training instance. A distinguishing aspect is that the features may take on continuous real values as well as discrete symbolic or numeric values. For example, suppose we want BACON to discover Kepler's law: The period of a planet's revolution around the sun, p, is related to its distance from the sun, d, as $d^3/p^2 = k$, for some constant k. First, BACON is supplied with training instances of the form:

		Features	
Instance	*Planet*	p	d
I_1	Mercury	1	1
I_2	Venus	8	4
I_3	Earth	27	9

BACON is told that p and d are dependent on the value of the planet variable. Once BACON has gathered a few training instances, it examines them to see if any of its rule-space operators are triggered. In this case, since p and d are both increasing and are not linearly related, an operator that creates the new term d/p is triggered. This rule-space operator is executed, and the training instances are reformulated to give:

		Features		
Instance	*Planet*	p	d	d/p
I_1	Mercury	1	1	1.0
I_2	Venus	8	4	.5
I_3	Earth	27	9	.33

Again, BACON checks to see if any of its rule-space operators are triggered. This time, the product operator is executed to create the term $(d/p)d$, since d and d/p are varying inversely. The data are reformulated to give:

		Features			
Instance	*Planet*	p	d	d/p	d^2/p
I_1	Mercury	1	1	1.0	1.0
I_2	Venus	8	4	.5	2.0
I_3	Earth	27	9	.33	3.0

On the third iteration, BACON again checks to see if any operators apply. The product operator is again triggered to create the term $(d/p)(d^2/p)$. The data are reformulated to give:

		Features				
Instance	*Planet*	p	d	d/p	d^2/p	d^3/p^2
I_1	Mercury	1	1	1.0	1.0	1.0
I_2	Venus	8	4	.5	2.0	1.0
I_3	Earth	27	9	.33	3.0	1.0

BACON examines these data, and its constancy operator is triggered to create the hypothesis that the d^3/p^2 term is constant. BACON then gathers more data to test this hypothesis before it halts.

BACON's Rule-space Operators

The various BACON programs have different rule-space operators. Each operator is stored as a production rule, of which the left-hand side performs extensive tests to search for possible patterns in the data and the right-hand side creates the new terms. Here is a brief survey of the operators implemented in the BACON.1 program:

1. *Constancy detection.* This operator is triggered when some dependent variable takes on the same value, v, at least two times. It creates the hypothesis that this variable is always constant with value v.
2. *Specialization.* This operator is triggered when a previously created hypothesis is contradicted by the data. It specializes the hypothesis by adding a conjunctive condition.
3. *Slope and intercept term creation.* This operator detects that two variables are varying together linearly and creates new terms for the slope and intercept of this linear relation.
4. *Product creation.* This operator detects that two variables are varying inversely without a constant slope. It creates a new term that is the product of the two variables.
5. *Quotient creation.* This operator detects that two variables are varying monotonically (increasing or decreasing) without constant slope. It creates a new term that is the quotient of the two variables.
6. *Modulo-n term creation.* This operator notices that one variable, v_1, takes on a constant value whenever an independent variable, v_2, has a certain value modulo n. The new term v_2-*modulo*-n is created. Only small values of n are considered.

Extensions to BACON

BACON.2 is an extended version of BACON.1 that includes two additional operators for detecting recurring sequences and for creating polynomial terms by calculating repeated differences. BACON.2 can solve a larger class of sequence extrapolation tasks as a result.

BACON.3 is another extension of BACON.1 that uses hypotheses proposed by the constancy-detection operators to reformulate the training instances. For BACON.3 to discover the ideal gas law (PV/NT is equal to a constant), for example, it is given the following training instances:

	Features			
Instance	*V*	*P*	*T*	*N*
I_1	.0083200	300,000	300	1
I_2	.0062400	400,000	300	1
I_3	.0049920	500,000	300	1
I_4	.0085973	300,000	310	1
I_5	.0064480	400,000	310	1
I_6	.0051584	500,000	310	1
I_7	.0088747	300,000	320	1
I_8	.0066560	400,000	320	1
I_9	.0053248	500,000	320	1
⋮	⋮	⋮	⋮	⋮

	Features			
Instance	*V*	*P*	*T*	*N*
⋮	⋮	⋮	⋮	⋮
I_{25}	.0266240	300,000	320	3
I_{26}	.0199680	400,000	320	3
I_{27}	.0159740	500,000	320	3

By applying the product-creation operator followed by the constancy-detection operator, BACON develops the hypothesis that PV is constant for particular values of N and T. This hypothesis, which BACON must rediscover for each particular value of N and T, is used to recast the data to give the following derived training instances:

	Features		
Instance	*PV*	*T*	*N*
I'_1	2,496	300	1
I'_2	2,579.1999	310	1
I'_3	2,662.3999	320	1
I'_4	4,991.9999	300	2
I'_5	5,158.3999	310	2
I'_6	5,324.7999	320	2
I'_7	7,488	300	3
I'_8	7,737.5999	310	3
I'_9	7,987.2	320	3

Each of these derived instances results from collapsing three of the original training instances. Thus, I'_1 is derived by noticing that PV takes on the constant value 2,496 in I_1, I_2, and I_3. By applying the slope-intercept operator to these derived instances, BACON develops the hypothesis that PV/T is constant for particular values of N. It uses this hypothesis to recast the training instances into the following form:

	Features	
Instance	*PV/T*	*N*
I''_1	8.32	1
I''_2	16.64	2
I''_3	24.95	3

By applying the slope-intercept operator to these doubly derived instances, BACON develops the hypothesis that PV/NT is constant and, thus, posits the ideal gas law.

BACON's Rule Space

What is the rule space that BACON is searching? BACON expresses hypotheses as feature vectors, some of whose values are omitted (i.e., turned to variables). For example, Kepler's law is expressed as

Features:	*Planet*	p	d	d/p	d^2/p	d^3/p^2
Values:	-	-	-	-	-	1.0

Thus, the rule space is the space of such feature vectors whose features are any terms that BACON can create with its operators.

BACON conducts a sort of depth-first search through this space. The conditions under which the operators are triggered are quite specialized. The constancy-detection operator, for example, only checks the values of the most recently created dependent variable against the most recently varied independent variable. Most of the other operators are invoked under similarly constrained conditions.

Strengths and Weaknesses of BACON

BACON's primary strength is its ability to discover simple laws relating real-valued variables. Also of interest is BACON's use of rule-space operators to create new terms as combinations of existing terms. Further, the BACON.3 strategy of reformulating the training instances when partial regularities are discovered may be important for future learning programs. Simon (1979) has discussed BACON as a model of data-driven theory formation in science.

There are some difficulties with the present BACON programs, however. First, the fact that the operators are evoked only under highly specialized conditions causes the program to be sensitive to the order of the variables and to the particular values chosen for the training instances. For some sets of training instances, for example, BACON is unable to discover Ohm's law (see Langley, 1980, p. 104). It is necessary to adjust the order of the variables and the particular training instances to get BACON to discover concepts efficiently. For example, when BACON is discovering the pendulum law, 40% more time is required if the variables are poorly ordered. Similarly, it cannot handle irrelevant variables well.

Second, BACON is unable to handle noisy training instances. The triggering of the constancy detectors, for example, is based on the near equality of the values seen in as few as *two* training instances. Such calculations are highly sensitive to noise. The slope detectors are similarly sensitive.

Third, BACON can handle only relatively simple concept-formation tasks involving nonnumeric variables. The program cannot, for example, discover concepts that involve internal disjunction (such as the concept of a *red or green cube*). It is also unable to discover the simple concept underlying the

letter sequence ABTCDSEFR ... and similar sequences appearing in Kotovsky and Simon (1973).

In summary, BACON is interesting primarily for its use of rule-space operators to create product, quotient, slope, and intercept terms and for its ability to recast the training instances on the basis of developed hypotheses.

CLS/ID3

CLS (Concept Learning System) is a learning algorithm devised by Earl Hunt (see Hunt, Marin, and Stone, 1966). It is intended to solve single-concept learning tasks and uses the learned concepts to classify new instances. A more recent version of the CLS algorithm, ID3, was developed by Ross Quinlan (1979, in press). In this article, we discuss the ID3 algorithm and its application to data compression and concept formation.

Like BACON, ID3 uses a feature-vector representation to describe the training instances. The features must each have only a small number of possible discrete values. Concepts are represented as decision trees. For example, if the features of size (small, large), shape (circle, square, and triangle), and color (red, blue) are used to represent the training instances, the concept of a *red circle* (of any size) could be represented as the tree shown in Figure D3b–1.

An instance is classified by starting at the root of the tree and making tests and following branches until a node is arrived at that indicates the class as YES or NO (see Article XI.D). For example, the instance (*large, circle, blue*) is classified as follows. Starting with the root node (shape), we follow the circle branch to the color node. From the color node we take the blue branch to a NO node indicating that this instance is not an instance of the concept of a red circle.

Decision trees are inherently disjunctive, since each branch leaving a decision node corresponds to a separate disjunctive case. The tree in Figure D3b–1,

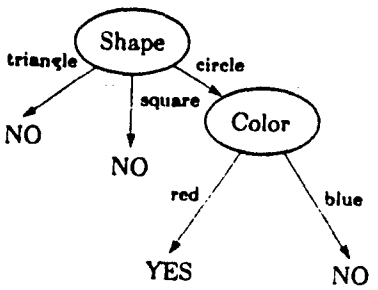


Figure D3b–1. Decision tree for the concept of a *red circle*.

for example, is equivalent to the predicate calculus expression:

$$\neg\text{SHAPE}(x, \textit{triangle}) \vee \neg\text{SHAPE}(x, \textit{square}) \vee$$
$$\text{SHAPE}(x, \textit{circle}) \wedge [\text{COLOR}(x, \textit{red}) \vee \neg\text{COLOR}(x, \textit{blue})] .$$

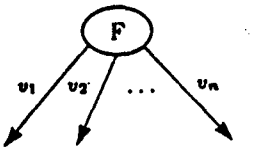
Consequently, decision trees can be used to represent disjunctive concepts such as *large circle or small square* (see Fig. D3b–2).

A drawback of decision trees is that there are many possible trees corresponding to any single concept. This lack of a unique concept representation makes it difficult to check that two decision trees are equivalent.

The CLS Learning Algorithm (as Used in ID3)

The CLS algorithm starts with an empty decision tree and gradually refines it, by adding decision nodes, until the tree correctly classifies all of the training instances. The algorithm operates over a set of training instances, C, as follows:

Step 1. If all instances in C are positive, then create a YES node and halt.
If all instances in C are negative, create a NO node and halt.
Otherwise, select (using some heuristic criterion) a feature, F, with values $v_1, \ldots, v_n$ and create the decision node:



Step 2. Partition the training instances in C into subsets $C_1, C_2, \ldots, C_n$ according to the values of V.

Step 3. Apply the algorithm recursively to each of the sets C_i.

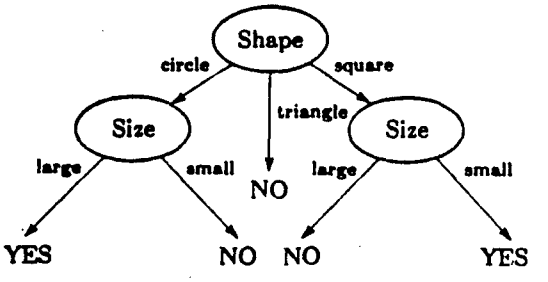


Figure D3b–2. Decision tree for a disjunctive concept.

The criterion used in step 1 by ID3 is to choose the feature that best discriminates between positive and negative instances. Hunt et al. (1966) describe several methods for estimating which feature is the most discriminatory. Quinlan chooses the feature that leads to the greatest reduction in the estimated entropy of information of the training instances in C. The exact criterion is to choose the feature F (with values $v_1, v_2, \ldots, v_n$) that minimizes

$$\sum_i \left[-V_i^+ \log_2 \left(\frac{V_i^+}{V_i^+ + V_i^-} \right) - V_i^- \log_2 \left(\frac{V_i^-}{V_i^+ + V_i^-} \right) \right],$$

where V_i^+ is the number of positive instances in C with $F = v_i$, and V_i^- is the number of negative instances in C with $F = v_i$.

This CLS algorithm can be viewed as a refinement-operator algorithm with only one operator:

> Specialize the current hypothesis by adding a new condition (a new decision node).

The CLS algorithm repeatedly examines the data during step 1 to decide which new condition should be added. The final decision tree developed by CLS is a generalization of the training instances, because in most cases not all features present in the training instances need to be tested in the tree. Thus, CLS begins with a very general hypothesis and gradually specializes it, by adding conditions, until a consistent tree is found.

The ID3 Learning Algorithm

The CLS algorithm requires that all of the training instances be available on a random-access basis during step 1. This places a practical limit on the size of the learning problems that it can solve. The ID3 algorithm (Quinlan, 1979, in press) is an extension to CLS designed to solve extremely large concept-learning problems. It uses an active experiment-planning approach to select a good *subset* of the training instances and requires only sequential access to the whole set of training instances. Here is an outline of the ID3 algorithm:

Step 1. Select a random subset of size W of the whole set of training instances (W is called the *window size*, and the subset is called the *window*).

Step 2. Use the CLS algorithm to form a rule to explain the current window.

Step 3. Scan through *all* of the training instances serially to find exceptions to the current rule.

Step 4. Form a new window by combining some of the training instances from the current window with some of the exceptions obtained in step 3.

Repeat steps 2 through 4 until there are no exceptions to the rule.

Quinlan has experimented with two different strategies for building the new window in step 4. One strategy is to retain all of the instances from the old window and add a user-specified number of the exceptions obtained from step 3. This gradually expands the window. The second strategy is to retain one training instance corresponding to each leaf node in the current decision tree. The remaining training instances are discarded from the window and replaced by exceptions. Both methods work quite well, although the second method may not converge if the concept is so complex that it cannot be discovered with *any* window of fixed size W.

Application of the ID3 Algorithm

The ID3 algorithm has been applied to the problem of learning classification rules for part of a chess end-game in which the only pieces remaining are a white king and rook and a black king and knight. ID3 has discovered rules to describe the concept of "knight's side lost (in at most) n moves" for $n = 2$ and $n = 3$. Table D3b–1 shows the results of these processes.

The features describing the board positions have been chosen to capture patterns believed to be relevant to the concept of *lost in n moves*. The actual raw data for the *lost in 2 moves* concept comprise 1.8 million distinct board positions. By choosing appropriate features, Quinlan was able to compress these into 428 distinct feature vectors. This is an excellent example of the importance to concept learning of good representation and of knowledge-based interpretation of the raw data. Quinlan (in press) points out that an important task for future learning research is to develop a program that can discover a good set of features.

Strengths and Weaknesses of CLS and ID3

The ID3 and CLS programs with their very simple representations and straightforward learning algorithms perform impressively on the single-concept

TABLE D3b–1
The Application of ID3 to a Chess End-game

Concept	Number of training instances	Number of features	Size of decision tree	Solution time
Lost in 2 moves	30,000	25	334 nodes	144 seconds[a]
Lost in 2 moves	428	23	83 nodes	3 seconds[a]
Lost in 3 moves	715	39	177 nodes	34 seconds[b]

[a] Using PASCAL implementation on a DEC KL-10.
[b] Using PASCAL implementation on a CDC CYBER 72.

learning problem. Much of the power of the ID3 algorithm derives from its sophisticated selection of training instances. This form of instance selection has been termed *expectation-based filtering* by Lenat, Hayes-Roth, and Klahr (1979). The basic value of expectation-based filtering is that it focuses the attention of the program on those training instances that violate its expectations. These are precisely the training instances needed to improve the program's representation of the concept being learned. Even this simple form of experiment planning allows ID3 to solve large learning problems efficiently.

One of the chief difficulties of the CLS/ID3 method is that the representation for learned concepts is a decision tree, and decision trees are difficult to check for equivalence. What is more important, it is difficult for people to understand the learned concept when it is expressed as a large decision tree.

References

The best discussion of BACON is Langley (1980). The ID3 algorithm is well described in Quinlan (in press).

D3c. Concept Learning by Generating and Testing Plausible Hypotheses

THE two model-driven approaches discussed in Article XIV.D1 on issues—generate-and-test and schema instantiation—have received little attention from people doing learning research. This article describes one method, developed by Dietterich and Michalski, that discovers a single concept from examples by model-driven generate and test. In spite of using only a very simple model, this method exhibits the strengths and weaknesses that are typical of model-driven methods: It is quite immune to noise but cannot incrementally modify its concept description as new training instances become available.

The INDUCE 1.2 Algorithm

Dietterich and Michalski (1981) address the problem of learning a single concept from positive training instances only. Their program, INDUCE 1.2, is intended to be applied in *structural-learning* situations, that is, situations in which each training instance has some internal structure. Winston's toy-block constructions, for example, are structural training instances; a toy-block construction is represented as a set of nodes connected by structural relations like ONTOP, TOUCH, and SUPPORTS (see Article XIV.D3a). Dietterich and Michalski's model, which guides the search for generalizations, expects the learned concept to be a conjunction involving both structural relations and ordinary features.

INDUCE 1.2 seeks to find a few concepts in the rule space, each of which covers all of the training instances while remaining as specific as possible. This learning problem is similar to the problem of finding the S set in the candidate-elimination algorithm. INDUCE 1.2, however, applies some model-based heuristics to drastically prune the S set so that only a few generalizations are discovered.

The program assumes that the training instances have been transformed so that they can be viewed as very specific points in the rule space (i.e., it uses the single-representation trick). A random sample of the training instances is chosen. These points in rule space serve as the starting points for a beam search upward through the rule space, that is, from the very specific training instances toward more general concepts. The concept descriptions are generalized by dropping conjunctive conditions and adding internal disjunctive options until they cover all of the training instances. By starting at the most specific points in the rule space and stopping as soon as it finds concepts that cover all of the training instances, INDUCE 1.2 is guaranteed to find the most specific concepts that cover the data.

The beam-search process has the following steps:

Step 1. *Initialize.* Set H to contain a randomly chosen subset of size W of the training instances (W is a constant called the *beam width*).

Step 2. *Generate.* Generalize each concept in H by dropping single conditions in all possible ways. This produces all the concept descriptions that are minimally more general than those in H. These form the new H.

Step 3. *Prune implausible hypotheses.* Remove all but W of the concept descriptions from H. The pruning is based on syntactic characteristics of the concept description, such as the number of terms and the user-defined cost of the terms. Another criterion is to maximize the number of training instances covered by each element of H.

Step 4. *Test.* Check each concept description in H to see if it covers all of the training instances. (This information was obtained previously in step 3.) If any concept does, remove it from H and place it in a set C of output concepts.

Repeat steps 2, 3, and 4 until C reaches a prespecified size limit or H becomes empty.

A schematic diagram of the beam-search process is shown in Figure D3c-1.

Extensions to the Basic Algorithm

Structural learning problems of the kind INDUCE 1.2 was designed to attack require binary (and higher order) predicates to represent the desired

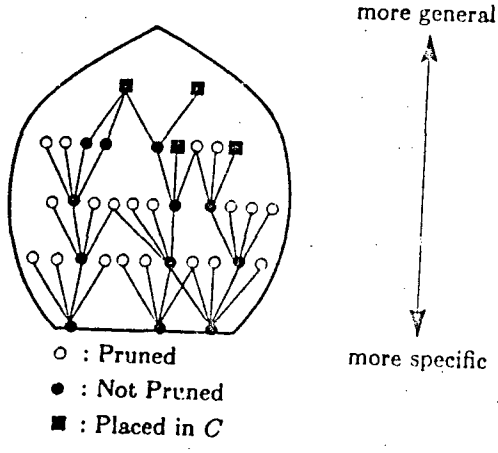


Figure D3c-1. A schematic diagram of INDUCE 1.2's beam search.

concepts. The binary predicates are needed to express relationships among the parts (e.g., toy blocks) that make up each training instance. In Winston's arch training instances, for example, binary predicates could be used to represent the fact that two blocks are touching—TOUCH(a, b)—or that one block is supporting another—SUPPORTS(a, b). Unary predicates and functions are, of course, still needed as well. Typically, they represent the attributes of the parts of an instance. In Winston's arches, for example, unary predicates could represent the size and shape of each block. The syntactic distinction between unary and binary predicates thus corresponds to a semantic distinction between feature values and binary relationships.

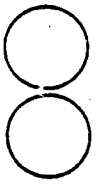
Although it is *possible* to represent structural relationships using only unary predicates or functions, such a representation is cumbersome and unnatural. Consequently, this distinction—by which binary and higher order predicates correspond to structural relationships and unary predicates and functions correspond to feature values—holds in most structural learning situations.

Dietterich and Michalski take advantage of this dichotomy to improve the efficiency of INDUCE 1.2's rule-space search. Two separate rule spaces are used. The first rule space, called the structure-only space, is the space of all concepts expressible using only the binary (and higher order) terms in the representation language. The training instances are abstracted into this space (by dropping all unary predicates and functions), and then the generate-and-test beam search is applied to this abstract rule space.

Once the set, C, of candidate structure-only concepts is obtained, each concept, c_i, in C is used to define a new rule space, consisting of all concepts expressible in terms of the attributes of the subobjects (e.g., blocks) referred to in c_i. This space can be represented with a simple feature-vector representation. The training instances are transformed into very specific points in this space, and another beam search is conducted to find a set, C', of plausible concept descriptions. The descriptions in C' specify the attributes for the subobjects referred to in c_i. Taken together, one concept in C' combined with c_i provides a complete concept description.

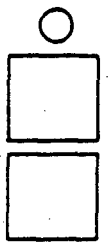
As an example of this two-space approach, consider the two positive training instances depicted below:

Instance 1.



$\exists\, u, v$: LARGE$(u) \wedge$ CIRCLE$(u) \wedge$
LARGE$(v) \wedge$ CIRCLE$(v) \wedge$ ONTOP(u, v).

Instance 2.



$\exists\, w, x, y$: SMALL(u) $\wedge$ CIRCLE(w) $\wedge$
LARGE(x) $\wedge$ SQUARE(x) $\wedge$
LARGE(y) $\wedge$ SQUARE(y) $\wedge$
ONTOP(w, x) $\wedge$ ONTOP(x, y).

When these two training instances are translated into the structure-only rule space, the following abstract training instances are obtained:

Instance 1'. $\exists\, u, v$: ONTOP(u, v).

Instance 2'. $\exists\, w, x, y$: ONTOP(w, x) $\wedge$ ONTOP(x, y).

The INDUCE 1.2 beam search discovers that $C = \{\text{ONTOP}(u, v)\}$ is the only, least general, structure-only concept consistent with the training instances. Now a new attribute-vector rule space is developed with the features of u and v:

$$\langle \text{SIZE}(u), \text{SHAPE}(u), \text{SIZE}(v), \text{SHAPE}(v) \rangle .$$

The training instances are translated to obtain:

Instance 1". (large, circle, large, circle).

Instance 2.1". (small, circle, large, square).

Instance 2.2". (large, square, large, square).

Notice that two alternative training instances are obtained from instance 2', since ONTOP(u, v) can match instance 2 in two possible ways (u bound to w, v bound to x; or u bound to x, v bound to y). During the beam search, only one of these two instances, 2.1" and 2.2", need be covered by a concept description for that description to be consistent.

The second beam search is conducted in this feature-vector space, and the concepts (large, $*$, large, $*$) and ($*$, circle, large, $*$) are found to be the least general concepts that cover all of the training instances ("$*$" indicates that the corresponding feature is irrelevant). By combining each of these feature-only concepts with the structure-only concept ONTOP(u, v), two overall consistent concept descriptions are obtained:

C_1: $\exists\, u, v$: ONTOP(u, v) $\wedge$ LARGE(u) $\wedge$ LARGE(v),

C_2: $\exists\, u, v$: ONTOP(u, v) $\wedge$ CIRCLE(u) $\wedge$ LARGE(v).

These correspond to the observations that in both instance 1 and instance 2 there are (C_1) "always a large object on top of another large object" and (C_2) "always a circle on top of a large object."

Strengths and Weaknesses of the INDUCE 1.2 Approach

The basic algorithm suffers from the absence of a strong model to guide the pruning of descriptions in step 3 and the termination of the search in step 4. The present syntactic criteria, of minimizing the number of terms in a proposed concept, minimizing the user-defined cost of the terms, and maximizing the number of training instances covered, are very weak. Dietterich and Michalski claim that domain-specific information could easily be applied at this point to improve the model-based pruning.

A second weakness is that step 2 involves exhaustive enumeration of all possible single-step generalizations of the hypotheses in H. This can be very costly in a large rule space. The method of plausible generate and test works best if the generator can be constrained to generate only plausible hypotheses. The generator in INDUCE 1.2 relies on a subsequent pruning step, which is quite costly.

A third weakness of the method is that, because it prunes its search, it is incomplete (see Dietterich and Michalski, 1981). It does not find all minimally general concepts in the rule space that cover all of the training instances.

As with all model-driven methods, this approach does not work well in incremental learning situations. All of the training instances must be available to the learning algorithm simultaneously.

The advantages of the algorithm are that it is faster and uses less memory than the full version-space approach. As with all model-based methods, INDUCE 1.2 has good noise immunity. In particular, if INDUCE 1.2 is to be given noisy training instances, then step 4 can be modified to include in C the concepts that cover *most*, rather than all, of the training instances.

References

Dietterich and Michalski (1981) describe INDUCE 1.2.

D3d. Schema Instantiation

SCHEMA-INSTANTIATION techniques have been used in many AI systems that perform comprehension tasks such as image interpretation, natural-language understanding, and speech understanding. Few learning systems have employed schema-instantiation methods, however. These methods are useful when a system has a substantial number of constraints that can be grouped together to form a schema, an abstract skeletal rule. The search of the rule space can then be guided to only those portions of the space that *fit* one of the available schemas. In this section, we describe one learning system, SPARC, that uses schema instantiation to discover single concepts.

Discovering Rules in Eleusis with SPARC

Dietterich's (1979) SPARC system attempts to solve a learning problem that arises in the card game Eleusis. Eleusis (developed by Robert Abbott, 1977; see also Gardner, 1977) is a card game in which players attempt to discover a secret rule invented by the dealer. The secret rule describes a linear sequence of cards. In their turns, the players attempt to extend this sequence by playing additional cards from their hands. The dealer gives no information aside from indicating whether or not each play is consistent with the secret rule. Players are penalized for incorrect plays by having cards added to their hands. The game ends when a player empties his hand.

A record of the play is maintained as a layout (see Fig. D3d-1) in which the top row, or *main line*, contains all of the correctly played cards in sequence. Incorrect cards are placed in *side lines* below the main-line card that they follow. In the layout shown in Figure D3d-1, the first card correctly played was the 3 of hearts (3H). This was followed by another correct play, the 9 of spades (9S). Following the 9, two incorrect plays were made (JD and 5D) before the next correct card (4C) was played successfully.

Main line:	3H	9S	4C	9D	2C	10D	8H	7H	2C	5H
Side lines:		JD		AH	AS			10H		
		5D		8H	10S					
				QD						

If the last card is odd, play black; if the last card is even, play red.

Figure D3d-1. An Eleusis layout and the corresponding secret rule.

The scoring in Eleusis encourages the dealer to choose rules of intermediate difficulty. The dealer's score is determined by the difference between the highest and lowest scores of the players. Thus, a good rule is one that is easy for some players and hard for others.

Schemas in Eleusis

In ordinary play of Eleusis, certain classes of rules have been observed. Dietterich has identified three rule classes and developed a parameterized schema for each:

1. **Periodic rules.** A periodic rule describes the layout as a sequence of repeating features. For example, the rule *Play alternating red and black cards* is a periodic rule. Dietterich's rule schema for this class can be described as an N tuple of conjunctive descriptions:

   $$(C_1, C_2, \ldots, C_N).$$

   The parameter N is the length of the period (the number of cards before the period starts to repeat). The above-mentioned periodic rule would be represented as a 2-tuple:

   $$(\text{RED}(card_i), \text{BLACK}(card_i)).$$

   More complex periodic rules may refer to the previous periods. Thus, the rule

   $$(\text{RANK}(card_i) \geq \text{RANK}(card_{i-1}), \text{RANK}(card_i) \leq \text{RANK}(card_{i-1}))$$

   describes a layout composed of alternating ascending and descending sequences of cards.

2. **Decomposition rules.** A decomposition rule describes the layout by a set of *if-then* rules. For example, the rule *If the last card is odd, play black; if the last card is even, play red* is a decomposition rule. The rule schema for this class requires that the set of *if-then* rules have single conjunctions for the *if* and *then* parts of each rule. The *if* parts must be mutually exclusive, and they must span all possibilities. The above-mentioned rule can be written as:

   $$\begin{aligned} \text{ODD}(card_{i-1}) &\Rightarrow \text{BLACK}(card_i) \vee \\ \text{EVEN}(card_{i-1}) &\Rightarrow \text{RED}(card_i). \end{aligned}$$

3. **Disjunctive rules.** The third class of rules includes any rules that can be represented by a single disjunction of conjunctions (i.e., an expression in disjunctive normal form, or DNF). For example, the rule *Play a card of the same rank or the same suit as the preceding card* is a DNF rule. This is represented as:

   $$\text{RANK}(card_i) = \text{RANK}(card_{i-1}) \vee \text{SUIT}(card_i) = \text{SUIT}(card_{i-1}).$$

Each schema has a few parameters that control its application. The N (length of period) parameter of the period schema has already been described. Each schema also has a parameter L, called the lookback parameter, that indicates how many cards back into the past the rule may consider. Thus, when $L = 0$, no preceding cards are examined. When $L = 1$, the features of the current card are compared with the previous card, and expressions such as $\text{RANK}(card_i) \geq \text{RANK}(card_{i-1})$ are permitted. Larger values of L provide for even further lookback.

Searching the Rule Space Using Schemas

Each schema can be viewed as having its own rule space—the set of all rules that can be obtained by instantiating that schema. SPARC uses the single-representation trick to reformulate the layout as a set of very specific rules for each of the schema-specific rule spaces. The overall algorithm works as follows:

Step 1. *Parameterize a schema.* SPARC chooses a schema and selects particular values for the parameters of that schema.

Step 2. *Interpret the training instances.* Transform the training instances (i.e., the cards in the layout) into very specific rules that fit the chosen schema.

Step 3. *Instantiate the schema.* Generalize the transformed training instances to fit the schema. SPARC uses a schema-specific algorithm to accomplish this step.

Step 4. *Evaluate the instantiated schema.* Determine how well the schema fits the data. Poorly fitting rules are discarded.

SPARC conducts a depth-first search of the space of all parameterizations of all schemas up to a user-specified limit on the magnitudes of the parameters. Notice that a separate interpretation step is required for each parameterized schema.

When these steps are applied to the game shown in Figure D3d-1, for example, step 1 eventually chooses the decomposition schema with $L = 1$. Step 2 then converts the training instances into very specific rules in the corresponding rule space. In this case, the first five cards produce the training instances shown below. The instances are represented by the feature vector (RANK, SUIT, COLOR, PARITY) to describe each card. (SPARC actually generates 24 features to describe each training instance.)

Instance 1 (positive). ⟨3, *hearts*, *red*, *odd*⟩ ⇒ ⟨9, *spades*, *black*, *odd*⟩.
Instance 2 (negative). ⟨9, *spades*, *black*, *odd*⟩ ⇒ ⟨*jack*, *diamonds*, *red*, *odd*⟩.
Instance 3 (negative). ⟨9, *spades*, *black*, *odd*⟩ ⇒ ⟨5, *diamonds*, *red*, *odd*⟩.
Instance 4 (positive). ⟨9, *spades*, *black*, *odd*⟩ ⇒ ⟨4, *clubs*, *black*, *even*⟩.

Step 3 produces the following instantiated schema (with irrelevant features indicated by *):

$$(*, *, *, odd) \Rightarrow (*, *, black, *) \vee (*, *, *, even) \Rightarrow (*, *, red, *).$$

Step 4 determines that this rule is entirely consistent with the training instances and is syntactically simple. Consequently, the rule is accepted as a hypothesis for the dealer's secret rule.

The schema-instantiation method works well when step 3, the schema-instantiation step, is easy to accomplish. A good schema provides many constraints that limit the size of its rule space. In SPARC, for example, the periodic and decomposition schemas require that their rules be made up of single conjuncts only. This is a strong constraint that can be incorporated into the model-fitting algorithm. On the other hand, the DNF schema provides few constraints and, consequently, an efficient instantiation algorithm could not be written. The general-purpose A^q algorithm (see Article XIV.D3a) was used instead.

Strengths and Weaknesses of SPARC

The schema-instantiation method used in SPARC was able to find plausible Eleusis rules very quickly. This is the primary advantage of the schema-instantiation approach—large rule spaces can be searched quickly. A second advantage of this approach is that it has good noise immunity. The schema-instantiation process has access to the full set of training instances, and, thus, it can use statistical measures to guide the search of rule space.

There are three important disadvantages of the schema-instantiation method as used in SPARC. First, it is difficult to isolate a group of constraints and combine them to form a schema. The three schemas in SPARC, although they cover most "secret rules" pretty well, are known to miss some important rules. The task of coming up with new schemas, however, is particularly difficult. A second problem with the schema-instantiation approach is that special schema-instantiation algorithms must be developed for each schema. This makes it difficult to apply the approach in new domains. The third disadvantage is that separate interpretation methods need to be developed for each schema. This was less of a problem in the Eleusis domain, because the interpretation processes for the different schemas were very similar.

References

Dietterich (1979) is the original description of the SPARC program. Dietterich (1980) is a more accessible source. See also Dietterich and Michalski (in press).

D4. Learning Multiple Concepts

A FEW AI learning systems have been developed that discover a *set* of concepts from training instances. These systems perform tasks, such as disease diagnosis and mass-spectrometer simulation, for which a single concept or classification rule is not sufficient.

To understand the problems of learning multiple concepts, it is helpful to review single-concept learning. In single-concept learning (see Sec. XIV.D3), the learning element is presented with positive and negative instances of some concept, and it must find a concept description that effectively partitions the space of all instances into two regions: positive and negative. All instances in the positive region are believed by the learning system to be examples of the single concept (see Fig. D4–1).

In multiple-concept learning, the situation is slightly more complicated. The learning element is presented with training instances that are instances of several concepts, and it must find several concept descriptions. For each concept description, there is a corresponding region in the instance space (see Fig. D4–2). An important multiple-concept learning problem is the problem of discovering disease-diagnosis rules from training instances. The learning element is presented with training instances that each contain a description of a patient's symptoms and the proper diagnosis as determined by a doctor. The program must discover a set of rules of the form:

⟨description of symptoms for disease A⟩ ⇒ Disease is A,
⟨description of symptoms for disease B⟩ ⇒ Disease is B,
⋮
⟨description of symptoms for disease N⟩ ⇒ Disease is N.

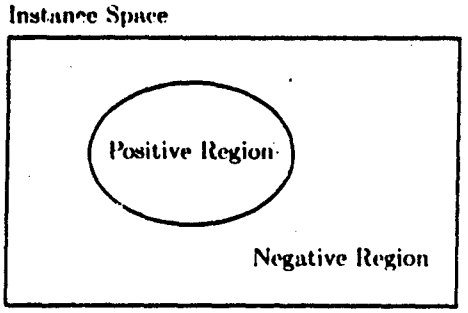


Figure D4–1. A single concept viewed as a region of the instance space.

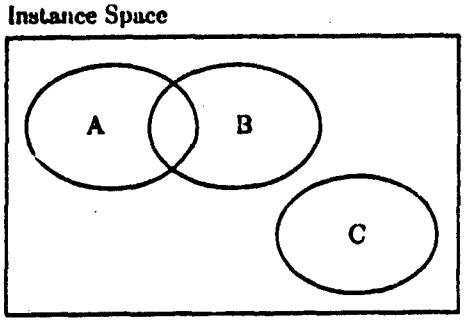


Figure D4–2. Regions of the instance space corresponding to different rules.

The left-hand side of each rule is a concept description that corresponds to a region in the instance space of all possible symptoms (see Fig. D4–2). Any patient whose symptoms fall in region A, for example, will be diagnosed as having disease A.

An important issue arising in multiple-concept learning is the problem of overlapping concept descriptions—that is, overlapping left-hand sides of diagnosis rules. In Figure D4–2, for example, when a patient's symptoms fall in the area where regions A and B overlap, the system will diagnose the patient as having both diseases A and B. This overlap may be correct, since there are often cases in which a patient has more than one disease simultaneously. On the other hand, it is often the case in multiple-concept problems that the various classes are intended to be mutually exclusive. For example, if, instead of diagnosing diseases, the performance task is to classify images of handwritten characters, it is important that the system arrive at a unique classification for each character.

The problem of overlap among multiple concepts can lead to *integration* problems, as described in Article XIV.A. When a new rule or concept is added to the knowledge base in a multiple-concept system, it may be necessary to modify the left-hand sides of existing rules, particularly if the concept classes are intended to be mutually exclusive.

The systems described in this section differ from those described in the Section XIV.D5 on multiple-step tasks in that the performance tasks discussed here can all be accomplished in a single step. The various disease-classification rules, for example, can be applied simultaneously to classify a patient's symptoms. Tasks for which this is not the case—like playing checkers or solving symbolic integration problems—are discussed in Section XIV.D5.

We first discuss the work of Michalski and his colleagues on the AQ11 program, which learns a set of classification rules for the diagnosis of soybean

diseases. Second, we describe the Meta-DENDRAL system, which learns a set of cleavage rules that describe the operation of a chemical instrument called the mass spectrometer. Finally, the AM system, which discovers new concepts in mathematics, is discussed in some detail. Since these systems do not all address the same learning problem, we begin each article with a description of the particular learning problem being attacked and then discuss the methods employed to accomplish the learning.

D4a. AQ11

MICHALSKI and his colleagues (Michalski and Larson, 1978; Michalski and Chilausky, 1980) have developed several techniques for learning a set of classification rules. The performance element that applies these rules is a pattern classifier that takes an unknown pattern and classifies it into one of n classes (see Fig. D4a-1). Many performance tasks, such as optical character recognition and disease diagnosis, have this form.

The classification rules are learned from training instances consisting of sample patterns and their correct classifications. For the classifier to be as efficient as possible, the classification rules should test as few features of the input pattern as necessary to classify it reliably. This is particularly relevant in areas like medicine, where the measurement of each additional feature of the input pattern may be very costly and dangerous. Consequently, Michalski's learning program AQ11 (Michalski and Larson, 1978) seeks to find the *most general rule* in the rule space that discriminates training instances in class c_i from all training instances in all other classes c_j ($i \neq j$). Dietterich and Michalski (1981) call these *discriminant descriptions* or *discrimination rules*, since their purpose is to discriminate one class from a predetermined set of other classes.

Using the A^q Algorithm to Find Discrimination Rules

The representation language used by Michalski to represent discrimination rules is VL_1, an extension of the propositional calculus. VL_1 is a fairly rich

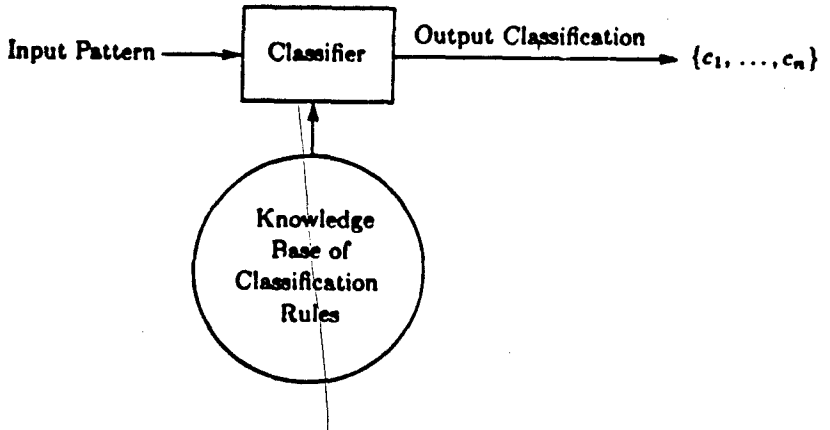


Figure D4a-1. The n-category classification task.

language that includes conjunction, disjunction, and set-membership operators. Consequently, the rule space of all possible VL_1 discrimination rules is quite large. To search this rule space, AQ11 uses the A^q algorithm, which is nearly equivalent to the repeated application of the candidate-elimination algorithm (see Article XIV.D3a). AQ11 converts the problem of learning discrimination rules into a series of single-concept learning problems. To find a rule for class c_i, it considers all of the known instances in class c_i as positive instances and all other training instances in all of the remaining classes as negative instances. The A^q algorithm is then applied to find a description that covers all of the positive instances without covering any of the negative instances. AQ11 seeks the most general such description, which corresponds to a necessary condition for class membership. Figure D4a-2 shows schematically how this works. The dots represent known training instances, and the circle represents the set of possible training instances that are covered by the description of class c_1.

For each class c_i, such a "concept" is discovered. The result is shown schematically in Figure D4a-"

Note that the discrimination rules may overlap in regions of the instance space that have not yet been observed. This overlap is useful because it allows the performance element to be somewhat conservative. In the areas in which the discrimination rules are ambiguous (i.e., overlap), the performance element can report this to the user rather than assign the unknown instance to one arbitrarily chosen class.

AQ11 also has a method for finding a nonoverlapping set of classification rules. Since the A^q algorithm uses the single-representation trick, it can accept not only single points in the instance space (as represented by very specific points in the rule space) but also generalized "instances" that are conjuncts

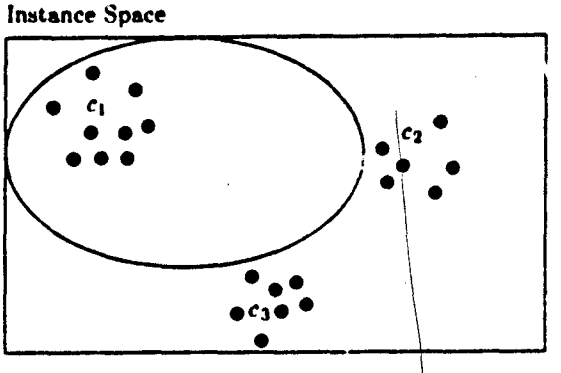


Figure D4a-2. Learning c_1 by treating all other classes as negative instances.

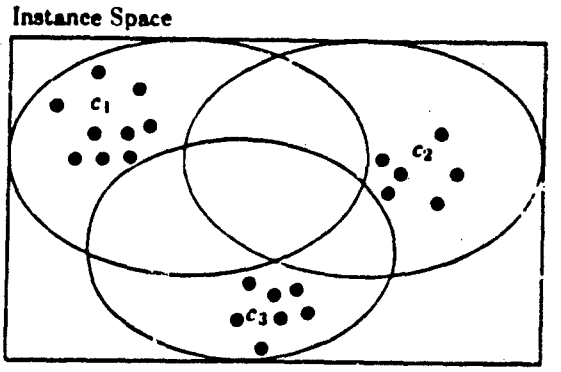


Figure D4a–3. Finding single concepts for each class.

in the rule space corresponding to *sets* of training instances. This allows AQ11 to treat the *concept descriptions* themselves as negative examples when it is learning the concept description for a subsequent class. Thus, in order to obtain a nonoverlapping set of discrimination rules, AQ11 takes as its positive instances all known instances in c_i and as its negative instances all known instances in c_j ($j \neq i$) *plus* all conjuncts that make up the discrimination rules for previously processed classes c_k ($k < i$). The resulting disjoint rules are shown schematically in Figure D4a–4 (assuming the classes were processed in the order c_1, c_2, c_3).

The rules that are developed split up the unobserved part of the instance space in such a way that c_1 gets the largest share, c_2 covers any space not covered by c_1, c_3 covers any space not covered by c_1 or c_2, and so on. The way in which the space is divided up depends on the order in which the classes are

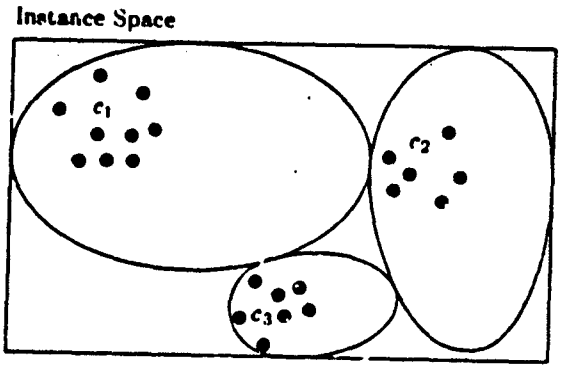


Figure D4a–4. Finding nonoverlapping classification rules.

processed. A performance element that uses such a disjoint set of concepts will be reckless in the sense that it will assign an unknown instance to an arbitrary class. The classifier arbitrarily prefers c_1 to c_2, c_2 to c_3, and so on.

The discrimination rules developed by AQ11 correspond (roughly) to the set of most general descriptions consistent with the training instances—the G set in the candidate-elimination algorithm (see Sec. XIV.D3). In many situations, it is also good to develop, for each class c_i, the most specific (S-set) description of that class. This permits very explicit handling of the unobserved portions of the space. Figure D4a–5 shows such a set of descriptions.

When S and G sets are both available, the performance element can choose among definite classification (the instance is covered by the S set), probable classification (the instance is covered by only one G set), and multiple classification (the instance is covered by several G sets). AQ11 has the ability to calculate an approximate S set for each class. When the description of the class is disjunctive, the S set is also disjunctive.

Applications of AQ11

The AQ11 program has been applied to the problem of discovering disease-diagnosis rules for 15 soybean diseases (Michalski and Chilausky, 1980). Here is an example of a classification rule for the disease Rhizoctonia root rot obtained by the overlapping-concept approach discussed above:

leaves ∈ {normal} ∧ stem ∈ {abnormal} ∧
stem cankers ∈ {below soil line} ∧ canker lesion color ∈ {brown} ∨

leaf malformation ∈ {absent} ∧ stem ∈ {abnormal} ∧
stem cankers ∈ {below soil line} ∧ canker lesion color ∈ {brown}
⇒ Rhizoctonia root rot.

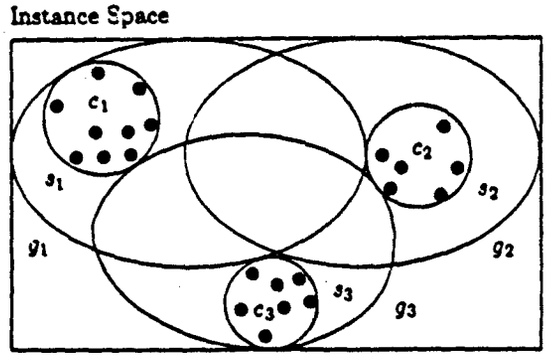


Figure D4a–5. Learning both the G and S set descriptions for each class.

An interesting experiment was conducted as part of the soybean disease project. The goal was to compare the quality of rules obtained through consultation with expert plant pathologists with rules developed by learning from examples. Descriptions of 630 diseased soybean plants were entered into the computer (as feature vectors involving 35 features) along with an expert's diagnosis of each plant. A special instance-selection program, ESEL, was used to select 290 of the sample plants as training instances. ESEL attempts to select training instances that are quite different from one another—instances that are "far apart" in the instance space. The remaining 340 instances were set aside to serve as a testing set for comparing the performance of the machine-derived rules with the performance of the expert-derived rules.

AQ11 was then run on the 290 training instances to develop overlapping rules such as the rule above. Simultaneously, the researchers consulted with the plant pathologist to obtain a set of rules. They adopted the standard knowledge-engineering approach of interviewing the expert and translating his expertise into diagnosis rules. The expert insisted on using a description language that was somewhat more expressive than the language used by AQ11. The expert's rules, for example, listed some features as necessary and other features as confirmatory; AQ11 was unable to make such a distinction.

As a consequence of the differing description languages, slightly differing performance elements had to be developed to apply the two sets of rules, and each performance element was adjusted to get the best performance from its classification rules. Surprisingly, the computer-generated rules outperformed the expert-derived rules. Despite the fact that the expert-derived rules were expressed in a more powerful language, the machine-generated rules gave the correct disease top ranking 97.6% of the time, compared to only 71.8% for the expert-derived rules. Overall, the machine-generated rules listed the correct disease among the possible diagnoses 100% of the time, in contrast to 96.9% for the expert's rules. Furthermore, the computer-derived rules tended to list fewer alternative diagnoses. The conclusion of the experiment was that automatic rule induction can, in some situations, lead to more reliable and more precise diagnosis rules than those obtained by consultation with the expert.

References

Michalski and Larson (1978) describe the AQ11 and ESEL programs in detail. The soybean work is described in Michalski and Chilausky (1980).

D4b. Meta-DENDRAL

META-DENDRAL (Buchanan and Mitchell, 1978) is a program that discovers rules describing the operation of a chemical instrument called a *mass spectrometer*. The mass spectrometer is a device that bombards small chemical samples with accelerated electrons, causing the molecules of the sample to break apart into many charged fragments. The masses of these fragments can then be measured to produce a *mass spectrum*—a histogram of the number of fragments (also called the *intensity*) plotted against their mass-to-charge ratio (see Fig. D4b-1).

An analytic chemist can infer the molecular structure of the sample chemical through careful inspection of the mass spectrum. The Heuristic DENDRAL program (see Sec. VII.C2, in Vol. II) is able to perform this task automatically. It is supplied with the chemical formula (but not the *structure*) of the sample and its mass spectrum. Heuristic DENDRAL first examines the spectrum to obtain a set of constraints. These constraints are then supplied to CONGEN, a program that can generate all possible chemical structures satisfying the constraints. Finally, each of these generated structures is tested by running it through a mass-spectrometer simulator. The simulator applies a set of *cleavage rules* to predict which bonds in the proposed structure will be broken. The result is a simulated mass spectrum for each candidate structure. The simulated spectra are compared with the actual spectrum, and the structure whose simulated spectrum best matches the actual spectrum is ranked as the most likely structure for the unknown sample.

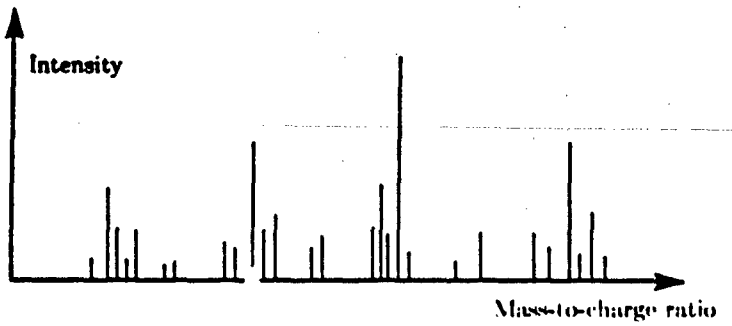


Figure D4b-1. A mass spectrum.

The Learning Problem

Meta-DENDRAL was designed to serve as the learning element for Heuristic DENDRAL. (For an alternate view of Meta-DENDRAL as an expert system, see Article VII.C2c, in Vol. II.) Its purpose is to discover new cleavage rules for DENDRAL's mass-spectrometer simulator. These rules are grouped according to *structural families*. Chemists have noted that molecules that share the same structural skeleton behave in similar ways inside the mass spectrometer. Conversely, molecules with vastly different structures behave in vastly different ways. Thus, no single set of cleavage rules can accurately describe the behavior of all molecules in the mass spectrometer.

Figure D4b–2 shows an example of a structural skeleton for the family of monoketoandrostanes. Particular molecules in this family are constructed by attaching keto groups (OH) to any of the available carbon atoms in the skeleton.

The learning problem addressed by Meta-DENDRAL is to discover the cleavage rules for a particular structural family. The problem can be stated as follows:

Given: (a) A representation language for describing molecular structures and substructures; and

(b) A training set of known molecules, chosen from a single structural family, along with their structures and their mass spectra;

Find: A set of cleavage rules that characterize the behavior of this structural family in the mass spectrometer.

This learning problem is difficult because it contains two sources of ambiguity. First, the mass spectra of the training molecules are noise-ridden. There may be falsely observed fragments (false positives) and important fragments that may not have been observed (false negatives). Second, the cleavage rules need

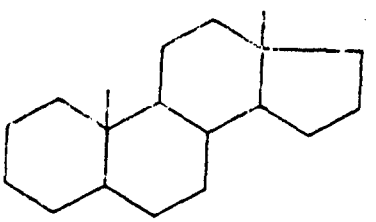


Figure D4b–2. The structural skeleton for the monoketoandrostane family.

not be entirely consistent with the training instances. A rule that correctly predicts a cleavage in more than half of the molecules can be considered to be acceptable; the rules need not be cautious. It is safer—from the point of view of DENDRAL's simulation task—to predict cleavages that do not occur than it is to fail to predict cleavages that do occur.

Meta-DENDRAL's representation language corresponds to the ball-and-stick models used by chemists. The molecule is represented as an undirected graph in which nodes denote atoms and edges denote chemical bonds. Hydrogen atoms are not included in the graph. Each atom can have four features: (a) the atom type (e.g., carbon, nitrogen), (b) the number of nonhydrogen neighbors, (c) the number of hydrogen atoms that are bonded to the atom, and (d) the number of double bonds in which the atom participates. A cleavage rule is expressed in terms of a *bond environment*—a portion of the molecular structure surrounding a particular bond. The bond environment makes up the condition part of a cleavage rule. The action part of the rule specifies that the designated bond will cleave in the mass spectrometer. Figure D4b–3 shows a typical cleavage rule.

The performance element (the simulator) applies the production rule by matching the left-hand-side bond environment to the molecular structure that is undergoing simulated bombardment. Whenever the left-hand-side pattern is matched, the right-hand-side predicts that the bond designated by * will break.

The Interpretation Problem and the Subprogram INTSUM

Meta-DENDRAL employs the method of model-driven generate-and-test to search the rule space of possible cleavage rules. Before it can carry out this search, however, it must first interpret the training instances and convert them into very specific points in the rule space (i.e., into very specific cleavage rules).

$x—y—z—w \Rightarrow x—y * z—w$

Node	Atom type	Neighbors	H-neighbors	Double bonds
x	carbon	3	1	0
y	carbon	2	2	0
z	nitrogen	2	1	0
w	carbon	2	2	0

Figure D4b–3. A typical cleavage rule.

The interpretation process is accomplished by the subprogram INTSUM (INTerpretation and SUMmary). Recall that the training instances have the form:

⟨whole molecular structure⟩ ⇒ ⟨mass spectrum⟩.

INTSUM seeks to develop a *set* of very specific cleavage rules of the form:

⟨whole molecular structure⟩ ⇒ ⟨one designated broken bond⟩.

To make this conversion, INTSUM must hypothesize which bonds were broken to produce which peaks in the spectrum. It accomplishes this by means of a "dumb" version of the DENDRAL mass-spectrometer simulator. Since Meta-DENDRAL is attempting to discover cleavage rules for this particular structural class, it cannot use those same cleavage rules to drive the simulation. Instead, a simple *half-order theory* of mass spectrometry is adopted.

The half-order theory describes the action of the mass spectrometer as a sequence of complete fragmentations of the molecule. One fragmentation slices the molecule into two pieces. A subsequent fragmentation may further split one of those two pieces to create two smaller pieces, and so on. After each fragmentation, some atoms from one piece of the molecule may migrate to the other piece (or be lost altogether). The half-order theory places certain constraints on this *split-and-migrate* process. It says that all bonds will break in the molecule *except* the following:

1. Double and triple bonds do not break;
2. Bonds in aromatic rings do not break;
3. Two bonds involving the same atom do not break simultaneously;
4. No more than three bonds break simultaneously;
5. At most, only two fragmentations occur (one after the other);
6. No more than two rings can be split as the result of both of the fragmentations.

Constraints are also placed on the kinds of migrations that can occur:

1. No more than two hydrogen atoms migrate after a fragmentation;
2. At most, one H_2O is lost;
3. At most, one CO is lost.

The parameters of the theory are flexible and can be adjusted by the user of Meta-DENDRAL.

Based on this theory, INTSUM simulates the bombarding and cleaving of the molecular structures provided in the training instances. The result is a simulated spectrum in which each simulated peak has an associated record of the bond cleavages that caused that peak to appear. Each simulated peak is compared with the actual observed peaks. If their masses match,

then INTSUM infers that the "cause" of the simulated peak is a plausible explanation of the observed peak. If a simulated peak finds no matching observed peak, it is ignored. If an observed peak remains unexplained, it is also ignored. However, unexplained peaks are reported to the chemist. A large proportion of unexplained peaks would indicate that the half-order theory was inadequate to explain the operation of the mass spectrometer in this training instance.

The half-order theory contributes another source of ambiguity to the learning problem. The interpreted set of training instances can easily contain erroneous instances. INTSUM's half-order theory tends to predict cleavages that did not, in fact, occur. It is also not unusual for the half-order theory to fail to predict cleavages that did occur. Thus, the training instances that guide the rule space search are very noisy indeed.

The Search of the Rule Space

Meta-DENDRAL searches the rule space in two phases. First, a model-driven generate-and-test search is conducted by the RULEGEN subprogram. This is a fairly coarse search from which redundant and approximate rules may result. The second phase of the search is conducted by the RULEMOD subprogram, which cleans up the rules developed by RULEGEN to make them more precise and less redundant.

RULEGEN. This subprogram searches the rule space of bond environments in order from most general to most specific. The algorithm repeatedly generates a new set of hypotheses, H, and tests it against the (positive) training instances developed by INTSUM, as follows:

Step 1. *Initialize H to contain the most general bond environment.*

$x \bullet y$

Node	Atom type	Neighbors	H-neighbors	Double bonds
x	any	any	any	any
y	any	any	any	any

This bond environment matches every bond in the molecule and thus predicts that every bond will break. Since the most useful (i.e., most accurate) bond environment lies somewhere between this overly general environment ($x \bullet y$) and the overly specific, complete molecular structure (with specified bonds breaking), the program generates refined environments by successively specializing the H set.

Step 2. *Generate a new set of hypotheses.* Specialize the set H by making a change to all atoms at a specified distance (radius) from the $\bullet$ bond—the bond designated to break. The change can involve either adding new neighbor atoms or specifying an atom feature. All possible specializations are made for which there is supporting

evidence. The technique of modifying *all* atoms at a particular radius causes the RULEGEN search to be coarse.

Step 3. *Test the hypotheses against the training instances.* The bond environments in H are examined to determine how much evidence there is for each environment. An *improvement criterion* is computed for each environment that states whether the environment is more plausible than the parent environment from which it was obtained by specialization. Environments that are determined to be more plausible than their parents are retained. The others are pruned from the H set. If all specializations of a parent environment are determined to be less plausible than their parent, the parent is output as a new cleavage rule and is removed from H.

Repeat steps 2 and 3 until H is empty.

Figure D4b–4 shows a portion of the RULEGEN search tree. Horizontal levels in the tree correspond to the contents of the H set after each iteration. Starting with the root pattern, S_0, the *number-of-neighbors* attribute is specialized (i.e., the pattern graph is expanded) for each atom at distance zero from (adjacent to) the break to give pattern S_1. The *atom type* is then specified for atoms adjacent to the break in S_2 and for atoms one bond removed from the break in S_3. At each step, there are many other possible successors corresponding to assignments of other values to these same attributes or to other attributes.

The improvement criterion used in step 3 states that a daughter environment graph is more plausible than its parent graph if:

1. It predicts fewer fragmentations per molecule (i.e., it is more specific);

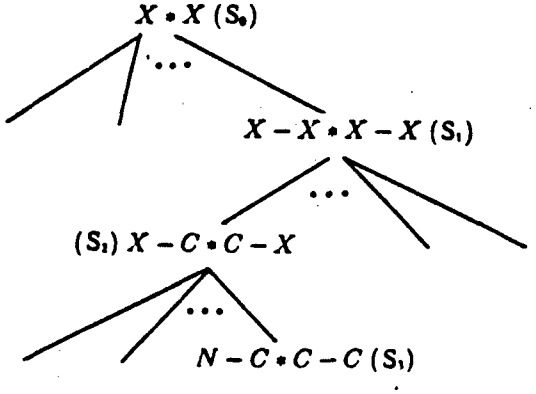


Figure D4b–4. A portion of the RULEGEN search tree.

2. It still predicts fragmentations for at least half of all of the molecules (i.e., it is sufficiently general);

3. It predicts fragmentations for as many molecules as its parent—unless the parent graph was "too general" in the sense that the parent predicts more than 2 fragmentations in some single molecule or on the average it predicts more than 1.5 fragmentations per molecule.

This algorithm assumes that the improvement criterion increases monotonically to a single maximum value (i.e., it is unimodal). This is usually true for the mass-spectrometry learning task. RULEGEN can thus be viewed as following monotonically increasing paths down through the partial order of the rule space until the criterion attains a local maximum value.

RULEMOD. The rules produced by RULEGEN are very approximate and have not been tested against negative evidence. RULEMOD improves these rules by conducting fine hill-climbing searches in the portions of the rule space near the rules located by RULEGEN. The subprogram RULEMOD proceeds in four steps:

Step 1. *Select a subset of important rules.* RULEGEN can produce rules that are different from one another but that explain many of the same data points. RULEMOD attempts to find a small set of rules that account for all of the data. Negative evidence is gathered for each rule by re-invoking the mass-spectrometer simulator. Each candidate rule is tested to see how many incorrect predictions are made as well as how many correct predictions. The rules are ranked according to a scoring function $(I \times (P + U - 2N)$, where I is the average intensity of the positively predicted peaks, P is the number of correctly predicted peaks, U is the number of correct peaks predicted uniquely by this rule and no other, and N is the number of incorrectly predicted peaks). The top-ranked rule is selected. All evidence peaks explained by that rule are removed, and the ranking and selection process is repeated until all positive evidence is explained or until the scores fall below a specified threshold.

Step 2. *Specialize rules to exclude negative evidence.* RULEMOD attempts to specialise the rules in order to exclude some negative evidence while retaining the positive evidence. For each candidate rule, RULEMOD attempts to fill in additional values for features that were left unspecified by RULEGEN. RULEMOD first examines all of the positive instances predicted by the candidate rule and obtains a list of all possible feature values that are common to all of the positive instances. Each of these feature values could individually be added to the rule without excluding any positive instances. RULEMOD attempts to select a mutually compatible set of values that will exclude a large amount of negative evidence.

The selection process uses a hill-climbing search. The feature value that excludes the largest number of negative instances is chosen and added to the candidate rule. Incompatible feature values are pruned from the list of possible refinements, and the process is repeated until further refinement is not possible or all negative evidence has been excluded.

Step 3. *Generalize rules to include positive evidence.* RULEMOD attempts to generalise the rules in order to include some positive evidence without including any new negative evidence. This is accomplished by relaxing the legal values for atom features that were specified by RULEGEN. RULEMOD examines each atom in the bond environment of the rule, starting with the atoms most distant from the • bond. It first checks to see if the whole atom can be removed from the graph without introducing any negative evidence. If it cannot, then a hill-climbing search is performed that iteratively removes the one atom feature that allows the rule to include the largest amount of new positive evidence without introducing any negative evidence. When the outermost atoms have been generalised as much as possible, RULEGEN examines the set of atoms that are one bond closer to the fragmentation site. This search continues until all possible changes have been made.

Step 4. *Select the final subset of rules.* The procedure used in step 1 is reapplied to select the final set of rules.

The key assumption made by RULEMOD is that RULEGEN has located rules that are approximately correct. RULEGEN points out the regions of the rule space in which detailed searches are needed.

Notice that RULEMOD must frequently invoke the mass-spectrometer simulator to assess the negative (incorrect) predictions of a proposed rule. INTSUM provides only positive training instances to RULEGEN. Negative instances are not provided to RULEGEN directly because there are many more negative instances than there are positive instances. This is a problem that frequently arises in systems that are attempting to explain why some particular set of events took place. Negative information must indicate everything that did *not* occur.

All three of Meta-DENDRAL's subprograms make use of some form of the mass-spectrometer simulator. These versions of the simulator are flexible and transparent. They allow the learning element to interpret the training instances and to reason about the performance of a hypothetical modification to the cleavage rules. Similar transparent performance elements are used in systems that learn to perform multiple-step tasks (see Sec. XIV.D5).

Experiment planning and the search of the instance space. Meta-DENDRAL does not conduct a search of the instance space. Such a search would require that Meta-DENDRAL select a molecular structure and ask the chemists to synthesize it and obtain its mass spectrum. To choose an

appropriate molecule, Meta-DENDRAL would need to invert the INTSUM process. Given a set of possible bond cleavages that it wanted to verify, Meta-DENDRAL would need to determine a molecule in which those bonds would cleave. Once the molecule was chosen, existing organic-synthesis programs could be used to plan the synthesis process (see Article VII.C4, in Vol. II). The chosen molecule might be difficult or impossible to synthesize. Instance-space searching was not incorporated into Meta-DENDRAL because of the complex and time-consuming nature of these procedures.

Another View of the Meta-DENDRAL Learning Algorithm

In the previous section, we discussed the RULEGEN/RULEMOD pair of subprograms as a coarse search followed by a fine search. Another view of this process is that RULEGEN converts a multiple-concept learning problem into a set of single-concept learning problems. This view regards the output of RULEGEN *not* as a set of rules but as a clustering of the training instances. Once RULEGEN has completed its search, the program knows approximately which training instances belong together as instances of a single cleavage rule. At this point, a single-concept learning algorithm could be applied to discover this rule directly from the RULEGEN-supplied cluster of training instances rather than by incremental modifications of the RULEGEN-supplied rule.

As part of his thesis work, Mitchell (1978) applied the candidate-elimination algorithm to this learning problem. Each approximate rule developed by RULEGEN was used to build a set of positive and negative training instances that were then processed by the version-space approach. This technique resulted in a better set of cleavage rules than those developed with RULEMOD. The version-space approach has the advantage of supporting incremental learning, so Mitchell's system can incorporate new training instances as they become available.

Strengths and Weaknesses of the Meta-DENDRAL System

Meta-DENDRAL is an effective learning system applied to a real-world domain. Meta-DENDRAL has discovered cleavage rules for five structural families of molecules. The system provides solutions to the problem of interpreting training instances and to the problem of learning in the presence of certain kinds of noise. These solutions are based on the incorporation into the program of a large amount of domain-specific knowledge. This knowledge enters the system in the form of the half-order theory of mass spectrometry (to guide interpretation) and in the use of a model-directed search of rule space.

The two-phase search of the rule space provides an efficient method for searching a large space and also suggests how a multiple-concept learning problem can be converted into a set of single-concept learning problems.

Among the weaknesses of the system are its domain-specific representation and the fact that much of the domain knowledge is buried in the code rather than represented as an explicit knowledge base.

References

Lindsay, Buchanan, Feigenbaum, and Lederberg (1980) present a comprehensive survey of the many programs developed during the DENDRAL project. Buchanan and Mitchell (1978) describe Meta-DENDRAL as an AI learning system. Mitchell (1978) discusses the application of the candidate-elimination algorithm to Meta-DENDRAL.

D4c. AM

AM is a computer program written by Douglas Lenat (1976) that discovers concepts in elementary mathematics and set theory. Unlike most of the learning systems described in this chapter, AM does not learn concepts for use in some performance task. Instead, it seeks simply to define and evaluate interesting concepts on the basis of a knowledge of mathematical aesthetics. It employs a refinement-operator approach (see Article XIV.D1) to conduct a heuristic search of a space of mathematical concepts.

AM starts with a substantial knowledge base of 115 concepts selected from finite set theory. As AM runs, it collects examples of these concepts, creates new concepts, and hypothesizes conjectures relating the concepts to each other. During one typical run of a few CPU hours' duration, AM defined about 200 new concepts, half of which were quite well known in mathematics. One of the synthesized concepts was equivalent to the concept of natural numbers. AM's knowledge of mathematical aesthetics led it to pursue this concept in depth, and it spent much time developing elementary number theory, including conjecturing the fundamental theorem of arithmetic (i.e., every number has a unique prime factorization). This impressive performance can be traced to AM's large body of knowledge about mathematics and its ability to apply this knowledge to discover new concepts and conjectures.

In this article, we first describe AM's architecture in terms of its representation for concepts and its control structure for deciding what tasks to perform. Then we change our perspective and show how AM can be viewed as searching an instance space and a concept space by the refinement-operator method. Third, we examine the initial contents of AM's knowledge base and review briefly the concepts that it discovered. Finally, we attempt to summarize the strengths and weaknesses of AM's approach to concept discovery.

AM's Architecture

AM is a blend of three powerful methods: *frame representation*, *production systems*, and heuristically guided *best-first search*. We discuss each of these in turn.

Frame representation. The concepts that AM discovers and manipulates are represented as frames (see Article III.C7, in Vol. I), each containing the same fixed set of slots. Each concept has slots for its *definition*, for known positive and negative *examples*, for links to other concepts that are *specializations* and *generalizations* of the concept, for telling the *worth* of the concept, and for several other things. Figure D4c-1 shows the frame representation of the PRIMES concept after it has been discovered and filled in by AM.

NAME: Prime Numbers

DEFINITIONS:

ORIGIN: Number-of-divisors-of(x) = 2

PREDICATE-CALCULUS: Prime(x) ≡ (∀z)(z | x ⇒ z = 1 ⊕ z = x)

ITERATIVE: (for x > 1): For i from 2 to sqrt(x), ¬(i | x)

EXAMPLES: 2, 3, 5, 7, 11, 13, 17

BOUNDARY: 2, 3

BOUNDARY-FAILURES: 0, 1

FAILURES: 12

GENERALIZATIONS: Nos., Nos. with an even no. of divisors,
Nos. with a prime no. of divisors

SPECIALIZATIONS: Odd Primes, Prime Pairs, Prime Uniquely-addables

CONJECTURES: Unique factorization, Goldbach's conjecture,
Extremes of Number-of-divisors-of

ANALOGIES:
Maximally divisible numbers are converse extremes of
Number-of-divisors-of,
Factor a nonsimple group into simple groups

INTEREST: Conjectures associating Primes with TIMES
and with Divisors-of

WORTH: 800

Figure D4c-1. AM's frame representation of the PRIMES concept.

The DEFINITIONS slot is the most important. It provides one or more LISP predicates that can be applied to determine whether something is an example of the concept. AM knows a concept when it has a definition for it. However, the frame representation allows AM to represent more knowledge about a concept than just its definition. The CONJECTURES, SPECIALIZATIONS, and GENERALIZATIONS slots, for example, all describe different ways in which concepts are related to each other. Furthermore, attached to each slot in a concept are *heuristic rules* (not shown in the figure) that can be executed to *fill in* the contents of a slot or to *check* the contents to see if they are correct. These heuristic rules form a production system that carries out the actual discovery process.

Production systems. AM operates as a *modified production system.* Each of the 242 heuristic rules attached to the concept slots of AM's knowledge base is written, as in all production systems, as a condition part and an action part. The condition part tells under what conditions the rule should be executed, and the action part carries out some task such as creating a new concept or finding examples of an existing concept. For instance, the following heuristic rule is attached to the EXAMPLES slot of the ANY-CONCEPT frame:

> If: The current task is "Fill in examples of X"
> and X is a specialization of some concept Y,
>
> Then: Apply the definition of X to each of the examples of Y
> and retain those that satisfy the definition.

The main difference between AM's production-system architecture and the standard recognize-act cycle is the way rules are selected for execution. Recall that in an ordinary production system, the condition part of each rule is compared to the contents of a working memory, and all rules that match are executed. In contrast, AM is much more selective about which rules it executes. It operates from an *agenda of tasks* of the form "Fill in (or check) slot S of concept C." Each task has a numeric "interestingness" rating. AM repeatedly selects the most interesting task from the agenda, gathers all heuristic rules relevant to performing that task, and executes those rules that are actually applicable.

To locate those heuristics that are relevant to the task "Fill in (or check) slot S of concept C," AM looks at slot S of concept C to see if it has any attached heuristics. If it does, those heuristics are executed. If not, AM examines relatives of concept C to see if any of them have heuristics that can be *inherited* by C and applied. For example, when AM is looking for rules relevant to the task "Fill in examples of sets," it finds no heuristics attached to the EXAMPLES slot of SETS. Consequently, it looks at concepts such as ANYCONCEPT, which are more general than SETS. The EXAMPLES slot of ANYCONCEPT has an attached heuristic that says:

> If: The current task is "Fill in examples of X"
> and X has a recursive definition,
>
> Then: Instantiate the base step of the recursion to get
> a boundary example.

When AM applies this heuristic rule, it creates the *null set* as a boundary EXAMPLE of SETS. Heuristics that are closely related to C are executed before heuristics of distant relatives.

A heuristic rule can do one or more of the following:

1. *Fill in slot S of some concept C.* This covers many activities, including finding new examples for a concept, proposing conjectures, and providing guidance for the search by modifying the WORTH slot of a concept.

2. *Check slot S of concept C.* The process of checking a slot involves verifying that the contents of the slot are correct and noticing interesting facts about a slot. Often, a rule will check a slot and notice that some new task should be performed as a result. For example, one rule notices that all of the examples of one concept, X, are also examples of a more specific concept, Y. It conjectures that X and Y are equivalent and proposes the task "Check examples of Y" to see if Y is actually equivalent to an even more specific concept, Z.

3. *Create new concepts.* New concepts are created by adding a new frame to the knowledge base and filling in the DEFINITIONS slot of the frame. Usually the WORTH slot is filled in as well.

4. *Add new tasks to the agenda.* Often, a rule will propose that a new task be added to the agenda. For example, a rule that creates a new concept, X, will propose the new task "Fill in examples of X." Most rules that generate examples of X will propose the task "Check examples of X."

5. *Modify the interestingness of a task on the agenda.* The numerical interestingness of a task is computed from a list of "reasons" for performing the task. Thus, a rule can add a new reason to an existing task. This is another way of providing guidance in the search for concepts and conjectures.

Best-first search. The procedure of always choosing the most interesting task from the agenda gives AM the flavor of best-first search. This search is well guided by heuristics that modify the INTERESTINGNESS and WORTH slots of concepts and that propose and justify agenda tasks. AM has 59 heuristics for assessing the interestingness of concepts and tasks. One rule, for example, says that a concept is interesting if each of its examples accidentally satisfies an otherwise rarely satisfied predicate P. (The satisfaction is accidental if the concept was not deliberately defined as the set of things satisfying P.)

Without heuristic guidance and the agenda mechanism, AM would be swamped by a combinatorial explosion of new concepts. However, the fact that it creates only 200 new concepts and that half of them are acceptable to a mathematician shows that its search is quite restrained. AM is an excellent example of the power of well-informed best-first search.

AM and the Two-space View of Learning

Thus far, we have discussed the architecture of AM. We now turn our attention to how this architecture is used to accomplish learning. Although its 242 heuristic rules are extremely varied and can perform many diverse functions, AM tends to behave as if it were executing the following loop:

Repeat:

Step 1. Select a concept to evaluate and generate examples of it.

Step 2. Check these examples looking for regularities. Based on the regularities,

(a) update the assessment of the interestingness of the concept,

(b) create new concepts, and

(c) create new conjectures.

Step 3. Propagate the knowledge gained (especially from new conjectures) to other concepts in the system.

In terms of the two-space view of learning, step 1 searches a space of instances, step 2 examines these instances and searches the space of concepts (the rule space) and conjectures, and step 3 performs bookkeeping to maintain the consistency and integration of the knowledge base. We examine each of these steps in more detail.

Searching the instance space. When a concept is created, AM knows very little about that concept aside from its LISP definition. In fact, when AM is first started up, none of its 115 initial concept frames has any examples filled in. Thus, one of the first tasks it must perform—in order to assess the value of the concepts and develop conjectures—is to gather examples (and negative examples) of its concepts. AM has more than 30 heuristic rules to guide this example-generating process. Here are some of the techniques they use:

1. *Symbolic instantiation of definitions.* Symbolic instantiation converts the definition of a concept into an example. Typically, each concept has, as one of its definitions, a recursive LISP predicate. The base step of this recursion can be instantiated to give an instance that satisfies the definition. For example, one of the definitions of the SET concept is:

```
(lambda (s)
  (or (= s {})
      (set.definition (remove (any-member s) s)))) .
```

   Since the first thing this definition checks is to see if s is the null set, we can conclude that the null set is an example of a set. Similarly, AM knows that *removing* is the opposite of *inserting*, so it can deduce that $\{\{\}\}$ is also a set by inserting $\{\}$ into itself.

2. *Generate and test.* Another approach used by the program is to generate examples and test them against the concept definition. In order to generate examples of some concept C, the program looks at "nearby" concepts in the knowledge base. For example, AM may look at generalizations of C (concepts more general than C), operations that have C in their range, cousins of C (concepts that share a common generalization or specialization with C), and even random LISP atoms from various internal lists inside AM (such as the list of users of the system).

3. *Inheritance of examples.* If concept C has other concepts that are more specialized than it, any example satisfying these more specialized concept definitions will satisfy C. Examples can thus be inherited "up" the

generalisation hierarchy. Similarly, negative examples can be inherited "down" the generalisation hierarchy.

4. *Applying the algorithm of the concept.* So-called active concepts (i.e., operators such as SET-UNION) have algorithms that compute an element in the range of the concept when given valid arguments from the domain. Thus, by randomly selecting domain items and applying these algorithms, AM can produce new examples. For instance, if $\{A\}$ and $\{B\}$ are sets, then SET-UNION.ALGORITHMS produces $\{A, B\}$, and the list $(\{A\}, \{B\}, \{A, B\})$ forms a positive example of SET-UNION.

5. *Reasoning by views or by analogy.* The VIEWS slot of a concept provides an algorithm for converting instances of one concept into instances of another. The ANALOGY slot gives less precise information about how instances of one concept are related to instances of another concept. AM can use these two slots to map existing examples into examples of the concept under construction.

When AM needs to fill in examples of a concept, it attempts to apply these methods until it has developed 26 examples of the concept (or until it has exhausted its time or space quota for the current task).

A particularly interesting feature of AM is its ability to locate the *boundary* of a concept. Examples of a concept are classified according to whether they are:

1. Normal positive examples,
2. Boundary positive examples,
3. Boundary negative examples (i.e., what Winston, 1970, calls *near misses*),
4. Normal negative examples, or
5. Just plain weird (i.e., have the wrong data structure).

Most examples produced by the above-mentioned techniques will turn out to be normal positive examples (or normal negative examples, if they do not satisfy the concept definition). Some of the example-generation techniques, however, are faulty. They can accidentally generate negative examples. A particular case is the VIEW slot of SETS that tells AM that it can view a bag as a set by changing the [] brackets (that represent a bag) to { } brackets. This does not always work (e.g., when the bag $[a, b, a]$ is viewed as that set $\{a, b, a\}$ which contains an impermissible duplicate element). When AM checks these examples against the definition of a set, it discovers that they fail. Such negative examples are classified as boundary negative examples.

Boundary positive examples can be found by such techniques as instantiating the base case of a recursion (which almost always produces a boundary case) or by taking boundary non-examples of more specialized concepts and determining that they satisfy the concept definition. Another technique is to take a normal positive example and progressively modify it until it fails to satisfy the definition. This isolates the boundary of the concept quite well.

By applying all of these techniques, AM is able to gather a good set of examples that can be used for analysis and generalization. AM can also assess how much effort was expended to obtain these examples. Thus, it can conclude that a predicate is "rarely satisfied" or "easily satisfied." All of these empirical data are used to drive the search of the rule space and the search for interesting conjectures.

Searching the rule space. The rule space for AM is the space of all possible instantiations of its concept frame. This is indeed an immense space. To search it, AM applies a refinement-operator method similar to the techniques employed by BACON and ID3 (see Article XIV.D3b). The current set of concept frames can be thought of as AM's current set of hypotheses. These hypotheses are repeatedly refined and extended by applying operators (i.e., heuristics) that create new concepts and conjectures.

AM has roughly 40 heuristics that create new concepts. These can be broken into two sets. One set of heuristics is general and can be applied to virtually any concept in AM. The second set is applicable only to functions and relations—active concepts that can be viewed as mapping elements from some domain set into some range set. The general methods are:

1. *Generalization.* AM implements, in some form, virtually all rules of generalization that have appeared in other AI programs. The dropping-condition, adding-option, and turning-constants-to-variables rules are all used. Also implemented is the technique of specializing a negative conjunct (e.g., $A \wedge \neg B$ is generalized to $A \wedge \neg B'$, where B' is more specific than B). AM can generalize expressions involving quantification, for example, converting $\exists x \in S : P(x)$ to $\exists x \in S' : P(x)$, where S' is a larger set than S. Since the definitions of concepts are typically recursive LISP functions, AM contains many rules of generalization that are applicable to recursion. For instance, a definition can be generalized by eliminating one of a conjoined pair of recursive calls or by disjoining a new recursive call. In particular, AM knows that if one recursive call involves CAR (or CDR), the other recursive call should use CDR (or CAR, respectively).

2. *Specialization.* AM also implements a wide variety of rules of specialization. These are the reversals of the rules of generalization mentioned above.

3. *Handling exceptions.* When a concept has a lot of exceptions (negative boundary examples), a new concept can be created whose instances are these negative examples. Also, AM can create the concept whose instances are those positive examples, but not boundary examples, of the original concept. This allows AM to represent the conjecture that all prime numbers are odd—except the number 2.

4. *Reasoning by analogy.* If J is a conjecture and J' is an analogous conjecture, then AM can create the concept $\{b' \mid J'(b')\}$ and also the concept

$\{b' \mid \neg J'(b')\}$, that is, the set of objects for which J' is true and the set of objects for which J' is false.

AM's concept-creation methods that apply to active concepts (mappings) usually produce new active concepts. New concepts can be created by the following:

1. *Generalization.* The domain and range of an existing concept can be expanded.
2. *Specialization.* The domain and range of an existing concept can be contracted (restricted).
3. *Inversion.* The inverse of an existing relation can be created. AM can also create interesting concepts such as the inverse image of an interesting subset of the range and the inverse image of an interesting value in the range.
4. *Composition.* Two functions $F(x)$ and $G(y)$ can be composed to obtain the new functions $F(G(y))$ and $G(F(x))$.
5. *Projection.* An existing multiple-argument function F can be projected onto a subset of its arguments. For example, Proj2($F(x, y)$) is just y.
6. *Coalesce.* The arguments of $F(x, y)$ can be coalesced to produce a new function, $G(x) = F(x, x)$.
7. *Canonization.* This method takes two predicates, P_1 and P_2, and defines a function, F, and a set, the range of F, such that $P_1(x, y) = P_2(F(x), F(y))$. If x and y are instances of concept C, then F maps C to the set of canonical C. Thus, P_2 applied to canonical C is the same as P_1 applied to C. AM uses this operation to invent NUMBERS by taking SAME-SIZE(x, y) as P_1, and EQUAL(x, y) as P_2, and applying them to bags to create the canonizing function SIZE-OF(x) and the concept of CANONICAL-BAGS (i.e., bags that contain only T). CANONICAL-BAGS can be interpreted as numbers.
8. *Parallel-replace and parallel-join.* These concept-creation operators come in many varieties and are used to create new concepts by repeated application of old concepts. Multiplication, for example, can be created by repeated addition (with the parallel-replace method).
9. *Permutation.* The arguments of a function or relation can be permuted to give a new function or relation.
10. *Cartesian product.* A new concept can be obtained by taking the Cartesian product of existing concepts.

Many of the refinement operators in this group (e.g., COALESCE, COMPOSITION) are also *concepts* defined in AM. It is perhaps only in mathematics that the means of study are also the objects of study.

Representing and proposing conjectures. Roughly 30 of AM's rules also propose conjectures based upon examination of the empirical data. Conjectures take one of the following forms:

1. C_1 is an example of C_2;
2. C_1 is a specialization (generalization) of C_2;
3. C_1 is equivalent to C_2;
4. C_1 is related by X to C_2 (where X is some predicate);
5. Operation C_1 has domain D or range R.

Most of these conjectures are discovered by performing rough statistical comparisons of examples. If all of the examples of C_1 are also examples of C_2, then AM conjectures that C_1 is a specialization of C_2. If AM is unable to find negative examples of C_1, it conjectures that C_1 is trivially true. If all examples of elements in the range of C_1 seem to be numbers, then AM conjectures that C_1 has numbers as its range. If all of the range elements of C_1 are equal to corresponding domain elements, then perhaps C_1 is the same as the identity function.

Conjectures, once proposed, are believed completely by AM. The relevant slots are changed, and the changes are propagated throughout the knowledge base. If two concepts are conjectured to be equivalent, they are merged and the space occupied by one is released. AM can also modify the LISP definitions to take advantage of new conjectures.

Propagating acquired knowledge. Several heuristics (including those that locate and generate examples) serve to propagate new information throughout the network of frames that constitutes AM's knowledge base. These are fairly straightforward and make heavy use of the three sets of inheritance links (IS-AN-EXAMPLE-OF/EXAMPLES, SPECIALIZATIONS/GENERALIZATIONS, DOMAIN/RANGE).

To complete our review of AM from the perspective of the two-space view of learning, we note that, although the example-generation techniques discussed above perform sophisticated instance selection, there is no corresponding need for complex interpretation routines like those found in Meta-DENDRAL. On the contrary, since mathematical objects are easily represented and manipulated in LISP, there is no need to convert them to some alternate representation. More sophisticated instance selection and interpretation routines would probably be needed for nonmathematical domains.

AM's Initial Knowledge Base

We now turn our attention to AM's actual performance. First we describe the knowledge that it started with, and then we give a summary of the concepts and conjectures it found.

AM's initial knowledge base contains the basic concept hierarchy shown in Figure D4c–2. In addition, beneath the concept of STRUCTURE are many important data structures: SETS, ORDERED SETS, BAGS, LISTS (i.e., ordered BAGS), and ORDERED PAIRS. Under the ACTIVITY concept are many operations such as SET-INTERSECT, SET-UNION, SET-DIFFERENCE, and SET-DELETION (and analogous operations for BAGS, ORDERED SETS, and LISTS). Also, several of the concept-creation operators such as PARALLEL-JOIN, RESTRICT, PROJECTION, and so forth, are included here. Under PREDICATES are the constant predicates TRUE and FALSE, as well as the concept of EQUALITY. Finally, the most important part of the initial knowledge base is the body of 242 heuristic rules attached to various concepts in this tree. Most of these were summarized above.

Results: AM as a Mathematician

Now we review the mathematics that AM explored. Throughout, AM acted alone, with a human user watching it and occasionally renaming some concepts for his (or her) own benefit. Like a contemporary historian summarizing the work of the Babylonian mathematicians, we will use present-day terms to describe AM's concepts, and we will criticize its behavior in light of our current knowledge of mathematics.

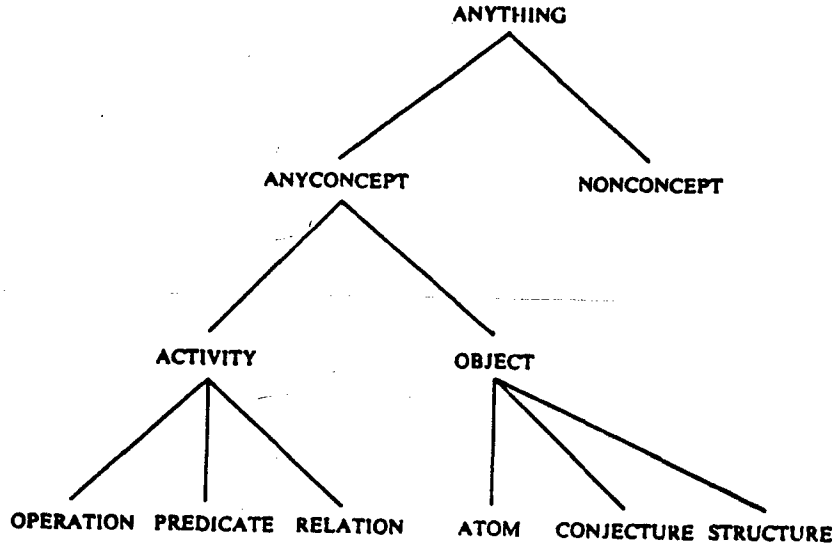


Figure D4c–2. AM's initial concept tree (partially shown).

AM began its investigations with scanty knowledge of a few set-theoretic concepts. Most of the obvious set-theoretical relations (e.g., de Morgan's laws) were eventually uncovered; since AM never fully understood abstract algebra, the statement and verification of each of these was quite obscure. AM never derived a formal notion of infinity, but it naively established conjectures like "A set can never be a member of itself" and procedures for making chains of new sets ("Insert a set into itself"). No sophisticated set theory (e.g., diagonalization) was ever done.

After this initial period of exploration, AM decided that "equality" was worth generalizing and thereby discovered the relation "same size as." Natural numbers were based on this discovery, and, soon after, most simple arithmetic operations were defined.

Since addition arose as an analogue to union, and multiplication as a repeated substitution, it came as quite a surprise when AM noticed that they were related (namely, $N + N = 2 \times N$). AM later rediscovered multiplication in three other ways: as repeated addition, as the numeric analogue of the Cartesian product of sets, and using the cardinality of the power set of the union of two sets.

Raising to fourth-powers and taking fourth-roots were discovered at this time. Perfect squares and perfect fourth-powers were isolated. Many other numeric operations and kinds of numbers were found to be of interest: odds, evens, doubling, halving, integer square root, and so on. Although it isolated the set of numbers that had no square roots, AM was never close to discovering rationals, let alone irrationals. No notion of "closure" was provided to—or discovered by—AM.

The associativity and commutativity of multiplication indicated to AM that it could accept a bag of numbers as its argument. When AM defined the inverse operation corresponding to "times," this property allowed the definition to be: "any bag of numbers greater than 1 whose product is x." This was just the notion of factoring a number x. Minimally factorable numbers turned out to be what we call primes. (Maximally factorable numbers were also thought to be interesting.)

Prime pairs were discovered in a bizarre way: by restricting the domain and range of addition to primes (i.e., solutions of $p + q = r$ in primes).

AM conjectured the fundamental theorem of arithmetic (unique factorization into primes) and Goldbach's conjecture (every even number greater than 2 is the sum of two primes) in a surprisingly symmetric way. The unary representation of numbers gave way to a representation as a bag of primes (based on unique factorization), but AM never came up with exponential notation. Since the key concepts of remainder, greater than, greatest common denominator, and exponentiation were never mastered, progress in number theory was arrested.

When a new base of *geometric* concepts was added, AM began finding some more general associations. In place of the strict definitions for the

equality of lines, angles, and triangles came new definitions of concepts comparable to parallel, equal measure, similar, congruent, translation, and rotation, together with many that have no common name (e.g., the relationship of two triangles sharing a common angle). A clever geometric interpretation of Goldbach's conjecture was found: Given all angles of a prime number of degrees ($0°, 1°, 2°, 3°, 5°, 7°, 11°, \ldots, 179°$), any angle between 0 and 180 degrees can be approximated (to within 1°) as the sum of two of those angles. Lacking a geometry "model" (an analogical representation like the one Gelernter, 1963, employed; see Article II.D3, in Vol. I), AM was doomed to propose many implausible geometric conjectures (see Article III.C5, in Vol. I).

Perhaps a full appreciation for the depth of AM's search of the concept space can be gained by examining Figure D4c-3, which shows the derivation path for prime numbers. It is eight levels deep and requires 14 concept-creation operations. This derivation is quite impressive, both because of its depth, and because the final concept is so far removed semantically from the initial concepts. Note, in particular, the fascinating way in which a new concept, SELF-COMPOSE, is used as a new operator to derive TIMES21 and TIMES22. AM is able to search in a highly directed, rational fashion.

Evaluating AM

It is important to ask how general the AM program is: Is the knowledge base "just right" (i.e., finely tuned to elicit this one chain of behaviors)? The answer is no: The whole point of this project was to show that a relatively small set of general heuristics can guide a nontrivial discovery process. Keeping the program general and not finely tuned was a key objective. Each activity or task was proposed by some heuristic rule (like "Look for extreme cases of X") that was used time and time again, in many situations. It was not considered fair to insert heuristics that provide guidance in only a single situation. For example, the same heuristics that lead AM to decompose numbers (using TIMES-inverse) and thereby discover unique factorization, also lead to decomposing numbers (using ADD-inverse) and the discovery of Goldbach's conjecture.

AM does, however, have some weaknesses. Although AM was able to discover and refine many interesting new concepts, it had no way of improving its stock of heuristic rules. Consequently, as AM ran longer and longer, the concepts it defined were further and further from the primitives it began with, and the efficacy of its fixed set of heuristics gradually declined. Lenat (1980) has proposed a solution to this problem. He advocates turning each heuristic rule into a concept and developing additional operators for creating new heuristics. The EURISKO project is presently pursuing this research.

A deeper problem has to do with some of the characteristics of the domain of mathematics that may not hold in other domains. One important fact about elementary mathematics is that the density of interesting concepts

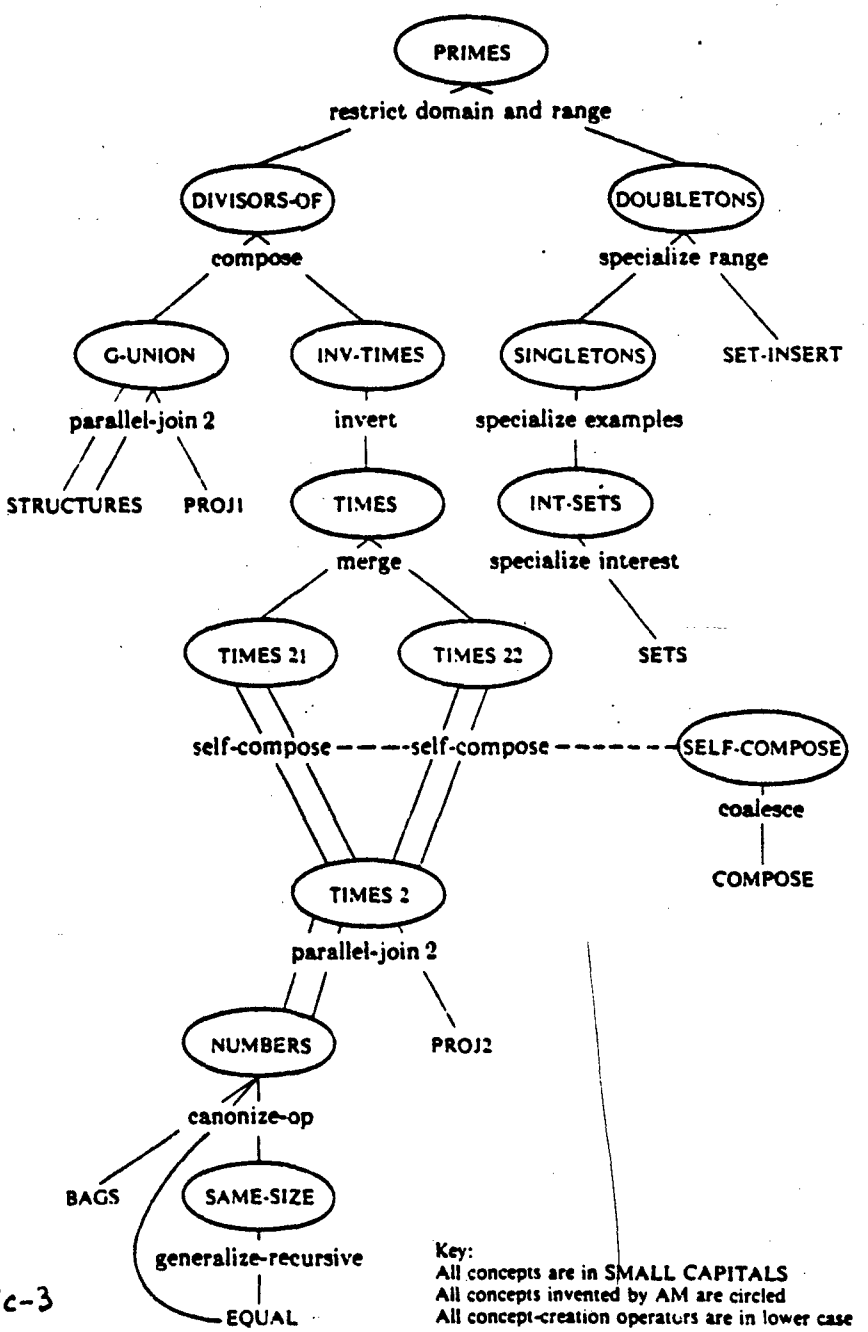

PRIMES
restrict domain and range
DIVISORS-OF
DOUBLETONS
compose
specialize range
G-UNION
INV-TIMES
SINGLETONS
SET-INSERT
parallel-join 2
invert
specialize examples
STRUCTURES
PROJ1
TIMES
INT-SETS
merge
specialize interest
TIMES 21
TIMES 22
SETS
self-compose
self-compose
SELF-COMPOSE
coalesce
COMPOSE
TIMES 2
parallel-join 2
NUMBERS
PROJ2
canonize-op
BAGS
SAME-SIZE
generalize-recursive
EQUAL
Key:
All concepts are in SMALL CAPITALS
All concepts invented by AM are circled
All concept-creation operators are in lower case

Figure 04c-3

is quite high. AM relies on the ability to build up complex concepts from more primitive concepts in a step-by-step fashion. At each step, the partial concepts must appear to AM to be interesting. In many domains, however, it is not possible to assess the interestingness of partial solutions. Consider, for example, the problem of credit assignment in a game such as chess. For a novice chess player, it is necessary to play an entire game before receiving any feedback on the quality of individual moves. Even as a player becomes expert, it is still necessary to search several moves in advance in order to evaluate a particular choice. Future efforts to develop AM-style discovery systems in other domains may face difficulties in evaluating the worth of concepts. More sophisticated interestingness heuristics may need to be developed. Work on the EURISKO project may provide some answers to these questions.

Conclusion

AM is a powerful discovery system that investigates and refines concepts in elementary set and number theory. It begins with a large body of knowledge about what kinds of concepts are mathematically interesting and how they can be synthesized from existing concepts. This knowledge can then carry AM far beyond its initial store of concepts to discover prime numbers and the fundamental theorem of arithmetic.

References

Lenat (1976) provides complete details on AM; see also Lenat (1977). Lenat (1980) describes the EURISKO project.

D5. Learning to Perform Multiple-step Tasks

MOST of the learning programs discussed so far in this chapter were designed to learn how to perform *single-step tasks*—that is, tasks in which one rule, or a set of independent rules, can be applied in one step to accomplish the performance task. In pattern classification (Article XIV.D2) and single-concept learning (Sec. XIV.D3), the performance element takes an unknown object or pattern and assigns it to one of two classes (e.g., an arch or a "nonarch"). These systems apply a single classification rule, or concept, to perform the classification. Even the sequence-extrapolation problems addressed by BACON (Article XIV.D3b) and SPARC (Article XIV.D3d) involve applying a single rule to predict the next item in the sequence from the previous items. Similarly, in the multiple-rule tasks of soybean-disease diagnosis (Article XIV.D4a) and mass-spectrometry simulation (Article XIV.D4b), several rules are applied in parallel to determine the unknown disease or to predict how the unknown molecule will break apart.

Multiple-step Tasks

In contrast, this section surveys a few learning systems that learn how to perform *multiple-step tasks*—that is, tasks in which several rules must be chained together into a sequence. Examples of multiple-step tasks include the game of checkers, in which rules for making individual moves must be chained together to play a whole game, and symbolic integration, in which several rules of integration must be applied sequentially to solve each integral. The goal of the learning system is to acquire a good set of rules for performing these tasks.

Multiple-step tasks are essentially *planning tasks* in which the performance element must find a sequence of operators to get from some starting state (e.g., the opening position in checkers) to some goal state (e.g., a won game). The chapters on search (Chap. II, in Vol. I) and planning (Chap. XV) describe various methods that have been used to accomplish this *state-space search* (see Article II.C3, in Vol. I). So far, AI learning systems have been developed only for simple, forward-chaining planning programs. No attempts have been made to learn how to perform hierarchical or constraint-based planning.

Viewing the Performance Element as a Production System

The first four systems described in this section—Samuel's (1959) checkers player, Waterman's (1970) poker player, Sussman's (1975) HACKER planning system, and Mitchell's LEX system for symbolic integration (Mitchell, Utgoff,

and Banerji, in press)—are all simple, forward-chaining problem solvers and, thus, can be viewed as simple production systems. The grammatical-inference systems discussed in the fifth article (Article XIV.D5e) employ context-free grammars, which can also be considered production systems. The knowledge base for each of these systems contains a set of production rules of the form:

$$\begin{aligned} \langle \text{situation}_1 \rangle &\Rightarrow \langle \text{action}_1 \rangle \\ \langle \text{situation}_2 \rangle &\Rightarrow \langle \text{action}_2 \rangle \\ &\vdots \\ \langle \text{situation}_n \rangle &\Rightarrow \langle \text{action}_n \rangle . \end{aligned}$$

The performance element repeatedly selects a rule whose situation part (left-hand side) matches the current state and applies the rule by performing the action indicated (right-hand side). The action usually has the effect of moving the performance element to a new state, closer to the goal.

For most of the programs discussed in this section, the possible actions are provided in advance. The problem addressed by the learning element is to determine under what situations the actions should be applied. This learning problem is similar in many ways to the problems addressed in Section XIV.D4 on learning multiple concepts.

However, two factors make this learning problem more difficult. First, because the rules must be chained together, the learning element has to consider possible interactions among the rules when it modifies the knowledge base. In LEX, for example, the learning element might decide that in any integral of the form

$$\int cf(x)\,dx,$$

the constant c should always be factored out. This is expressed in LEX as the production rule

If the integral has the form $\int cf(x)\,dx$, then apply OP03,

where OP03 converts $\int cf(x)\,dx$ to $c\int f(x)\,dx$. Unfortunately, if the constant c is 0 or 1, this is not an advisable step. Instead, OP08 (convert $1\cdot f(x)$ to $f(x)$) or OP15 (convert $0\cdot f(x)$ to 0) should be applied. When LEX is learning the production rule for OP03, it must take into account these possible interactions with OP08 and OP15. In fact, LEX's goal is to discover the best operator to apply in every situation. Thus, any time more than one operator is applicable because of overlapping left-hand sides, LEX must eliminate the overlap. In this case, the appropriate rule for OP03 is:

If the integral has the form $\int cf(x)\,dx \wedge c \neq 0 \wedge c \neq 1$, then apply OP03.

This is a particular instance of the general problem of incorporating new knowledge into the knowledge base (see Article XIV.A).

The second difficult aspect of multiple-step tasks is the problem of credit assignment. In single-step tasks, the system has available a *performance standard* that can be employed immediately after a rule is applied to determine whether or not the rule is correct. In disease diagnosis, for example, the learning element receives the correct disease classification along with each training instance. The performance element can apply its diagnosis rules and receive immediate feedback on the correctness of those rules. The performance standard can even be incorporated directly into the learning process as in the version-space method, in which the correct classification determines how the version space is updated.

In multiple-step tasks, however, feedback from the performance standard is not usually available until the game is completed or the problem is solved. The program can determine only whether the entire sequence of rules was good or bad. The credit-assignment problem is the problem of converting this overall performance standard into a performance standard for each rule. The overall credit or blame must be parceled out somehow among the individual rules that were applied.

The Importance of a Transparent Performance Element

To solve these problems of integration and credit assignment, it is critically important for the performance element to be transparent. A transparent performance element can provide the learning element with a trace of all actions that it *considered*, as well as those it actually performed. This allows the learning element to determine all of the rules that might have been applicable at each step of the problem-solving process. Such information makes it easier to solve the problem of integrating new rules into the knowledge base.

A complete performance trace also aids the credit-assignment task. During credit assignment, it is very useful to know why the performance element chose the rules that it did and what it expected those rules to do. By comparing the goals and expectations of the performance element with what really transpired, credit and blame can be assigned to individual decisions.

Extracting Local Training Instances from the Performance Trace

When the learning system for a multiple-step task is presented with a training instance—such as a board position in checkers and knowledge of which side can win from that position—it cannot immediately learn from the training instance. Instead, it must actually perform the task—that is, play out the checkers game—and compare the result with the information supplied by the performance standard—that is, which side should have won. During credit assignment, it can actually decide which individual decisions were good and which bad, and these evaluated decisions can serve as training instances for learning the left-hand sides of the production rules in the knowledge base.

By performing the task and assigning credit and blame, the "global" training instances can be converted into "local" training instances.

For example, in LEX, a global training instance consists of an integral such as

$$\int 2x^2\,dx$$

along with knowledge of whether or not the integral can be solved. The solution trace (see Fig. D5–1) shows that OP12 should not have been applied, since it leads to a complicated expression that requires several more steps to solve, but that OP03 and OP02 were used correctly.

Thus, three local training instances can be extracted:

$$\int 2x^2\,dx \;\Rightarrow\; \text{OP12 (negative)}.$$

$$\int 2x^2\,dx \;\Rightarrow\; \text{OP03 (positive)}.$$

$$2\int x^2\,dx \;\Rightarrow\; \text{OP02 (positive)}.$$

Once local training instances have been extracted, the techniques for doing concept learning discussed in Sections XIV.D3 and XIV.D4 can be applied to learn the left-hand sides of the production rules in the knowledge base. Figure D5–2 shows a slight perturbation of the simple learning-system model presented in Article XIV.A. The model now contains a loop in which the performance trace is analyzed by the learning element to extract local training instances. Global training instances are still supplied by the environment.

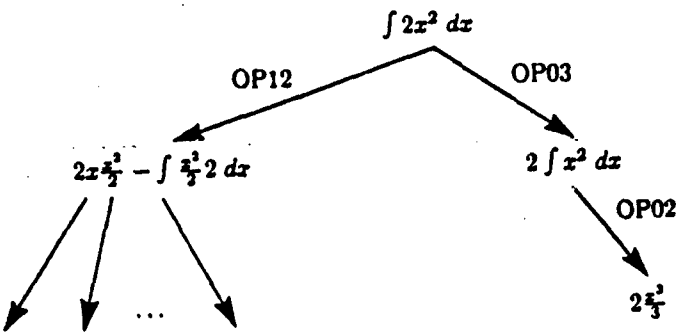


Figure D5–1. A sample performance trace.

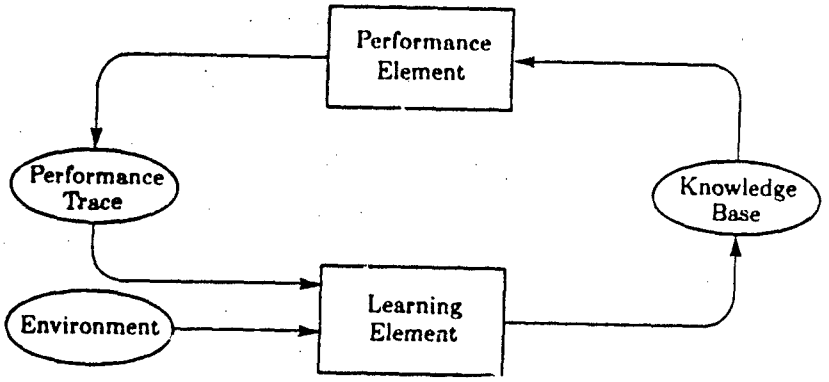


Figure D5-2. A modified model of learning systems.

Outline of This Section

The five systems presented in this section all perform multiple-step tasks and, consequently, must address problems of integrating new rules and assigning credit and blame. Waterman, and to some extent Samuel, simplifies the credit-assignment problem by obtaining a move-by-move performance standard from the environment. Furthermore, all of the systems, except Waterman's poker system, ignore the problem of integrating new rules into the knowledge base. Work in this area is still in its infancy, and more sophisticated learning systems for multiple-step tasks can be expected in the future.

References

Buchanan, Mitchell, Smith, and Johnson (1977) provide another perspective on the use of feedback in learning systems.

D5a. Samuel's Checkers Player

FROM 1947 to 1967, Arthur Samuel conducted a continuing research project aimed at developing a checkers-playing program that was able to learn from experience. Samuel investigated three different representations for checkers knowledge—memorized moves, polynomial evaluation functions, and signature tables—and two different training methods—self-play and book-move learning. The work on rote learning of checkers moves is discussed in Article XIV.B2. The present article discusses two specific learning situations: (a) self-play as it was used to learn a polynomial evaluation function and (b) book-move training as it was used to learn a set of signature tables. Samuel experimented with several other combinations of training methods and representations (for more details, see Samuel, 1959, 1967).

The performance element in all of Samuel's systems employs a look-ahead, game-tree search to determine which moves to make (see Articles II.B3 and II.C5, in Vol. I). The performance element uses a *static evaluation function* (Article II.C5) to evaluate possible future positions in the game and applies alpha-beta minimaxing to determine the best move to make. The goal of the learning process is to establish and improve this static evaluation function through experience.

Learning a Polynomial Evaluation Function Through Self-play

The first static evaluation function investigated by Samuel was a polynomial of the form

$$\text{value} = \sum_i w_i f_i ,$$

where f_i are board features and w_i are real-valued weights (coefficients). For most of Samuel's experiments, a polynomial with 16 features was employed. Each board feature provides a numerical measure of some aspect of the board position under evaluation. For example, the EXCH feature measures the *relative exchange advantage* of the player whose turn it is to move. EXCH is computed by taking $T_{current}$, the total number of squares into which the player to move may advance a piece, and in so doing force an exchange, and subtracting $T_{previous}$, the corresponding quantity for the previous move by the opposing player.

Samuel's program faced two tasks in attempting to learn such a polynomial evaluation function: (a) discovering which features to use in the function and (b) developing appropriate weights for combining the various features to obtain a value for the board position. We describe the weight-learning task first and later return to the problem of discovering which features to use.

In the self-play mode of training, the checkers program learns by playing a copy of itself. The version of the program that is doing the learning is referred to as Alpha, while the copy that serves as an opponent is called Beta. The learning procedure employed by Alpha is to compare at each turn its estimate of the value for the current board position with a performance standard that provides a more accurate estimate of that value. The difference between these two estimates controls the adjustment of the weights in the evaluation function. Alpha's estimate is developed by conducting a shallow minimax search applying the evaluation polynomial to tip board positions and backing up these values (see Article II.C5a, in Vol. I). The performance standard is obtained by conducting a deeper minimax search into future board positions using the same evaluation function as in the shallow search. Samuel takes advantage of the fact that a deep search is usually more accurate than a shallow one.

How does Alpha use this move-by-move performance standard to guide its search for proper weighting coefficients? First, the difference, Δ, between the performance standard and Alpha's estimate is computed. If Δ is negative, Alpha's polynomial is overestimating the value of the position. If Δ is positive, Alpha is underestimating it. For each board feature, a count is kept of the times that the sign of that feature agrees or disagrees with the sign of Δ. From these tallies, a correlation coefficient is developed that indicates the degree to which that feature predicts Δ. The goal of the learning procedure is to minimize Δ (so that Alpha is duplicating the evaluations of the performance standard). The weights of the polynomial are determined by scaling the correlation coefficients onto the range -2^{18} to 2^{18}. Large positive coefficients are given to features that strongly predict positive values of Δ and vice versa, so that the polynomial will tend to "follow" Δ and thus reduce it.

The overall effect of this scheme is to independently assign blame for Alpha's estimation errors to the individual features. This is sensible, since the features are combined independently (i.e., by addition, without any interaction terms) to form the polynomial.

Alpha can be viewed as conducting a hill-climbing search through the "rule space"—the space of possible weights. Each move in the checkers game serves as a training instance to guide this search. The correlation coefficients summarize the entire body of training instances and indicate in which direction the search must move in order to minimize Δ.

Hill-climbing is known to have many drawbacks, including convergence to local maxima. Samuel addresses this problem as follows. When Alpha and Beta commence play, they are identical. However, while Alpha proceeds to search the rule space, Beta does not change. As Alpha improves, it begins to defeat Beta regularly. When Alpha has won a majority of the games played, Beta adopts Alpha's improved evaluation function, and the count of games won and lost is started again from zero. Beta is thus used to "remember" a good point in the rule space. If Alpha is at a local maximum, however, its

performance will tend to worsen whenever it makes a minor modification to its polynomial. To prevent a local maximum from halting Alpha's improvement, an arbitrary change is made to Alpha's scoring polynomial whenever Alpha loses three games to Beta. The largest weight in Alpha's polynomial is set at zero to jump Alpha to some new point in the rule space.

Now that we have seen how Samuel's program determines the weights for the evaluation polynomial, we turn our attention to the first learning problem—determining what features should be used to evaluate a board position. This is a variant of the *problem of new terms* (see Article XIV.D1): How can a learning program discover the appropriate terms for representing its acquired knowledge? Samuel offers a partial solution to this problem, namely, *term selection*. The learning program is provided with a list of 38 possible terms. Its learning task is to select a subset of 16 of these terms to include in the evaluation polynomial.

The selection process is quite straightforward. The program starts with a random sample of 16 features. For each feature in the polynomial, a count is kept of how many times that feature has had the lowest weight (i.e., the weight nearest zero). This count is incremented after each move by Alpha. When the count for some feature exceeds 32, that feature is removed from the polynomial and replaced by a new term. At all times, 16 features are included in the polynomial, and the remaining 22 features form a reserve queue. New features are selected from the top of the queue, while features removed from the polynomial are placed at the end of the queue. Viewed in the context of credit assignment, Samuel's program assigns blame to features whose weights have values near zero, since those features are making no contribution to the evaluation function.

Samuel (1959) was dissatisfied with this term-selection approach to the new-term problem. He writes:

> It might be argued that this procedure of having the program select new terms for the evaluation polynomial from a supplied list is much too simple and that the program should generate terms for itself. Unfortunately, no satisfactory scheme for doing this has yet been devised. (p. 220)

The feature-selection and weight-adjustment learning processes take place concurrently. In Samuel's experiment with these learning methods, the set of selected features and their weights started to stabilize after roughly 32 games of self-play. The resulting program was able to play a "better-than-average" game of checkers (Samuel, 1959, p. 222).

Learning a Signature Table by Book Training

The second kind of static evaluation function investigated by Samuel was a system of *signature tables*. A signature table is an n-dimensional array. Each dimension of the array corresponds to one of the measured board features.

To obtain the estimated value of a board position, we measure each of the board features and index these values into the signature-table array. The contents of each cell in the table is a number that gives the value of the corresponding board position. In a sense, the signature table maps all possible board positions into a small n-dimensional feature space. Every point in that feature space is represented as a cell in the signature table that gives the value of all board positions mapped to that point.

Suppose, for example, that we had only three features: KCENT (king center control), MOB (total mobility), and GUARD (back-row control). The cube shown in Figure D5a–1 is a schematic diagram of the resulting signature table. Notice that KCENT and GUARD take on only the values -1, 0, and 1, while MOB is allowed to take on values from -2 to $+2$. If we have a board position for which KCENT $= 1$, GUARD $= 0$, and MOB $= 2$, then we look into the signature table at the cell addressed by $(1, 0, 2)$ to obtain the value: .8.

It is possible to view this signature table as a set of $3 \times 3 \times 5 = 45$ production rules. There is one rule for every possible combination of features—every cell—in the table. The rule for the situation illustrated in Figure D5a–1 could be stated as

If: KCENT $= 1 \wedge$ GUARD $= 0 \wedge$ MOB $= 2$,
Then: Value of position $= .8$.

Signature tables are more expressive than linear polynomials because they can capture interactions among all of the features. Their main drawbacks, however, are their large size and related problems with learnability. A full signature table for the entire set of 24 terms used by Samuel would contain roughly 6×10^{12} cells—far too large to be stored or effectively learned. Two techniques were applied to overcome these problems. First, the number of possible values for each feature was substantially reduced. Most features were restricted to three values: $+1$ (if the position is good for the program), 0 (if the position is even), and -1 (if the position is bad for the program). Second,

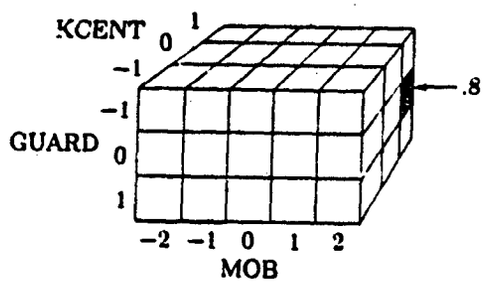


Figure D5a–1. A three-dimensional signature table.

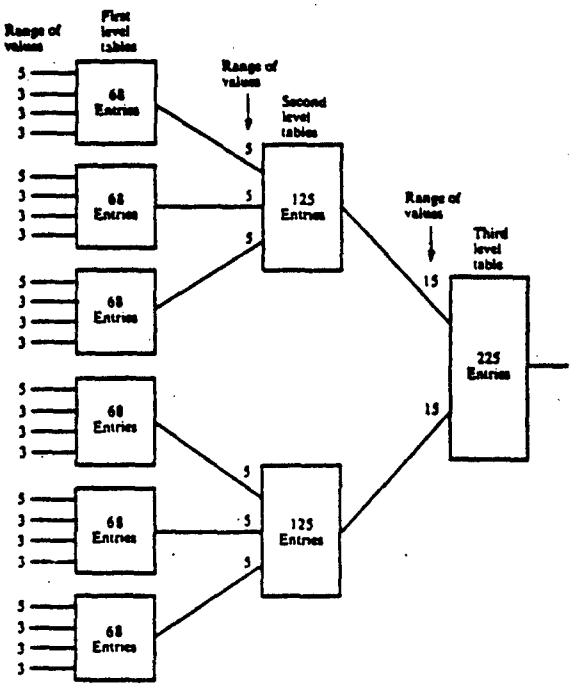


Figure D5a–2. Three-level hierarchy of signature tables (from Samuel, 1967).

instead of one giant signature table, Samuel adopted the three-level hierarchy shown in Figure D5a–2.

The 24 board features are partitioned into six important subgroups, and a separate signature table is developed for each group. The outputs of the six first-level signature tables are values between -2 and $+2$ that are used as indexes to two second-level signature tables. The second-level tables produce values between -7 and $+7$ that are used as indexes to the final signature table to obtain the estimated value of the board position. This hierarchical system was found to be expressive enough to support excellent checkers play and small enough to be learnable.

The program learns the values for the cells in these tables by following "book games" played between two master checkers-players. Approximately 250,000 board situations of master play were presented to the program. Most of these moves were selected from games ending in a draw. The program operates as follows. Each cell in the signature table is associated with two counts, called A (agree) and D (differ). Initially, A and D are zero for each cell. At each move, the program is faced with a set of alternative moves, one

of which is the book-designated move. Each of these possible moves can be mapped into one cell in *each* signature table. The program adds a one to the D count of each cell whose corresponding move was *not* the book-preferred move. A total of n (where n is the number of nonbook moves) is added to the A count of each cell corresponding to the book-preferred move. Periodically, the contents of the signature-table cells themselves are updated to reflect the A and D counts. Each cell is given the value

$$C = \frac{(A - D)}{(A + D)},$$

which is a rough correlation coefficient indicating the extent to which the board positions mapped to that cell are the book-preferred moves. The correlation coefficients are then scaled into the -2 to $+2$ (or -7 to $+7$) range.

This learning process can be viewed as a technique of learning from examples. Each move provides a training instance that is used to update several signature-table entries. Credit assignment is easy, because the book provides a fairly reliable performance standard on a move-by-move basis. Credit is assigned to the signature-table cell corresponding to the book move, and blame is allotted to all cells corresponding to rejected alternative moves. It is the learning-by-doing approach that allows the program to determine which moves are the alternative moves.

The second- and third-level tables are trained at the same time, and by the same techniques, as the first-level tables. The current contents of the signature tables are used to determine which second- and third-level cells correspond to the alternative moves under consideration, and their A and D totals are updated during each move. The learning process is quite erratic at the start, since most of the first-level signature-table cells contain zeros initially. Thus, incorrect second- and third-level cells are selected during the early stages of learning. As learning progresses, these errors are overcome.

To make the tables more reliable during the early stages of training, some smoothing is done to fill in cells for which the A and D counts are still near zero. Smoothing is a form of generalization involving interpolating and extrapolating from surrounding cells in the table. The smoothing has no effect on the A and D counts—these are used later to replace the interpolated values with more accurate, induced values.

One other refinement of the signature-table system is to break the game of checkers into seven chronological *phases* and to use a different signature table for each phase. Samuel reasoned that the board features relevant to determining good moves during the opening of the game are unlikely to be the same as those used during the ends of games. The seven-phase approach leads to an increase in the number of cells, thus making the tables more difficult to learn. However, Samuel was able to fill in empty cells by smoothing from the tables of adjacent phases.

Results

Samuel's signature-table system was much more effective as a checkers player than any of the other configurations he tested. To assess the goodness of play, Samuel tested the program on 895 book moves that were not used during the training. A count was made of the number of times that the program rated 0, 1, 2, etc., moves as equal to or better than the book-recommended move. After training on 173,989 book moves, the test gave the results shown in Table D5a–1. By summing the first two columns, we see that the program chooses the best move or the second-best move, as defined by the book, 64% of the time. These ratings are made without employing any forward search. Minimax look-ahead search improves the performance of the program substantially.

Despite this impressive level of performance, champion checkers players are still able to beat the program. In 1965, the world champion, W. F. Hellman won all four correspondence games played against the program. He drew with the program during one "hurriedly played cross-board game" (Samuel, 1967, p. 601, n. 2).

Comparison of the Signature-table and Polynomial Methods

The signature-table method substantially outperformed the polynomial-evaluation-function approach. Even when both methods were trained by following book moves, the moves chosen by the polynomial evaluation function correlated with the book-indicated moves only half as well as the moves chosen by the signature tables. This difference is due to the improved representational power of the signature tables. The signature table can represent nonlinear relationships among the various terms, since there is a different table cell for each possible combination of terms. In the polynomial representation, only linear relationships are possible. Such a representation assumes that each term contributes independently to the value of a board position. This assumption is evidently incorrect for checkers.

Conclusion

Samuel developed and tested several different representations and training techniques for teaching a program to play checkers. Among the contributions

TABLE D5a–1
Evaluation of Signature-table Performance

Number of moves rated as better than or equal to book move	0	1	2	3	4	5	6
Relative proportion	38%	26%	16%	10%	6%	3%	1%

of this work are (a) the demonstration that machine-learning techniques can be highly successful, (b) the technique of using a deeper search and book-supplied moves to solve the credit-assignment problem, (c) the term-selection methods for determining which features to include in the polynomial evaluation function, and (d) the demonstration that signature tables provide a much more effective representation for checkers knowledge than either the linear-polynomial or the rote-learning techniques.

References

All of this work is discussed in Samuel (1959, 1967). See Buchanan, Mitchell, Smith, and Johnson (1977) for a discussion of Samuel's term-selection technique as an instance of a *layered* learning system.

D5b. Waterman's Poker Player

AS PART of his thesis project, Donald Waterman (1968) developed a computer program that learns to play draw poker. Draw poker is a game of imperfect information in which psychological factors, such as how easily one's opponent is bluffed, become important. Minimax look-ahead search is not possible because the overall state of the game (i.e., the contents of all the hands) is not completely known. Instead, approximate heuristic methods must be used. Waterman developed a production system (see Article III.C4, in Vol. I) to encode a set of heuristics for poker, and he sought to have his program discover these production rules through experience. In this article, we first describe Waterman's production-rule knowledge representation and its application in the poker-playing performance element; we then discuss in detail the methods used in the learning element to acquire and refine these production rules.

Waterman's Performance Element for Draw Poker

Each game of draw poker is divided into five stages. First, each player is dealt five cards. This is followed by a betting stage in which the players alternately choose to place a bet larger than the opponent's bet (RAISE), place a bet equal to the opponent's bet (CALL), or give up (DROP) the hand; a CALL or DROP action ends this stage. In the third stage, each player has the option of replacing up to three of his (or her) cards with new cards drawn from the deck. This is followed by another betting stage like the first. Finally, the hands are compared (except in a DROP), and the player with the best hand wins the game.

Waterman's performance element has built-in routines for carrying out the deal, the draw, and the final comparison of hands. The two betting stages, however, are performed by a modifiable production system. It is the production rules making up this production system that the program attempts to learn and improve.

The production system developed by Waterman contains two basic kinds of rules: *interpretation rules* that compute important features of the game situation and *action rules* that decide which action (CALL, DROP, or RAISE) to take.

The action rules make their decisions based on the values of seven key variables that make up the so-called *dynamic state vector:*

(VDHAND, POT, LASTBET, BLUFFO, POTBET, ORP, OSTYLE).

VDHAND, for example, is a measure of the value of the program's hand, POT is the current amount of money in the pot, and BLUFFO is an estimate of the opponent's "bluffability."

The interpretation rules compute the values of these seven variables from directly observable quantities. To compute the value of BLUFFO, for example, features such as OBLUFFS (the number of times the opponent has been caught bluffing) and OCORREL (the correlation between the opponent's hands and his bets) are examined. Once numeric values for the seven variables have been computed, they are converted into symbolic values that describe important subranges of values. For example, the rule

```
If POT > 50, then POT = BIGPOT.
```

gives POT the symbolic value BIGPOT whenever POT is larger than 50.

The action rules are stated solely in terms of these symbolic values. A typical action rule is

```
(SUREWIN, BIGPOT, POSITIVEBET, *, *, *, *)
⇒  (*, POT + (2 × LASTBET), 0, *, *, *, *) CALL,
```

which can be paraphrased as

```
If:      VDHAND = SUREWIN
and         POT = BIGPOT
and     LASTBET = POSITIVEBET,
Then:       POT := POT + (2 × LASTBET)
        LASTBET := 0
                   CALL.
```

The condition and action parts of the rule have the same form as the state vector. The left-hand side of the rule is a pattern that is matched against the state vector to determine whether the rule should be executed. The right-hand side of the rule indicates which action to take and provides instructions for modifying the value of the state vector.

These production rules are applied by the performance element as follows. First, all of the interpretation rules are used to analyze the current game situation in order to develop the dynamic state vector. Next, the action rules are examined one by one in a *fixed order* until a rule is found whose condition pattern matches the state vector. That rule is executed to make the program's move. This fixed ordering for the production rules serves as a conflict-resolution technique (see Article III.C4, in Vol. I). If more than one rule is applicable in a given situation, only the first rule in the list is executed. Hence, when new rules are acquired or old rules are modified, the order of the rules must be carefully considered.

There are two basic ways to generalize the left-hand side of an action rule. One method is to drop a condition by replacing one of the symbolic values on the left-hand side (e.g., BIGPOT) by *, which matches *any* value. The other method is to modify the interpretation rule that defines a symbolic value so that it includes a larger set of underlying numeric values (e.g., changing BIGPOT

to be any POT > 40). This is the same as Michalski's method of generalizing by internal disjunction (see Article XIV.D1). We will see below how Waterman makes use of these two generalization methods.

Learning to Play Poker

Waterman sought to have the program learn the interpretation rules, the action rules, and the ordering of the action rules by playing poker games against an expert opponent. As the poker games proceed, the learning element analyzes each of the decisions of the performance element and extracts training instances. Each training instance is in the form of a *training rule*, that is, a specific production rule that would have made the correct decision had it been chosen and executed. The training rules guide the learning element as it determines which production rules to generalize and specialize.

The task of extracting a training rule is quite difficult, because the environment provides very little information that could serve as a performance standard. Unlike deterministic games such as checkers or chess that have no chance element, poker is probabilistic. Even an expert player will lose from time to time. Thus, the program must play several hands before it can assess the quality of the production rules in its knowledge base. As discussed in the introduction to this section (Article XIV.D5), however, even when a reliable performance standard is available on a full-game basis, the problem of assigning credit or blame to individual moves in that game is still very difficult. Consequently, Waterman sought to provide the program with some form of move-by-move performance standard. Three different techniques were developed: advice-taking, automatic training, and analytic training.

In advice-taking, the program plays a series of poker games against a human expert. After each turn by the performance element, the learning element asks the expert whether the performance-element action is correct. The expert responds either with (OK) or with some advice such as (CALL BECAUSE YOUR HAND IS FAIR, THE POT IS LARGE, AND THE LASTBET IS LARGE). This advice provides the training rule directly.

In the automatic-training approach, an expert program serves as the opponent and advice-giver. The expert program uses a knowledge base of production rules developed by Waterman himself to determine, at each move, what action to take. During play against the learning program, the expert program compares each move made by the learning program with the move it would have made and provides advice exactly as a human expert would.

Finally, the most interesting method of instruction, the analytic method, involves no advice-taking whatsoever. After each full round of play (i.e., each single hand), the learning element analyzes the moves made by the performance element and attempts to deduce which moves were incorrect. In place of an externally supplied performance standard, the learning element is provided with a predicate-calculus axiomatization of the rules of poker. From

these axioms, the program is able to deduce, after the hand is over, what the correct decisions would have been, thus providing the learning element with a performance standard.

Once the learning element has a move-by-move performance standard, it can extract a training rule and modify the production system. The modification process works by first locating the production rule that made the incorrect decision and then examining the list of production rules for a rule before or after the error-causing rule that could have made the correct decision. If such a rule is found, generalization and specialization techniques are applied to modify the production rules so that the proper rule would have been executed. If no such rule is found, the training rule itself is inserted into the production-rule list immediately in front of the error-causing rule.

In the remainder of this article, we discuss how each of these three training techniques allows the learning element to develop a training rule. For the advice-taking and automatic-training methods, this is straightforward. In the analytic approach, however, a series of credit-assignment problems must be solved. We describe Waterman's solutions in detail. Finally, we describe how the training rule acquired by any one of these methods is used to modify the current set of production rules in the knowledge base.

Advice-taking and Automatic Training

In the advice-taking and automatic-training methods, the program is supplied after each move with advice such as:

```
(CALL BECAUSE YOUR HAND IS FAIR, THE POT IS LARGE,
AND THE LASTBET IS LARGE).
```

This advice provides the training rule directly. The proper action (i.e., the right-hand side of the training rule), CALL, is indicated along with the relevant variables and their values. This advice is equivalent to the production rule:

```
(FAIR, LARGE, LARGE, *, *, *, *)
⇒ (*, POT ← (2 × LASTBET), 0, *, *, *, *) CALL.
```

The details of the right-hand side of the rule can be filled in automatically for each action from knowledge of the rules of the game. In this case, for example, CALL requires the program to match its opponent's bet, and thus the POT must increase by twice LASTBET, once for the opponent's bet and again for the program's reply. The other possibilities, DROP and RAISE, are handled similarly.

It is interesting to note that Waterman's program accepts fairly low-level advice. The expert's advice can easily be interpreted in terms of the present game situation, so there is no need to interpret or operationalize the advice (see Article XIV.C1). Waterman's advice-taking research concentrates, instead,

on the problem of integrating this advice into the current knowledge base. We describe how this happens after we discuss the methods employed during analytic training to obtain the training rule.

Learning by the Analytic Technique

The main difficulty facing Waterman's program during analytic training is credit assignment. The learning element has to deal with a pair of credit-assignment problems. The first problem is *determining the quality of a round of play*. As we mentioned above, the probabilistic nature of draw poker makes this difficult, since the loss of a single hand does not necessarily indicate that the program is playing poorly. Furthermore, the fact that poker is a game of imperfect information leads to difficulties. If, for example, the program "drops" its bid (i.e., folds its hand and gives in to the other player), the contents of the opponent's hand are never known. The program solves this first credit-assignment problem by always "calling" the bid (i.e., meeting the opponent's bet and requesting to see his hand), instead of dropping, and by applying its knowledge of the rules of poker to deduce whether the program could have improved its play within the round.

If the program could have done better, it turns its attention to the second credit-assignment problem—*determining which individual moves were poor.* During the round of play, a complete trace of the actions of the performance element is kept. To solve the second credit-assignment problem, the learning element applies its axiomatization of the rules of poker to evaluate each move in detail. The rules of poker are axiomatized in predicate calculus as a set of implications such as:

$$\texttt{ACTION(CALL)} \wedge \texttt{HIGHER(YOURHAND, OPPHAND)}$$
$$\supset \texttt{ADD(LASTBET, POT)} \wedge \texttt{ADD(POT, YOURSCORE)}.$$

These statements define the effects of each of four possible actions: `BET HIGH`, `BET LOW`, `CALL`, and `DROP`. To evaluate a particular move in the game, the learning element takes the value of the dynamic state vector at that point and uses it to determine the truth value of certain predicates in this axiom system (e.g., `GOOD(OPPHAND)`, `HIGHER(OPPHAND, YOURHAND)`). Then it tries to prove the statement

`MAXIMIZE(YOURSCORE)`

by backward-chaining through the axiom system (see Article III.C4, in Vol. I). The resulting proof indicates the action that should have been performed and provides the move-by-move performance standard. When the performance standard differs from the move made by the program, blame is assigned to that move, and the learning element builds a training rule.

The correct decision, obtained from the performance standard, forms the right-hand side (action part) of the training rule. Waterman axiomatized the

RAISE action as two possible subactions, BET HIGH and BET LOW, so that the program would not have to learn how big a bet to make. For BET HIGH, the performance element chooses a random bet between 10 and 20. Similarly, a BET LOW action leads to a random bet between 1 and 9. Thus, the performance standard provides the complete right-hand side of the training rule.

The left-hand side of the training rule is obtained by examining a table called the *decision matrix*. The decision matrix contains four abstract rules, one for each possible action. These rules tell which values of the seven state variables are relevant for the indicated action. The exact values of the variables are not given—only a general indication of whether the values should be large or small. For instance, the abstract rule for the DROP action is

(CURRENT, LARGE, LARGE, SMALL, SMALL, CURRENT, LARGE) ⇒ DROP,

or more clearly,

If:	VDHAND = (current symbolic value of VDHAND)
and	POT = LARGE
and	LASTBET = LARGE
and	BLUFFO = SMALL
and	POTBET = SMALL
and	ORP = (current symbolic value of ORP)
and	OSTYLE = LARGE,
Then:	DROP.

Once the learning element has deduced from the axioms that the proper action would have been DROP, it takes the corresponding rule from the decision matrix and uses it as the training rule. Notice that the level of abstraction of the rules in the decision matrix is the same as the level of abstraction of the advice supplied by the human expert or expert program.

It could be argued that the use of the decision matrix is improper, since it provides the learning element with essential information that a person who was learning to play poker would have to discover himself. Waterman (1968) suggests some methods by which the decision matrix could be learned from experience, but none of these was implemented.

Using the Training Rule to Modify the Knowledge Base

Once the training rule is obtained, whether by advice from a person, by advice from the expert program, or by analysis, it must be used to modify the production rules in the knowledge base. The training rule is first used to modify the interpretation rules. The left-hand side of the training rule is compared with the state vector computed by the interpretation rules. LARGE matches symbolic values that correspond to large values of the underlying variable. Similarly, SMALL matches small values. If a symbol does not match,

the interpretation rules that computed that symbol are assigned blame. They are then either modified or augmented to include a new interpretation rule.

Suppose, for example, that the state vector listed POT as having the value P3, where P3 is derived by the interpretation rule:

```
If POT > 20, then POT = P3.
```

Furthermore, suppose that the value of POT in the game situation being analyzed is 45. By comparing P3 with LARGE, the learning element determines that this interpretation rule is incorrect (since P3 can refer to very small values of POT). The learning element can either modify the rule (by substituting 44 for 20) or create a new rule. A user-supplied parameter, KK, specifies the largest allowable change that can be made to a numeric value in an interpretation rule. In this case, we will assume that the learning element creates the new rule

```
If POT > 44, then POT = P4.
```

and modifies the state vector so that POT has the value P4.

Once the interpretation rules have been checked and modified, the updated state vector is matched against the action rules to find the rule that made the incorrect decision. This rule is called the *error-causing rule*. The training rule is then used to locate a production rule that could have made the correct decision had it been executed. This is accomplished by comparing the right-hand side of the training rule with each production rule in the rule base.

Waterman's program classifies action rules as either *recently hypothesized* or *accepted*. A recently hypothesized rule is one that was recently added to the knowledge base, whereas an accepted rule is one that the program believes to be nearly correct. The learning element follows a strategy of first attempting to make minor changes in accepted rules and then, if minor changes do not suffice, attempting to make major changes in recently hypothesized rules. Finally, if a suitable recently hypothesized rule cannot be found, the training rule is added to the rule base and is labeled as recently hypothesized.

The learning element searches upward ahead of the error-causing rule for an accepted rule that would have made the correct decision. If such a rule is found, it is checked to see if the pattern of its left-hand side can be generalized to match the current state vector. Only minor generalizations—that is, changes to the interpretation rules—are considered. No conditions are dropped (i.e., replaced by •).

If no accepted rule can be found, the learning element again searches upward before the error-causing rule, this time looking for a recently hypothesized rule that would have made the correct decision. If such a rule is found, major changes—including both dropping conditions and modifying interpretation rules—are made in the left-hand-side pattern so that it matches the state vector.

If no suitable rules can be found before the error-causing rule, the learning element searches for an accepted rule *after* the error-causing rule. If an appropriate rule is found there, the error-causing rule and all intervening rules must be specialized so that they will *not* match the state vector, and the target rule must be generalized—by changing the interpretation rules—so that it *will* match the state vector.

Finally, if no rules can be found that could be generalized to make the correct decision, the training rule is inserted into the ordered list of production rules immediately in front of the error-causing rule. The training rule is marked as being recently hypothesized. Figure D5b–1 depicts this four-step process of modifying the rule base.

This four-step process combines the task of integrating new knowledge into the knowledge base with the task of generalizing the training rule. Notice that the integration process must have knowledge about how the performance element chooses which rule to execute, so that it can decide how to update the rule base. The generalization process is fairly ad hoc. For example, recently hypothesized rules become accepted when enough conditions are dropped from the left-hand side so that only N conditions remain (N is a parameter given to the program). This is a very weak technique for preventing rules from becoming overgeneralized.

Results

Waterman's poker program learned to play a fairly good game of poker. Separate tests were conducted with each of the three training techniques. In each case, the program started with only one rule: "In all situations, make a random decision." For advice-taking from a human expert and for learning

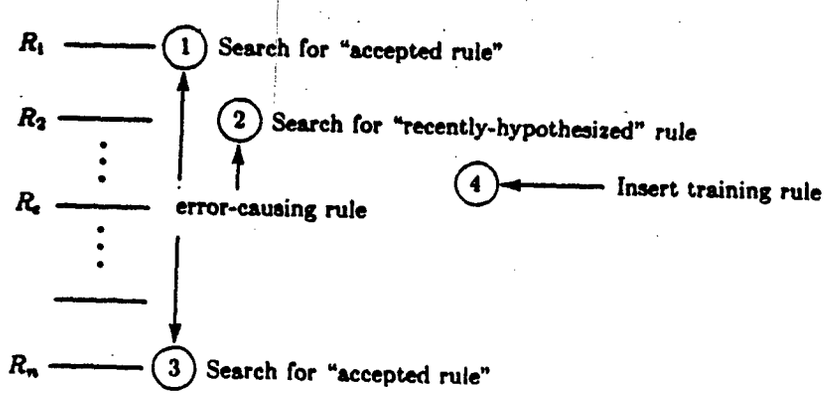


Figure D5b–1. The four steps to modifying the production-rule base.

from the expert program, training was continued until the program played one complete game of five hands without once making an incorrect decision (as judged by the expert). For the analytic method, the program continued to play games until the original "random decision" production rule was executed only 5% of the time. The results are shown in Table D5b–1.

The rightmost column shows the results of a proficiency test in which the program and a human expert played two sets of 25 hands. During the first set of 25 hands, the cards were drawn at random from a shuffled deck as in ordinary play. However, during the second set of 25 hands, the same hands were used as in the first set, except that the program received the hands originally dealt to the person and vice versa. At the end, the cumulative winnings of the program and person were compared.

The results show that in all three training methods, performance improved markedly. The automatic training provided the best performance improvement, perhaps because the automated expert played more consistently than the human expert. Although the analytic method performed the poorest, the results are not strictly comparable, since the axiom set provided it with only four possible actions, whereas the advice-based methods were given eight possible actions. Consequently, the analytic method may not actually be inferior to the two advice-taking methods.

Conclusion

Waterman's poker-playing program faces a very difficult learning problem. Poker is a multiple-step task that provides very little feedback to the learning program. For the two advice-taking methods, this problem is sidestepped by allowing the program to accept a training rule directly from an expert. However, for the analytic method, two credit-assignment problems must be solved: evaluating a round of play and evaluating a particular move. To solve these problems, the program modifies its betting strategy (to call instead

TABLE D5b–1
Comparison of Three Training Methods (from Waterman, 1970)

Training method	Number of training trials	Final number of rules	Percent difference in winnings[a]
Before training	0	1	−71.0
Advice-taking	38	26	−6.8
Automatic training	20	19	−1.9
Analytic method	57	14	−13.0

[a]These percentages are computed by subtracting the amount of money won by the opponent from the amount of money won by the program and dividing by the amount of money won by the opponent. In all cases, the program won less than the opponent and, hence, the percentages are all negative.

of dropping) and applies knowledge available from the axiom set and from the decision matrix. This permits the credit-assignment process to extract a training rule from the trace of decisions taken by the performance element. Once the training rule is acquired by any of these three methods, it is used to guide the generalization and specialization of the production rules in the knowledge base. Since only positive training instances are available, the program must make use of arbitrary constraints to prevent overgeneralization.

References

Waterman (1970) describes this work in detail.

D5c. HACKER

HACKER is a learning system developed by Gerald Sussman (1975) to model the process of acquiring programming skills. HACKER's performance task is to plan the actions of a hypothetical one-armed robot that manipulates stacks of toy blocks. This planning task is described in detail in Article XV.C.

HACKER learns by doing. It develops plans and simulates their execution. The plan and the trace of the execution are examined by HACKER to acquire two kinds of knowledge: *generalized subroutines* and *generalized bugs*. A generalized subroutine is similar to a STRIPS macro operator (see Article II.D5, in Vol. I), in that it provides a sequence of actions for achieving a general goal. A generalized bug is a *demon* that inspects new plans to see if they contain an instance of the bug and provides an appropriate bug fix.

An example of a generalized subroutine is the following procedure for stacking one block on top of another:

```
(TO (MAKE (ON a b))
    (EPROG
       (UNTIL (y) (CANNOT (ASSIGN (y) (ON y a)))
                  (MAKE (NOT (ON y a)))
       (PUTON a b))).
```

The goal of this procedure is (MAKE (ON a b)): The procedure changes the world so that (ON a b) is true. This subroutine is general and works for any two blocks a and b (a and b are variables that are bound to particular blocks—denoted by capital letters—when the subroutine is invoked). The procedure removes everything that is on a and then picks up a and puts it on b.

Viewed as a production rule, this procedure could be written as:

```
(MAKE (ON a b))  ⇒  (EPROG
                       (UNTIL (y) (CANNOT (ASSIGN (y) (ON y a)))
                                  (MAKE (NOT (ON y a)))
                       (PUTON a b)).
```

From this perspective, we see that when HACKER learns a generalized subroutine, it is learning both a generalized left-hand side, the goal, and a generalized right-hand side, the plan. As we will see below, the left-hand sides of the production rules are generalized by turning constants into variables, while the right-hand sides are developed by concatenating subplans and ordering them properly to form macro operators.

An example of the other kind of knowledge gained by HACKER—a generalized bug—is the demon:

```
(WATCH-FOR  (ORDER (PURPOSE 1line (ACHIEVE (ON a b)))
                   (PURPOSE 2line (ACHIEVE (ON b c))))
            (PREREQUISITE-CLOBBERS-BROTHER-GOAL
                              current-prog 1line 2line
                              (CLEARTOP b))).
```

It tells HACKER to watch for plans in which one step, `1line`, has the goal of achieving `(ON a b)` and a subsequent step, `2line`, has the goal of achieving `(ON b c)`. In such cases, the prerequisite of the second step—that `b` have a clear top—requires undoing the goal of the first step. When this demon detects such bugs, it invokes the `PREREQUISITE-CLOBBERS-BROTHER-GOAL` repair procedure to fix them.

Generalized bugs can also be viewed as production rules. This particular bug demon could be written as:

```
(ORDER  (PURPOSE 1line (ACHIEVE (ON a b)))
                       (PURPOSE 2line (ACHIEVE (ON b c)))) ⇒
(PREREQUISITE-CLOBBERS-BROTHER-GOAL
                  current-prog 1line 2line
                  (CLEARTOP b)).
```

HACKER learns both the left- and the right-hand sides of these bug demons.

HACKER's Architecture

HACKER is a complex program that contains several interleaved components (see Fig. D5c-1). These include:

1. The *planner*, which develops plans by pattern-directed expansion of planning operators;
2. The *critics' gallery*, which inspects the plans for known generalized bugs;
3. The *simulator*, which simulates the execution of the plans and checks for errors;
4. The *debugger and generalizer*, which locate and repair bugs in the plans for later use by the critics' gallery; and
5. The *generalizer and subroutinizer* which generalize plans and install them in HACKER's knowledge base.

The first two components comprise the performance element, which develops block-stacking plans. The simulator creates a performance trace of the simulated execution of the plan. The last two components perform the actual process of learning generalized subroutines and generalized bugs.

These components interact continually. As the planner is developing the plan, for example, the critics' gallery is interrupting to repair known bugs and the simulator is symbolically executing the evolving plan. The debugger may step in to fix a new bug and then resume the planning process. In this article, however, we describe each of these components separately and pretend that the plan is first developed in its entirety and then successively criticized, simulated, debugged, and generalized. This false architecture corresponds fairly closely to our simple model of learning multiple-step tasks. There are two learning elements, however: one for developing generalized subroutines

and one for developing generalized bugs. Figure D5c-1 summarizes this false architecture. We will explain the operation of HACKER by following the flow through this model.

HACKER's Performance Element: The Planner and the Critics' Gallery

HACKER employs a simple problem-reduction planner (Chap. XV; see also Article II.B2, in Vol. I), which is presented with an initial situation and a goal block-structure to create. Figure D5c-2 shows a sample situation and goal.

The goal is matched against HACKER's knowledge base of known plans, subroutines, and refinement rules. If a known plan or subroutine is found that

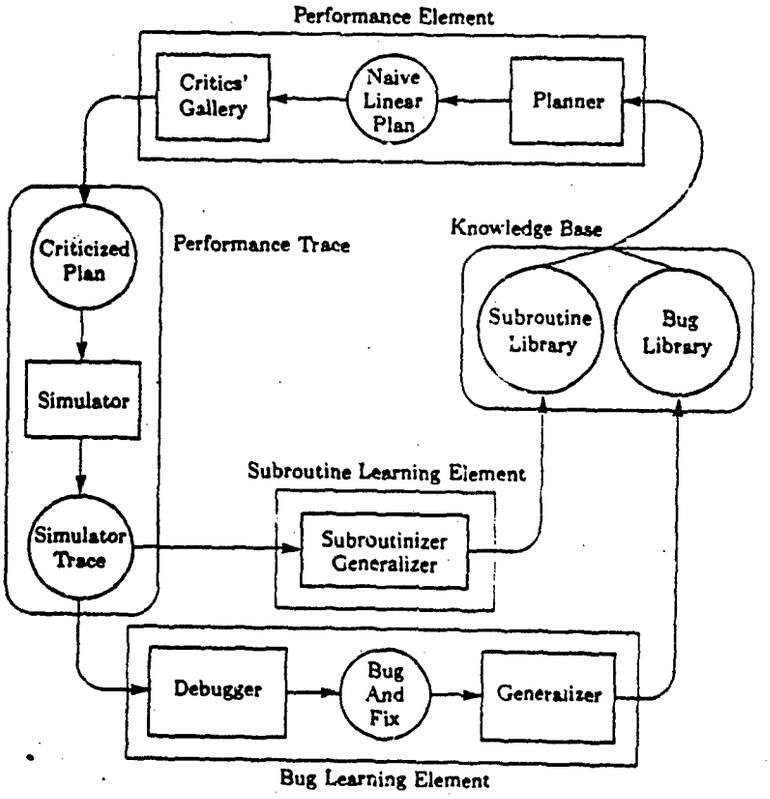
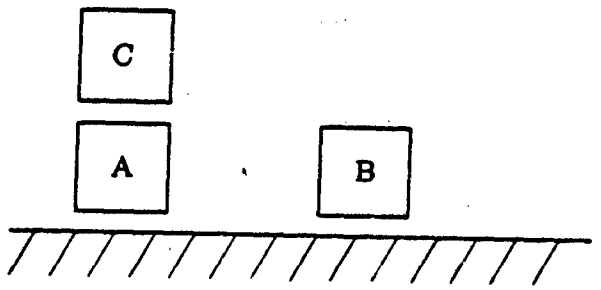


Figure D5c-1. A simplified architecture for HACKER.



Goal: `(ACHIEVE (AND (ON A B) (ON C A)))`

Figure D5c–2. A sample situation and goal.

can accomplish the goal, it is used. Otherwise, a refinement rule is applied to reformulate the goal as a set of subgoals. These subgoals, in turn, are matched against the knowledge base to locate known methods for achieving them. The expansion into subgoals proceeds until HACKER finds existing plans or primitive operators that can achieve each of the subgoals.

HACKER is noted for its *linearity assumption*. Whenever the planner is faced with the problem of achieving a pair of conjunctive subgoals, it assumes that they can be achieved *independently*. This assumption is represented in the `AND` rule for refining a conjunctive goal:

```
(TO  (ACHIEVE (AND a b))
     (AND  (ACHIEVE a)
           (ACHIEVE b))) .
```

This says "To achieve goals *a* and *b*, first achieve *a* and then achieve *b*." As a result of this linearity assumption, the plan developed by the planner is a naive plan that may not work (see Article XV.C).

The naive plan is criticized by the critics in the critics' gallery, which attempt to find instances of the generalized bugs kept in the bug library. When a bug is found, the associated bug fix is applied to improve the plan—usually by rearranging plan steps. The result of this criticism is a plan that reflects all of HACKER's past experience but still may not be correct.

HACKER's Performance Trace: Plans and Simulation

HACKER's plans contain a large amount of information about the planning process itself. Each step of a plan is justified by giving the *purpose* of the step—the subgoal it is intended to achieve. There are two fundamental kinds of steps: main steps and prerequisite steps. Main steps are directed at goals relating to the goals of the overall plan. Prerequisite steps are computations

needed to establish preconditions for the main steps. For example, the plan for the problem of Figure D5c–2 contains three steps:

Step 1. (PUTON C TABLE) [purpose: (CLEARTOP A) span: step 2] .

Step 2. (PUTON A B) [purpose: (ON A B) span: full plan] .

Step 3. (PUTON C A) [purpose: (ON C A) span: full plan] .

Steps 2 and 3 are main steps, while step 1 is a prerequisite step needed to clear off the top of A so that the robot can move A. As HACKER simulates the execution of the plan, it verifies that the goal of each step has been attained.

Each step in the plan also includes an indication of the *time span* of the goal it is attaining. The purpose of a step may be to accomplish something that will remain true for only a short time. In this example, (CLEARTOP A) will be true only until step 3. For HACKER to know that this is *not* a bug, step 1 includes a time-span indication that its goal is intended to be true only until the end of step 2.

The criticized plan is simulated to verify that it works properly. The simulator detects bugs in three forms: illegal operations, failed steps, and unaesthetic actions. An illegal operation is one that is considered impossible in the hypothetical blocks world. For instance, it is illegal to pick up a block unless it has a clear top. A failed step is one that does not achieve its goal for the designated time span. The simulator uses the goal information attached to each plan step to verify that at all times the goals intended by the planner have actually been met. Lastly, an unaesthetic action is a situation in which the robot moves the same block two times in succession without any intervening actions. These three methods for detecting bugs provide a *performance standard* for HACKER, which states that a plan must execute legally, achieve all intended goals and subgoals, and also be aesthetically correct. The simulation halts whenever one of these problems is identified, and a trace of the simulation is provided to the bug learning element.

HACKER's Learning Elements: The Subroutine Learning Element and the Bug Learning Element

As mentioned above, there are two learning elements in HACKER. One, the subroutine learning element, inspects the criticized plan and simulation trace to identify possible subroutines. The other, the bug learning element, examines the performance trace to diagnose and correct bugs uncovered by the simulation.

The subroutine learning element attempts to detect when two subgoals in the plan are sufficiently similar to allow a single subroutine to accomplish both. The trace of the planning and simulation processes indicates which constants in a goal or subgoal—for example, the constants A and B in the goal (ON A B)—can be generalized. A constant cannot be generalized if the

plan somehow refers to that constant explicitly (e.g., the constant TABLE has special status). HACKER generalizes each subgoal in the plan by turning all generalizable constants into variables. The generalized subgoal is then compared with all other goals in the program. Any two subgoals found to have an allowable common generalization are replaced by calls to a parameterized procedure. This generalization process is similar to the technique used in STRIPS to generalize macro operators.

As an example, consider the block-stacking task of Figure D5c-2. The initial plan involves separate steps for achieving (ON A B) and (ON C A). However, traces of the planning and simulation processes indicate that the code for (ON A B) will work for any variables u and v. The generalized goal (ON u v) is checked against other goals in the plan and found to match the subgoal (ON C A). As a result, HACKER formulates a generalized subroutine, (MAKE-ON u v), and replaces the subplans for steps 2 and 3 with calls to MAKE-ON. The MAKE-ON subroutine is placed in the knowledge base for use in future plans as well.

The subroutine learning element can be regarded as learning from examples. The goals and subgoals in a particular plan form the training instances, which are generalized by turning constants into variables. The distinctive aspect of the HACKER approach is that the search of the rule space is accomplished very directly. HACKER (and its predecessor, STRIPS) is able to reason about how the different steps in the plan depend on particular values for the arguments of the goal statement. From this dependency analysis, the correct generalization can be deduced directly. HACKER thus differs from most of the other learning methods described in this chapter in that it is able to use the meanings of its operators to guide the generalization process.

The bug learning element faces a much more difficult learning task. It must determine why the plan failed and repair the plan. Then it must attempt to generalize the discovered bug and create a bug critic that will prevent the bug from reappearing in future plans. The first task—determining why the plan failed—is the problem of credit assignment. The traditional credit-assignment problem is to determine which rule, used in the performance element, led to the mistake. In HACKER's case, there is one fundamental source of error: the linearity assumption as implemented by the AND rule. HACKER's credit assignment, instead, involves determining how the current planning task violates this linearity assumption—that is, how do the subplans in this problem interact?

HACKER's solution to the credit-assignment problem is to compare the intentions and expectations of the performance element with what actually happened. This approach again relies on knowledge of the semantics of the operators to assign blame to individual steps. This is more direct than the weaker, more empirical approach of comparing many possible plans obtained through a more widespread search, as in Samuel's checkers program and the LEX system.

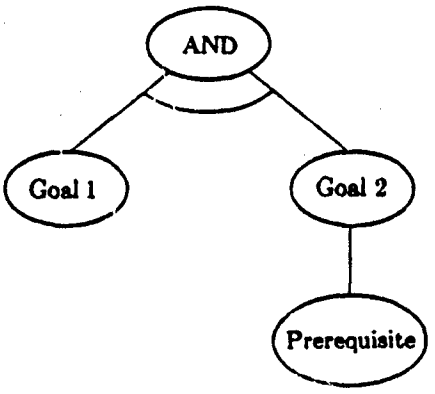


Figure D5c–3. The PREREQUISITE-CLOBBERS-BROTHER-GOAL bug schema.

HACKER has a small library of schemas that describe possible subgoal interactions. Credit assignment is accomplished by matching these schemas to the goal structure of the current plan and performance trace. For example, one class of interactions, the PREREQUISITE-CLOBBERS-BROTHER-GOAL, involves the goal structure depicted in Figure D5c–3.

The prerequisite step of goal 2 somehow makes goal 1 no longer true. For example, if the overall goal is (ACHIEVE (AND (ON A B) (ON B C))), we have the subgoal structure shown in Figure D5c–4.

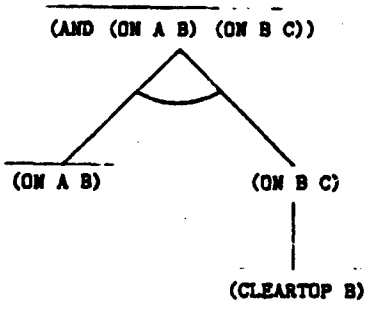


Figure D5c–4. A subgoal structure that matches the bug schema of Figure D5c–3.

HACKER simulates this plan by first placing block A on block B, then clearing off B so that it can place B on C. The clearing-off process makes (ON A B) false—the prerequisite of goal 2 has clobbered goal 1. (This is detected by the simulator when it checks the time span of each subgoal.)

Each of HACKER's bug schemas describes some general goal structure that can be matched to the goal structure of the current plan. The matching process is implemented in an ad hoc fashion as a series of six questions that the debugger asks of the performance trace. As a result of the matching process, the bug is ignored as innocuous, is properly classified, or is found to be too difficult to repair.

The process of repairing the plan is straightforward. Each bug schema contains instructions on how to repair the bug. These can involve reordering plan steps, creating new subplans that establish prerequisite conditions, and even removing unnecessary plan steps. The resulting repaired plan is simulated again to detect further bugs.

The process of generalizing the bug is also easily accomplished. Each bug schema contains instructions regarding which components of the goal structure can be generalized by turning constants into variables. For instance, the bug schema for PREREQUISITE-CLOBBERS-BROTHER-GOAL contains the instructions

```
(CSETQ  goal1 (VARIABLIZE (GOAL line1))
        goal2 (VARIABLIZE (GOAL line2))
        prereq (VARIABLIZE pre)),
```

where line1 refers to the first goal (whose prerequisite was clobbered), line2 refers to the search goal, and prereq refers to the prerequisite that did the clobbering. These instuctions tell HACKER to analyze the dependencies in the performance trace and generalize all three of these goal expressions. The resulting generalized goal structure shown in Figure D5c-5 is compiled into a demon and added to the bug library for use in subsequent criticism of naive plans.

The bug learning element can be regarded as learning by schema instantiation. Over time, HACKER discovers new situations in which particular kinds of subgoal interactions occur, generalizes these situations, and watches for them in future plans. It does not tackle the problem of discovering these classes of bugs in the first place, nor does it address the problem of discovering techniques for fixing bugs.

Conclusion

HACKER is a system that learns to develop plans for manipulating toy blocks. It acquires two kinds of knowledge—generalized subroutines and generalized bugs. Both of HACKER's learning elements make extensive use of the performance trace, which consists of the plan (annotated with goal information) and a trace of the simulated execution of the plan. The subroutine

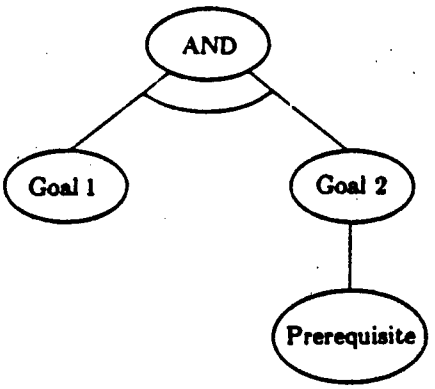


Figure D5c-3. The PREREQUISITE-CLOBBERS-BROTHER-GOAL bug schema.

HACKER has a small library of schemas that describe possible subgoal interactions. Credit assignment is accomplished by matching these schemas to the goal structure of the current plan and performance trace. For example, one class of interactions, the PREREQUISITE-CLOBBERS-BROTHER-GOAL, involves the goal structure depicted in Figure D5c-3.

The prerequisite step of goal 2 somehow makes goal 1 no longer true. For example, if the overall goal is (ACHIEVE (AND (ON A B) (ON B C))), we have the subgoal structure shown in Figure D5c-4.

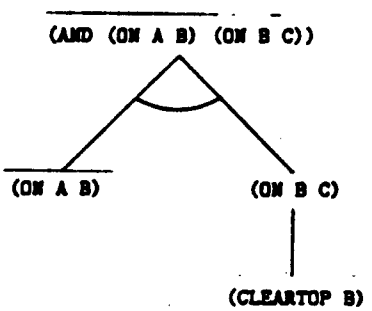


Figure D5c-4. A subgoal structure that matches the bug schema of Figure D5c-3.

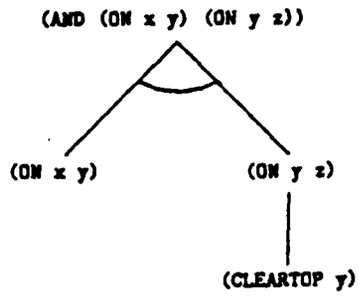


Figure D5c-5. A generalized goal structure.

learning element generalizes by analyzing the goal structure in the performance trace to determine which constants can be turned into variables. The bug learning element accomplishes credit assignment by instantiating schemas that describe bug-inducing goal structures. The schemas provide guidance for bug repair and generalization. Much of HACKER's impressive behavior derives from its ability to reason about the semantics of its task. The value of a transparent performance element for credit assignment and generalization is very evident in HACKER.

References

HACKER is described in Sussman's (1973) thesis. Doyle (1980) describes a formalization of the concepts of *goal* and *intention* as used by HACKER. An alternative to the linearity assumption is described in Article XV.D1.

D5d. LEX

LEX, a system designed by Thomas Mitchell (see Mitchell, Utgoff, and Banerji, in press; Mitchell, Utgoff, Nudel, and Banerji, 1981), learns to solve simple symbolic integration problems from experience. LEX is provided with an initial knowledge base of roughly 50 integration and simplification operators, some of which are shown in Table D5d–1. The goal of LEX is to discover *heuristics* for *when* to apply these operators. That is, LEX seeks to develop production rules of the form

$$\langle \text{situation} \rangle \Rightarrow \text{Apply operator OP}i,$$

where ⟨situation⟩ is a pattern that is matched against the current integration problem. The situations are expressed in a *generalization language* of possible patterns. For instance, a heuristic rule for operator OP12 might be:

$$\int f(x)\,\text{transc}(x)\,dx \Rightarrow \text{Apply OP12 with } u = f(x) \text{ and } dv = \text{transc}(x)\,dx.$$

This tells the LEX performance element that if it sees any problem whose integrand is the product of any function, $f(x)$, with a transcendental function, $\text{transc}(x)$, then it should apply OP12 with u bound to $f(x)$ and dv bound to $\text{transc}(x)\,dx$. The concepts of $f(x)$ and $\text{transc}(x)$ are part of the generalization language (illustrated later in Fig. D5d–4).

Mitchell calls these production rules *heuristics* because they provide heuristic guidance to LEX's performance element, which is a simple, forward-chaining production system (see Sec. II.B, in Vol. I). Without any heuristic rules, the performance element conducts a blind uniform-cost search (see Article II.C1, in Vol. I) of the space of all legal sequences of operator applications. Consider the problem of integrating $\int 3x \cos x\,dx$. Without any heuristics, LEX produces the rather large search tree shown in Figure D5d–1. It is no surprise that

TABLE D5d–1
Selected Integration Operators in LEX

OP02	convert	$\int x^r\,dx$	to	$x^{r+1}/(r+1)$	(power rule)
OP03	convert	$\int rf(x)\,dx$	to	$r\int f(x)$	(factor out a real constant)
OP06	convert	$\int \sin x\,dx$	to	$-\cos x$	
OP08	convert	$1 \cdot f(x)$	to	$f(x)$	
OP10	convert	$\int \cos x\,dx$	to	$\sin x$	
OP12	convert	$\int u\,dv$	to	$uv - \int v\,du$	(integration by parts)
OP15	convert	$0 \cdot f(x)$	to	0	

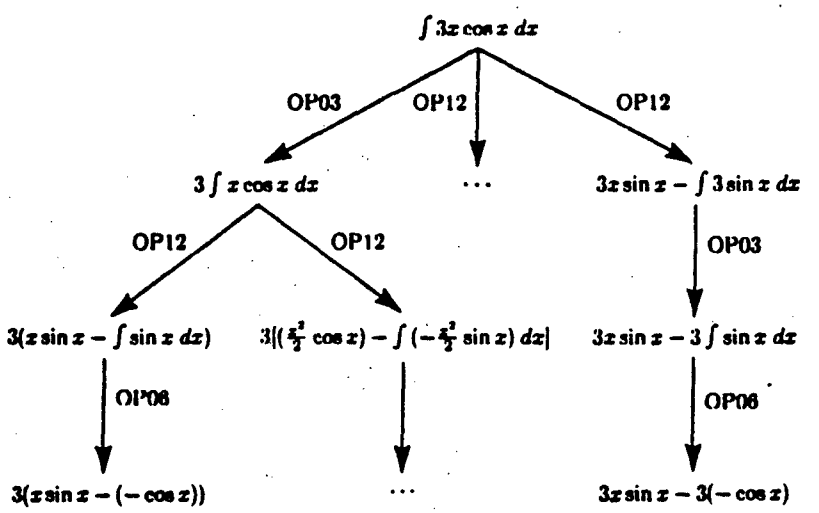


Figure D5d–1. Partial search tree for $\int 3x \cos x\, dx$ without heuristics.

when LEX has no heuristics, it often cannot solve integration problems before exhausting the time and space available to it.

The task of learning the left-hand sides of heuristic rules can be thought of as a set of concept-learning tasks. LEX tries to discover, for each operator OP*i*, the definition of the concept *situations in which* OP*i* *should be used*. It accomplishes this by gathering positive and negative training instances of the use of the operator. By analyzing a trace of the actions taken by the performance element, LEX is able to find cases of appropriate and inappropriate application of the operators. These training instances guide the search of a rule space of possible left-hand-side patterns. The candidate-elimination algorithm (see Article XIV.D3a) is employed to search the rule space, and partially learned heuristics, for which the candidate-elimination algorithm has not found a unique left-hand-side pattern, are stored as version spaces of possible patterns. Thus, the general form of a heuristic rule in LEX is:

(version space represented as S and G sets) $\Rightarrow$ Apply OP*i*.

For example, after a few training instances, LEX might have the following partially learned heuristic for the integration-by-parts heuristic, OP12:

Version space for OP12:

$G = \int f(x)g(x)\, dx \Rightarrow$ OP12, with $u = f(x)$ and $dv = g(x)\, dx$;

$S = \int 3x \cos x\, dx \Rightarrow$ OP12, with $u = 3x$ and $dv = \cos x\, dx$.

This heuristic tells LEX to apply OP12 in any situation in which the integral has the form $\int f(x)g(x)\,dx$. It also indicates that the correct left-hand-side pattern lies somewhere between the overly specific S pattern, $\int 3x\cos x\,dx$, and the overly general G pattern, $\int f(x)g(x)\,dx$. Below, we show how this partially learned heuristic was discovered by LEX.

LEX's Architecture

LEX is organized as a system of four interacting programs (see Fig. D5d-2) that correspond closely to our modified model of learning for multiple-step tasks. The *problem solver* is the performance element. It solves symbolic integration problems by applying the current set of operators and their heuristics. When the problem solver succeeds in solving an integral, a detailed trace of its performance is provided to the *critic*, which examines the trace to assign credit and blame to the individual decisions made by the problem solver. Once credit assignment is completed, the critic extracts positive (and negative) instances of the proper (and improper) application of particular operators. These training instances are used by the *generalizer* to guide the search for proper heuristics for the operators involved. Finally, the *problem generator* inspects the current contents of the knowledge base (i.e., the operators and their heuristics) and chooses a new problem to present to the problem solver.

LEX thus incorporates all four components of our simple model: the knowledge base (of operators and heuristics), the performance element, the performance trace, and the learning element (composed of the critic and the generalizer). Furthermore, LEX is one of the few AI learning systems to include an experiment planner—the problem generator.

In this article, we first present an example of how LEX solves problems and refines the version spaces of its heuristics. Then we describe each of LEX's components in detail and discuss some open research problems.

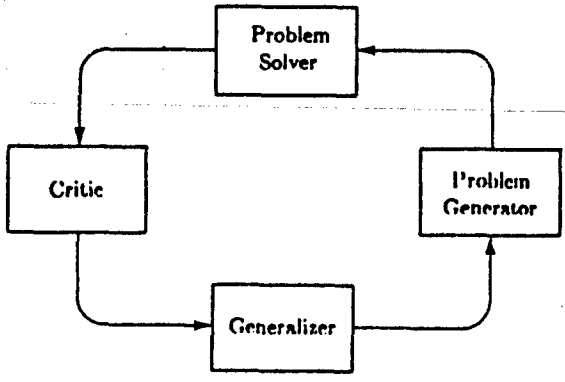


Figure D5d-2. LEX's architecture.

An Example

To show how LEX works, suppose that the problem generator has chosen the problem $\int 3x \cos x \, dx$ and the problem solver has produced the trace shown earlier in Figure D5d–1. The critic analyses the trace and extracts several training instances, including:

$$\int 3x \cos x \, dx \Rightarrow \text{OP12, with } u = 3x \text{ and } dv = \cos x \, dx \quad \text{(positive)}.$$

$$\int 3 \sin x \, dx \Rightarrow \text{OP03, with } r = 3 \text{ and } f(x) = \sin x \quad \text{(positive)}.$$

$$\int \sin x \, dx \Rightarrow \text{OP06} \quad \text{(positive)}.$$

We will watch how the generalizer handles the training instance for OP12. Let us assume that this is the first training instance that has been found for this operator, so the knowledge base does not yet contain any heuristics for when to use it. Consequently, the generalizer will create and initialize a new OP12 heuristic. The left-hand side of the heuristic is a version space of the form:

Version space for OP12:

$G = \int f(x)g(x) \, dx \Rightarrow$ OP12, with $u = f(x)$ and $dv = g(x) \, dx$;

$S = \int 3x \cos x \, dx \Rightarrow$ OP12, with $u = 3x$ and $dv = \cos x \, dx$.

Notice that S is a copy of the training instance and G is the most general pattern for which OP12 is *legal*. This heuristic will recommend that OP12 be applied in any problem whose integrand is less general than $\int f(x)g(x) \, dx$. This is not a highly refined heuristic.

To see how LEX refines this heuristic, let us assume that the other training instances shown above have been processed. At this point, the problem generator chooses the problem $\int 5x \sin x \, dx$ to solve. The problem solver will apply OP12, since the G set of the heuristic matches the integrand. Figure D5d–3 shows a portion of the solution tree.

Some of the training instances extracted by the critic are:

$$\int 5x \sin x \, dx \Rightarrow \text{OP12, with } u = 5x \text{ and } dv = \sin x \, dx \quad \text{(positive)}.$$

$$\int 5 \cos x \, dx \Rightarrow \text{OP03, with } r = 5 \text{ and } f(x) = \cos x \quad \text{(positive)}.$$

$$\int \cos x \, dx \Rightarrow \text{OP10} \quad \text{(positive)}.$$

$$\int 5x \sin x \, dx \Rightarrow \text{OP12, with } u = \sin x \text{ and } dv = 5x \, dx \quad \text{(negative)}.$$

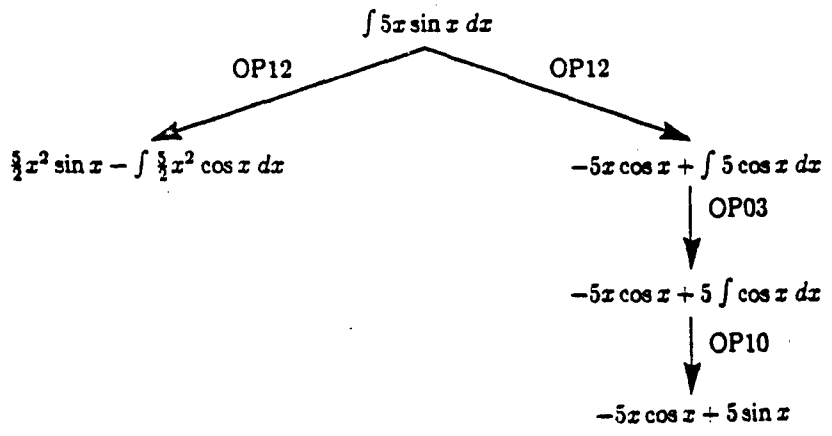


Figure D5d-3. The solution tree for $\int 5x \sin x\, dx$.

The generalizer updates the version space for OP12 to contain:

$G = \{g_1, g_2\}$, where
g_1: $\int \text{polynom}(x) g(x)\, dx \Rightarrow$ OP12,
with $u = \text{polynom}(x)$ and $dv = g(x)\, dx$;
g_2: $\int f(x) \text{transc}(x)\, dx \Rightarrow$ OP12,
with $u = f(x)$ and $dv = \text{transc}(x)\, dx$;

$S = \{s_1\}$, where
s_1: $\int kx \text{trig}(x)\, dx \Rightarrow$ OP12,
with $u = kx$ and $dv = \text{trig}(x)\, dx$.

The positive training instance forces the constants 3 and 5 to be generalized to k, which represents any integer constant, and "sin" and "cos" to be generalized to "trig," which represents any trigonometric function, as shown in s_1. Similarly, the negative training instance leads to two alternative specializations. In g_1, f was specialized to "polynom" to avoid $u = \sin x$, and in g_2, g was specialized to "transc" to avoid $dv = 5x\, dx$. These two specializations no longer cover the negative training instance. With a few more training instances, the heuristic for OP12 converges to the form shown at the start of this article, that is, $\int f(x) \text{transc}(x)\, dx$. The concepts "$k$," "trig," "polynom," and so on, are all part of the generalization language known to LEX from the start (see Fig. D5d-4, shown later).

Now that we have seen an example of LEX in action, we describe each of the four components of LEX in turn.

The Problem Solver

As discussed above, the problem solver conducts a forward search of possible operator applications in an attempt to solve the given integration problem. Initially, this search is blind. However, as the heuristics for the operators are refined, the search becomes more focused.

The problem solver conducts a *uniform-cost search*. At each step, it chooses the one expansion of the search tree that has the smallest estimated cost. The search tree is maintained as a list of *open* nodes—that is, nodes to which not all legal integration operators have been applied. The cost of an open node is measured by summing the cost of each search step (for both time and space) back to the root of the search tree. In addition, the cost of a proposed expansion is weighted to reflect the strength of the heuristic advice available. In detail, the problem solver chooses an expansion as follows:

Step 1. For each open node and each legal operator, compute the "degree of match" according to the formula:

- 0 if no heuristic recommends this operator for this node;
- m/n if there is a heuristic, and m out of the n patterns in the boundary sets of the version space (i.e., the S and G sets) match the current situation.

Step 2. Choose the expansion that has the lowest weighted cost, computed as:

(1.5 − degree of match) × (cost so far + estimated expansion cost).

The effect of the (1.5 − degree of match) weight on the cost is to emphasize the cost of the path when little heuristic guidance is available but to ignore cost considerations as the heuristic recommendation becomes stronger.

The problem solver continues to select nodes and apply operators until the integral is solved. Notice that, in LEX, a simple performance standard is available: solution of the integral. This is a substantially simpler situation than that faced by Waterman's poker player, which needs to play several hands to evaluate how well it is doing. LEX knows when it is doing well. LEX also knows when it is doing poorly. For each integration problem, the problem solver is given a time and space limit. If it runs out of time or space before solving the problem, it gives up and the problem generator selects a new problem to solve.

The Critic

The problem solver provides the critic with a detailed trace of each successfully solved problem. The critic's task is to extract positive and negative training instances from this trace by assigning credit and blame to individual

decisions made by the problem solver. The critic solves the credit-assignment problem as follows:

1. Every search step along the minimum-cost solution path found by the problem solver is a positive instance;
2. Every step that (a) leads from a node on the minimum-cost path to a node not on this path and (b) leads to a solution path whose length is greater than or equal to 1.15 times the length of the minimum-cost path is a negative instance.

These criteria are intended to produce applicability heuristics that guide the performance element to *minimum-cost solutions*. To evaluate these criteria (especially 2b), the critic must re-invoke the problem solver to follow out paths that appear to be bad. This deeper search is in some ways analogous to the deep search Samuel used in his checkers-playing program for solving the credit-assignment problem. The criterion of minimum-cost solution is convenient because it can be measured by the computer itself—by its own experience in attempting to solve the problem.

The critic is fairly conservative. It provides the generalizer only with the training instances that can be most reliably credited or blamed. However, the critic is not infallible. It can produce false positive and false negative training instances when the knowledge base contains incorrect heuristics. Since the problem solver follows the guidance provided by the heuristics in the knowledge base, it may believe it has found the lowest cost solution when in fact, the heuristics have led it astray. Since LEX does not conduct an exhaustive search of the space, it will not always detect this fact. As a result, the critic may create false positive and false negative instances. Its reliability can be improved by increasing the safety factor (normally 1.15) when the problem solver is re-invoked by the critic. This causes the problem solver to search more deeply along alternative paths and improves the chances of finding the true minimum-cost path.

The Generalizer

The generalizer simply applies the candidate-elimination algorithm to process each of the training instances provided by the critic and to refine the version spaces of each of the operators. The multiple-boundary-set form of the algorithm (see Article XIV.D3a) was adopted to handle erroneous training instances.

The generalizer is able to learn disjunctions in certain cases. During generalization based on a positive training instance, for example, if the version space would normally be forced to collapse because no consistent rule exists, a second version space is created instead. This second version space contains the patterns that are consistent with all of the negative instances and the single new positive instance. As additional positive instances are received,

they are processed against any version space whose G set covers them. When more than one heuristic rule is created for a single operator, the effect is the same as if a single disjunctive heuristic had been developed.

The generalization language (and, thus, the rule space) in LEX is based on the tree of functions shown in Figure D5d–4. The most general pattern is $f(x)$, that is, any real function. The most specific functions are integer and real constants, sine, cosine, tangent, and so on. This language is known to have shortcomings (e.g., it cannot describe the class of twice continuously differentiable functions), but it is adequate for expressing some of the heuristics useful in the domain of symbolic integration.

LEX relies entirely on syntactic generalization methods. It cannot, for example, analyze the solution of $\int 3x \cos x \, dx$ and realize that, since OP03 requires only a real constant r, the particular constant 3 can be generalized to any real constant. This kind of analysis, based on the semantics of the operators, is done in STRIPS and HACKER. The advantage of LEX's syntactic approach is that it is general—it can be applied to any generalization language.

The Problem Generator

The purpose of the problem generator is to select a set of integration problems that form a good teaching sequence (see Article XIV.A). This portion of LEX is still under development, so only some strategies that have been proposed for the design of the problem generator are discussed here.

One strategy for selecting a new problem is to find an operator whose version space is still unrefined and select a problem that "splits" the version space—that is, an integral that matches only half of the patterns in the S and G sets. If the problem solver can solve such a problem, LEX will be able to refine the version space for that operator.

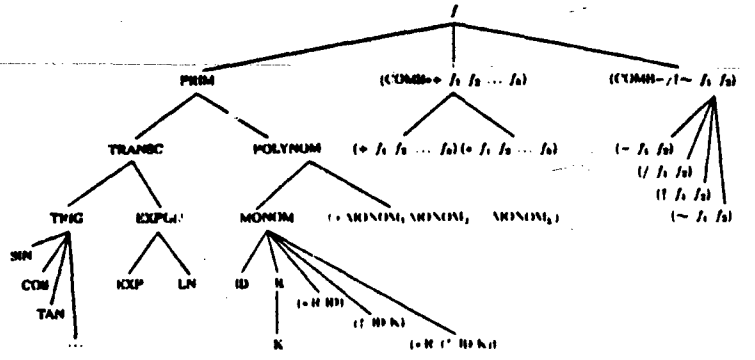


Figure D5d–4. Function hierarchy used in LEX's generalization language.

A second, related strategy is to take a problem that LEX has already solved and modify it in some way. For instance, having solved the integral $\int 3x \sin x \, dx$, LEX could consider attempting the integral $\int 5x \sin x \, dx$. This would force it to generalize its version space to indicate that any constant could appear (not just 5 or 3). The generalization hierarchy in Figure D5d–4 can be used to create such training problems.

A third strategy is to look for overlaps in the knowledge base. If there are two operators whose version spaces overlap, the problem generator can choose a problem for which both operators are believed to be applicable. The resulting attempt to solve the problem may show that only one of the operators should be used in such situations.

Finally, when LEX is just beginning to learn, it may be necessary to apply the inverses of the integration operators to create problems of known difficulty for the problem solver to solve. This is analogous to the technique of providing students in chemistry courses with an "unknown" that is, in fact, deliberately synthesized by the professor. LEX must learn how to control its search so that it can solve the training problem without being overwhelmed by combinatorial explosion.

The problem generator, more than any other component of the LEX system, must have meta-knowledge of what LEX already knows and where its weaknesses are. It must keep a history of previous problem-solving attempts, so that it does not repeatedly propose unsolvable or uninformative problems. The design of the problem generator is, in fact, the most difficult part of the LEX project.

Conclusion

LEX learns *when* to apply the standard operators of symbolic integration. For each integration operator, the system learns a heuristic pattern. The problem solver matches these patterns against the expression being integrated to determine which operators should be applied. LEX obtains training instances by observing its own attempts to solve integration problems. Similarly, LEX obtains its performance standard by computing the cost of the shortest solution path that it found when it tried to solve the problem. The credit-assignment problem is solved by conducting a deeper search and crediting those decisions that led to the minimum-cost solution. Decisions that caused the problem solver to depart from the minimum-cost path are blamed. Positive and negative training instances are thus extracted and processed by the generalizer to update the version spaces of the integration operators.

Experiment planning is implemented in LEX by the problem generator, which employs a variety of strategies to select problems that will help the other components of the system refine the knowledge base.

The primary weakness of LEX, and a source of its generality, is that it employs only syntactic methods of generalization. It is unable to reason

about the meanings of its operators, and thus it cannot use knowledge about dependencies among operators to determine how the heuristics should be generalized.

LEX does not attack the problems of learning new operators (i.e., right-hand sides of heuristic rules) or learning operator sequences (i.e., macros). To learn a new integration operator, LEX would need much more knowledge about mathematics and the goals of integration. This is a very difficult learning problem. The problem of learning macro operators (i.e., useful sequences of operators) and their applicability conditions has been addressed in HACKER and STRIPS. Further work on LEX may include the learning of such operators.

References

Mitchell, Utgoff, and Banerji (in press) and Mitchell, Utgoff, Nudel, and Banerji (1981) provide descriptions of LEX.

D5e. Grammatical Inference

MOST AI RESEARCHERS employ numerical or logical representations in their learning systems. In work on adaptive systems, for example, the concept to be learned is often represented as a vector of numerical weights. Most of the other systems described in this chapter represent their knowledge in logic-based description languages (e.g., predicate calculus, semantic nets, feature vectors). A number of researchers, however, have developed systems that employ formal grammars to represent the learned concepts. This article discusses the body of work, known as *grammatical inference*, that seeks to learn a grammar from a set of training instances.

The primary interest in grammar learning can be traced to the use of formal grammars for modeling the structure of natural language (see Chomsky, 1957, 1965). The question of how people learn to speak and understand language led to studies of language acquisition; interest in modeling the languages of other cultures encouraged the development of computer programs to help field researchers construct grammars for unfamiliar languages (Klein and Kuppin, 1970); and recent attempts by pattern-recognition researchers to use grammars to describe handwritten characters, visual scenes, and cloud-chamber tracks have created a need for grammatical-inference techniques. Thus, all of these researchers are interested in methods for learning a grammar from a set of training instances.

A grammar is a system of rules describing a *language* and telling which sentences are allowed in the language (see Article IV.C1, in Vol. I). Grammars can describe natural languages—that is, languages spoken by people—and formal languages—that is, simple languages amenable to mathematical analysis. In natural languages, grammar rules indicate the generally accepted ways of constructing sentences. In formal languages, however, grammars are applied much more strictly. A formal grammar for a language, L, can be viewed as a predicate that tells, for any sentence, whether it is *grammatical*, that is, "in" the language L, or *ungrammatical*, that is, not a legal sentence in L. From this formal perspective, a language is simply a potentially infinite set of all legal sentences, and a grammar is simply a description of that set.

One might expect the task of learning a grammar to be the same as the task of learning a single concept (see Sec. XIV.D3), since a single concept can also be viewed as a predicate describing some set of objects. Usually, however, this is not the case. Most formal languages are too complex to be described by a single concept or rule. Instead, a grammar is usually written as a set of rules that describe the *phrase structure* of the language. For example, we might have one rule that says: *A sentence is an article followed by a noun phrase followed by a verb phrase*. This could be written as the grammar rule:

$$\langle \text{sentence} \rangle \rightarrow \langle \text{article} \rangle \langle \text{noun phrase} \rangle \langle \text{verb phase} \rangle .$$

This rule describes the overall structure of a sentence. Of course, there are many different kinds of noun and verb phrases. These can also be described by phrase-structure rules. We might, for example, write another rule

$$\langle \text{verb phrase} \rangle \rightarrow \langle \text{verb} \rangle$$

for the simplest case in which the verb phrase is just a single word, as in *The boy cried.* A more complex verb phrase could be written as

$$\langle \text{verb phrase} \rangle \rightarrow \langle \text{verb} \rangle \langle \text{article} \rangle \langle \text{noun phrase} \rangle$$

for sentences like *The program learned the grammar.*

A grammar can thus be built out of a set of phrase-structure rules (also called *productions*). These rules break the problem of determining whether a sentence is *grammatical* into the subproblems of determining whether it is composed, for example, of a grammatical article followed by a grammatical noun phrase followed by a grammatical verb phrase. In this way, the single concept *grammatical sentence* is broken into the subconcepts of *noun phrase* and *verb phrase*. Moreover, such subconcepts are not independent but interact according to the grammar rules. Thus, determining whether a sentence is grammatical is a *multiple-step* task involving the sequential application of phrase-structure rules. It is for this reason that we include grammatical inference in our survey of systems that learn to perform multiple-step tasks.

In this article, we first introduce formal grammars and their uses and then discuss the theoretical limits of grammatical inference. The problem of learning a grammar from training instances has received a fair amount of mathematical analysis. We describe the principal results of this work along with their relevance for practical learning systems. Finally, we present the four major methods that have been developed for learning grammars.

Grammars and Their Uses

In the theory of formal languages, a language is defined as a set of strings, where each string is a finite sequence of symbols chosen from some finite vocabulary. In natural languages, the strings are sentences, and the sentences are sequences of words chosen from some vocabulary of possible words. To describe languages, Chomsky (1957, 1965) introduced a hierarchy of classes of languages based on the complexity of their underlying grammars. We will focus primarily on the *context-free languages* (and grammars).

A context-free language is defined by the following:

1. A *terminal vocabulary* of symbols—the words of the language;
2. A *nonterminal vocabulary* of symbols—the syntactic categories (e.g., "noun," "verb") of the language;

3. A set of *productions*—the phrase-structure rules of the language; and
4. The *start symbol*.

The best way to understand these definitions is by considering an example. Examine the following context-free grammar, G, with

(a) the terminal vocabulary {a, the, boy, girl, petted, held, puppy, kitten, wall, hill, by, on, with};

(b) the nonterminal vocabulary $\{Z, S, V, A, P, W, O, X\}$;

(c) the productions

$Z \rightarrow ASV$,
$V \rightarrow X$, $V \rightarrow XAO$, $V \rightarrow VP$,
$P \rightarrow WAS$, $P \rightarrow WAO$,
$A \rightarrow$ a, $A \rightarrow$ the,
$S \rightarrow$ boy, $S \rightarrow$ girl,
$W \rightarrow$ by, $W \rightarrow$ on, $W \rightarrow$ with,
$O \rightarrow$ puppy, $O \rightarrow$ kitten, $O \rightarrow$ hill, $O \rightarrow$ wall,
$X \rightarrow$ petted, $X \rightarrow$ held; and

(d) the start symbol, Z.

This grammar, G, describes a language of simple sentences such as *The boy held the puppy* and *The girl on the hill held a kitten*. It describes a sentence by *deriving* it from the start symbol. We start with the symbol Z and choose a production that has Z as the left-hand side. There is only one such rule in G: $Z \rightarrow ASV$. We apply this rule by *rewriting* Z as the string ASV. Now we choose one of the nonterminals, A, S, or V, and find a rule that can be used to rewrite it. If we choose the rule $V \rightarrow XAO$, our current sentence becomes $ASXAO$. We continue rewriting nonterminals (according to the production rules) until the sentence contains only *terminal* symbols. A complete derivation for the sentence *The boy held the puppy* is as follows:

Current sentence	*Chosen production rule*
Z	
	($Z \rightarrow ASV$)
ASV	
	($V \rightarrow XAO$)
$ASXAO$	
	($A \rightarrow$ the)
the $SXAO$	
	($S \rightarrow$ boy)
the boy XAO	
	($X \rightarrow$ held)
the boy held AO	
	($A \rightarrow$ the)
the boy held the O	
	($O \rightarrow$ puppy)
The boy held the puppy	

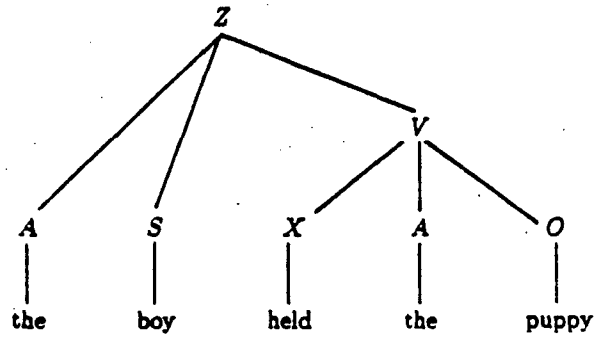


Figure D5e–1. Derivation tree for the sentence *The boy held the puppy.*

This is usually depicted as a derivation tree (see Fig. D5e–1).

Depending on which rules we choose during the rewriting process, we get different sentences. If we choose "$O \rightarrow$ kitten" instead of "$O \rightarrow$ puppy," we get the sentence *The boy held the kitten.* The context-free language described by G is the set of *all possible* sentences that can be derived from Z by the rewrite rules in G. Notice that we can also start our derivation with some symbol other than Z. If we start with the nonterminal V, for example, we generate the *sublanguage* of all verb phrases in G. Each nonterminal has a sublanguage. Thus, each nonterminal represents a subconcept, such as *noun phrase* (S) or *verb phrase* (V), of the overall concept of *grammatical sentence* (Z).

In pattern recognition and language understanding, the performance task facing a computer program is not the generation of grammatical sentences but their recognition. Given a sentence, the problem of determining whether it is grammatical—that is, of finding a derivation for the sentence—is called *parsing*. Many efficient algorithms have been developed for parsing sentences in context-free languages (see Article IV.D, in Vol. I; Hopcroft and Ullman, 1969).

Extensions to Context-free Grammars

Context-free grammars are able to capture much of the structure of natural and artificial languages, especially computer programming languages. However, many problems require extensions to the basic context-free grammar framework.

Transformational grammars. Some characteristics of natural language cannot be modeled with context-free grammars. One example that is frequently cited is the "respectively" construction in sentences such as *The*

boy and the girl held the puppy and the kitten, respectively. Other examples include the conversion of sentences from active to passive voice and discontinuous constituents like *throw out* in the sentence *He threw the junk out.* In response to these shortcomings of context-free grammars, Chomsky (1965) developed the theory of *transformational grammar* (see Article IV.C2, in Vol. I), in which a sentence is first derived as a so-called *deep structure*, then manipulated by *transformation rules*, and finally converted into *surface form* by phonological rules. The deep structure, which corresponds to the basic declarative meaning of the sentence, is derived by a context-free grammar. The transformation rules can modify the structure—but not the meaning—by altering the derivation tree. For example, a transformation rule can convert a declarative sentence into a question by flipping branches of the tree to change the word order. Under such a transformation, the sentence *The boy is holding the dog* becomes the question *Is the boy holding the dog?* Some methods have been developed for learning transformation rules, as well as context-free grammars, from examples. Particular attention has been given to learning these rules under conditions believed to be similar to those under which a child learns a language.

Stochastic grammars. Although context-free grammars (and transformational grammars) can represent the phrase structure of a language, they tell nothing about the relative frequency or likelihood of appearance of a given sentence. It is common, for instance, in context-free grammars to use *recursive productions* to represent repetition. In our sample grammar above, the production $V \rightarrow VP$ is recursive. If we apply it over and over again, we can generate sentences like *The boy held the puppy on the wall by the hill with the kitten...* Although the sentence is technically grammatical, it would be nice to represent the degree of acceptability of such a sentence.

Stochastic grammars provide one approach to this problem. Each production in a stochastic grammar is assigned a probability of selection—that is, a number between zero and one. During the derivation process, productions are selected for rewriting according to their assigned probabilities. Consequently, each string in the language has a probability of occurrence computed as the product of the probabilities of the rules in its derivation. If we took our sample grammar, for instance, and assigned probabilities of .5 to all of the rules except $X \rightarrow ASV$ (probability 1.0) and $V \rightarrow XAO$ (probability .33), the string "The boy held the puppy" has probability $1(.33)(.5)(.5)(.5)(.5)(.5) = .01$, while the string "The boy held the puppy on the wall by the hill with the kitten" has probability 1.58944×10^{-7}. This expresses the intuition that the second sentence is very unlikely to be considered acceptable.

Stochastic grammars have been employed by pattern recognition researchers in noisy and uncertain environments where it is better to have an indication of the degree of grammaticality of a sentence than a single yes-no decision. Stochastic grammars also allow grammatical-inference programs to

represent uncertainty about the true language when noisy and unreliable training instances are presented.

Graph grammars. In syntactic pattern-recognition problems, it is often important to represent the two- or three-dimensional structure of "sentences" in the language. Traditional context-free grammars, however, generate only one-dimensional strings. Context-free *graph grammars* have been developed that construct a *graph of terminal nodes* instead of a string of terminal symbols (see Article XIII.E3). Rewrite rules in the grammar describe how a nonterminal node can be replaced by a subgraph. Evans (1971) employs a set of graph grammars to describe visual scenes. Other researchers have applied graph grammars to the pattern recognition of handwritten characters and cloud-chamber tracks. This latter use of grammars is especially appropriate in that the rewrite rules in the grammar directly correspond to properties of the pattern. For example, subatomic particles decay into other particles only in certain ways, and these decay events can be modeled naturally with productions whose left-hand sides have the decaying particles and whose right-hand sides state the corresponding particles into which they decay.

Theoretical Limitations of Grammatical Inference

Now that we have reviewed some of the important kinds of formal languages and grammars, we turn our attention to the problem of learning these formal languages from examples. As with other forms of learning from examples, it is profitable to view grammatical inference as a search through a rule space of all possible context-free grammars for a grammar that is consistent with the training instances chosen from an instance space. In language learning, the training instances are usually sample sentences that have been classified by a teacher to indicate whether or not they are grammatical. The goal of the grammatical-inference program is to find a grammar for the "true" language that underlies the training instances.

Under what conditions is it possible to learn the correct context-free language from a set of training instances? This question has received a fair amount of study, and several results have been obtained. The most important result is that it is impossible to learn the correct language (or the correct single concept) from positive examples alone. Gold (1967) proved that if a program is given an infinite sequence of positive examples—that is, sentences known to be "in" the language—the program cannot determine a grammar for the correct context-free language in any finite time. To see why this is so, consider that at some point the program has received k strings $\{s_1, s_2, \ldots, s_k\}$. There are many possible languages that are consistent with these examples. The most general, *universal language*, which contains all possible strings of the terminal symbols, certainly contains all of the strings in the sample. Similarly, the trivial language $L = \{s_1, s_2, \ldots, s_k\}$ is the most specific language that

contains all of the strings in the sample. There are many possible languages between these two extremes. No finite sample will allow the learning program to choose the correct language from these various possibilities.

Fortunately, in most learning situations, additional information is available that can help constrain the choices of the learning program so that a reasonable language, and its grammar, can be found. Let us examine possible sources of this additional information.

Negative examples. Negative training instances allow the program to eliminate grammars that are too general (see Article XIV.D3a, on the candidate-elimination algorithm). Gold (1967) showed that if the learning program could pose questions to an *informant*, that is, ask a person whether or not a given string was grammatical, the true language could be learned. The informant could be used to obtain complete positive and negative examples and thus determine exactly the true language. Gold called this learning situation *informant presentation.*

Stochastic presentation. When a program is trying to learn a *stochastic* context-free grammar, learning is also possible if the training instances are presented to the program repeatedly, with a frequency proportional to their probability of being in the language. In this *stochastic-presentation* method, the program can estimate the probability of a given string by measuring its frequency of occurrence in the finite sample. In the limit, stochastic presentation gives as much information as informant presentation of positive and negative examples: Ungrammatical strings have zero probability, and grammatical strings have positive probability.

Prior distributions. As we have seen above, even after a set of positive instances has been processed, there are still many possible languages, and hence many possible grammars, for the learning program to choose from. Furthermore, even when a unique language has been determined, as with informant presentation, there may be several different grammars that all generate the same language. One way to tell a program how to choose the right grammar is to define a prior probability (or desirability) distribution over all possible grammars. The program can then choose the most probable grammar that is consistent with the training instances. Horning (1969) employs a prior distribution that makes simple grammars more likely than complex ones, where simple grammars are those that have fewer nonterminals, fewer productions, shorter right-hand sides, and so on.

Semantics. According to cognitive psychologists, children receive little negative feedback when they are learning a language. Consequently, we are faced with the puzzle of how people are able to learn natural language almost entirely from positive training instances. One important source of information for children may be the meaning of the sentences they hear. A few psychological theories, and some computer programs (see below), have been developed that incorporate semantic constraints as a source of information. These theories basically claim that the grammatical structure of a language

parallels the semantic structure of the internal representation that people employ.

Structural presentation. One technique employed by pattern recognition researchers to aid grammatical inference is *structural presentation*, in which the program is given some information about the derivation tree of the sample sentences. This is similar to the use of book training in Samuel's checkers program. The derivation tree provides a move-by-move (or, in this case, a rule-by-rule) performance standard along with each training instance.

Grammar restriction. One final way to get around Gold's results is to learn only special subclasses of the context-free languages. In particular, grammatical inference is much easier for *regular* and *delimited* languages, which, though not as powerful as the context-free languages, have important practical applications.

In summary, then, although Gold's theorems show that the formal problem of learning a context-free grammar from positive instances alone is impossible, there are many alternative sources of information that allow programs, and presumably people, to learn language.

Methods of Grammatical Inference

In this section, we survey four basic techniques that have been used to learn context-free grammars from training instances. The various methods, some of which parallel the basic learning methods discussed in Article XIV.D1, differ primarily in the way that they search the rule space and the kinds of information that they use to guide that search.

The first approach we discuss is *enumeration*. Enumerative, or generate-and-test, methods propose possible grammars and then test them against the data. The second basic grammatical-inference technique is *construction*. Constructive methods usually learn from positive examples only. They collect information about the structure of the sample strings and use it to build a grammar reflecting that structure. *Refinement* methods form a third important class of grammatical-inference techniques. They start with a hypothesis grammar and gradually improve it by means of various heuristics based on additional training instances. Finally, *semantics-based* methods employ knowledge of the meanings of the sample sentences to decide how to search the rule space. Most semantics-based methods have been developed to model how children learn natural languages.

Rules of generalization and specialization for grammars. Before describing these learning methods in more detail, we first discuss three methods for the syntactic generalization and specialization of grammars:

1. *Merging.* A context-free grammar can be generalized by an operation called *merging*. Suppose the grammar G contains two nonterminals, A

and B. We can modify G to obtain a more general grammar by merging A and B—that is, by creating a new nonterminal, Q, and replacing all occurrences of A and B by Q. This has the effect of pooling the sublanguages of A and B to create a new sublanguage, Q, whose strings may appear anywhere that either the strings of A or the strings of B could have appeared. Suppose, for example, that in our sample grammar discussed above, we merged S (subjects) and O (objects) to obtain Q. The productions of the grammar G become:

$Z \rightarrow AQV$
$V \rightarrow X, \quad V \rightarrow XAQ, \quad V \rightarrow VP,$
$P \rightarrow WAQ,$
$A \rightarrow$ a, $\quad A \rightarrow$ the,
$W \rightarrow$ by, $\quad W \rightarrow$ on, $\quad W \rightarrow$ with,
$Q \rightarrow$ puppy, $\quad Q \rightarrow$ kitten, $\quad Q \rightarrow$ hill, $\quad Q \rightarrow$ wall,
$Q \rightarrow$ boy, $\quad Q \rightarrow$ girl,
$X \rightarrow$ petted, $\quad X \rightarrow$ held.

Previously ungrammatical sentences like *The puppy petted the boy* are now allowed. The language is thus larger and, consequently, more general.

2. *Splitting.* The inverse of merging is a specialization process called *splitting.* We can specialize a grammar by splitting the sublanguage of one nonterminal, N, into two smaller sublanguages, N_1 and N_2. This is accomplished by replacing some occurrences of N in the grammar by N_1 and others by N_2. In the grammar above, for instance, we could split the A (article) nonterminal into A_1 and A_2 to obtain the grammar:

$Z \rightarrow A_1 QV,$
$V \rightarrow X, \quad V \rightarrow XA_2 Q, \quad V \rightarrow VP,$
$P \rightarrow WA_2 Q,$
$A_1 \rightarrow$ a, $\quad A_2 \rightarrow$ the,
$W \rightarrow$ by, $\quad W \rightarrow$ on, $\quad W \rightarrow$ with,
$Q \rightarrow$ puppy, $\quad Q \rightarrow$ kitten, $\quad Q \rightarrow$ hill, $\quad Q \rightarrow$ wall,
$Q \rightarrow$ boy, $\quad Q \rightarrow$ girl,
$X \rightarrow$ petted, $\quad X \rightarrow$ held.

Now all sentences must begin with "a," and all prepositional phrases and object phrases must use "the." The previously grammatical sentence *The boy petted the puppy* is now illegal. This language is therefore more specialized.

3. *Disjunction.* One operation that is similar to merging is called disjunction. In disjunction, we choose two *strings*, s_1 and s_2, and create a new nonterminal, D, whereby the rules $D \rightarrow s_1$ and $D \rightarrow s_2$ are added to the grammar. Every occurrence of the strings s_1 and s_2 in existing productions is replaced by D. For example, we could disjoin AO and AS in our sample grammar to create the new nonterminal, N (noun phrase). The grammar then becomes:

$Z \rightarrow NV$,
$V \rightarrow X$, $V \rightarrow XN$, $V \rightarrow VP$,
$P \rightarrow WN$,
$N \rightarrow AS$, $N \rightarrow AO$,
$A \rightarrow$ a, $A \rightarrow$ the,
$S \rightarrow$ boy, $S \rightarrow$ girl,
$W \rightarrow$ by, $W \rightarrow$ on, $W \rightarrow$ with,
$O \rightarrow$ puppy, $O \rightarrow$ kitten, $O \rightarrow$ hill, $O \rightarrow$ wall,
$X \rightarrow$ petted, $X \rightarrow$ held.

This operation is similar to merging, except that it can be applied to *strings* of terminals and nonterminals. If both of s_1 and s_2 are simple nonterminal symbols, disjunction has the same effect as merging. If only one of s_1 or s_2 is a nonterminal, the operation is called *substitution*.

These rules of generalization can be applied to move from one point in the rule space (i.e., one grammar) to another. We now turn our attention to the four basic methods of grammatical inference and show how they apply these operations to search the space of possible context-free grammars.

Enumerative Methods

Enumerative methods generate grammars one by one and test each to determine how well it accounts for the training instances. The first enumerative method we consider is that of Horning (1969), who developed a procedure for finding the most plausible stochastic grammar consistent with a set of stochastically presented training instances. The general idea behind Horning's method is to enumerate all possible grammars in order of simplicity and choose the first grammar that is consistent with the training data. The actual algorithm is somewhat more complicated, however, since Horning seeks the *most likely* stochastic grammar, that is, the grammar G that is most likely to have generated the observed set S of sample strings. This is expressed formally as the grammar G that maximizes $P(G \mid S)$, that is, the probability of G given S. Unfortunately, it is difficult to compute $P(G \mid S)$ directly from the training instances. Bayes' theorem, however, provides a way of computing $P(G \mid S)$ from three other quantities, $P(G)$, $P(S \mid G)$, and $P(S)$:

$$P(G \mid S) = \frac{P(G) \times P(S \mid G)}{P(S)},$$

where $P(G)$ is the a priori probability that G is the "true" grammar, $P(S)$ is the a priori probability of observing the particular sample S, and $P(S \mid G)$ is the probability of observing S given the grammar G. Since $P(S)$ is independent of G, we can maximize $P(G \mid S)$ by just maximizing the numerator $P'(G \mid S) = P(G) \times P(S \mid G)$. The probabilities $P(G)$ and $P(S \mid G)$ can be computed for any particular grammar G.

The probability $P(S \mid G)$ that the training instances S will be generated by the stochastic grammar G can be computed directly from G by parsing each sentence in S. The problem of computing $P(G)$ is more difficult, however. Horning sought to have the a priori probability of G reflect the complexity of the grammar G. Simple grammars should be highly probable; complex grammars should be improbable. Consequently, he developed the idea of a grammar-grammar, that is, a stochastic grammar that generates a *stochastic grammar* as its terminal string. Such a grammar-grammar can be constructed from a terminal vocabulary of symbols such as A, B, C, Z, $\rightarrow$, etc. Since, as we have seen above, a stochastic grammar generates short strings with a much higher probability than it does long strings, the grammar-grammar generates simple grammars with a much higher probability than it does complex ones. In particular, the probability $P(G)$ is the probability that the grammar-grammar would generate G.

Since we can compute $P(G)$ and $P(S \mid G)$, we can use Bayes' theorem to compute $P'(G \mid S)$. Therefore, if we compute $P'(G \mid S)$ for all possible grammars, G, we can find the grammar that most likely generated S. Such a procedure is impossibly inefficient, however. Instead, Horning used the following technique. First, he developed a procedure that could enumerate all possible stochastic grammars starting with the most likely grammar, G_1, and continuing on in order of decreasing probability $P(G_i)$. Next, he noticed that $P'(G_i \mid S)$ did not have to be computed for all grammars but only for those grammars whose probability $P(G_i)$ was greater than $P'(G_1 \mid S)$. This is because once $P(G_i)$ falls below $P'(G_1 \mid S)$, there is no way that multiplying by $P(S \mid G_i)$ will ever exceed $P'(G_1 \mid S)$, since $P(S \mid G_i)$ is always less than or equal to 1.

Consequently, Horning's method enumerates all grammars G_i starting with G_1 and continuing until $P(G_i) < P'(G_1 \mid S)$. The probability $P'(G_i \mid S)$ is computed for each grammar G_i, and the grammar that maximizes $P'(G_i \mid S)$ is output as the grammar most likely to have produced the set of examples, S.

The algorithm is theoretically correct—it always finds the best grammar—but it is still too inefficient for all but the smallest grammars. Therefore, Horning modified the grammar generator to generate only grammars that were *deductively acceptable* (DA). A grammar is deductively acceptable if it generates every string in the sample, S, and if every production in G is used to derive at least one of the training instances. In other words, a DA grammar must be consistent with the training instances and must not be overly specific or cluttered by useless productions. It can be shown that all DA grammars with $k + 1$ nonterminals can be obtained by splitting DA grammars with k nonterminals. Furthermore, once a grammar ceases to be deductively acceptable, no further splits will make it deductively acceptable, since it is already overly specific.

These facts were used by Horning to organize the rule-space search. Starting with the most general (and most likely) DA grammars, repeated splits

are made until either the grammars cease to be deductively acceptable or their a priori probability $P(G_i)$ falls below the bound $P'(G_1 \mid S)$. The probability $P'(G_i \mid S)$ is computed for all of the generated grammars, and the grammar that maximizes $P'(G_i \mid S)$ is selected. This procedure, although more efficient than the first one, is still of theoretical interest only.

A second enumerative method makes use of training instances to guide the enumeration of plausible grammars. Pao (1969) describes an approach to grammatical inference that resembles the plan-generate-test paradigm of the DENDRAL program (see Sec. VII.C2, in Vol. II). In the initial planning phase, Pao's algorithm analyzes the (positive) training instances and constructs a trivial grammar—that is, a very specific grammar that generates only the training examples. A partially ordered set (actually, a lattice) of plausible grammars can be generated by merging nonterminals from this trivial grammar. During the generate-and-test phase, Pao's algorithm enumerates all of these grammars in order, from most specific to most general, and tests them by consulting an informant.

Pao's algorithm generates two grammars at a time, G and H, and uses an informant to eliminate one of the two. The informant is presented with a new sentence, s, that is generated by G but not by H. If the informant says s is in the "true" language, then H and all grammars more specific than H are removed from further consideration. Also, the set of grammars more general than H (but not more general than G) is searched in order from general to specific, and grammars that do not generate s are discarded. If, on the other hand, the informant says that s is not in the "true" language, then G and all grammars more general than G are removed from further consideration. The generating and testing of possible grammars continues until only one possible grammar remains. This search through the partially ordered set of all possible grammars is similar to Mitchell's (1978) candidate-elimination algorithm (see Article XIV.D3a). In Pao's program, though, an active experimentation approach is employed to search the space rather than waiting for new training instances to drive the search.

Unfortunately, this method does not work for general context-free grammars. The basic algorithm works only for regular grammars—that is, grammars whose productions all have the form $N \rightarrow tM$ or $N \rightarrow t$ for t, a single terminal symbol, and M, a single nonterminal symbol. In regular languages, there is no difficulty finding a test sentence s to distinguish between two grammars G and H. Unfortunately, this cannot be done for general context-free languages. Pao has extended the method to handle delimited grammars—a somewhat larger class of grammars than the regular grammars.

Constructive Methods

Constructive methods attempt to build a plausible grammar using only the information from a positive sample with no informant. From Gold's

theorems, it is clear that this problem is ill-formed, since no unique language is determined by a set of positive instances. However, various heuristics have been developed for constructing simple, fairly general grammars from positive instances only.

One important set of heuristics is based on the idea of the *distribution* of substrings in the language. In context-free languages, certain classes of strings, such as noun phrases and prepositional phrases, tend to appear in the same contexts in different sentences. This suggests that we might be able to discover interesting classes of strings by looking at their surroundings in the set of sample sentences. For instance, the words *a* and *the* both tend to occur at the beginnings of sentences, so perhaps they should be grouped together to form the class of *articles*. This is done by creating a nonterminal A and inventing the production rules "$A \rightarrow$ a" and "$A \rightarrow$ the." Distributional analysis has been employed by Harris (1964), Fu (1975), Kelley (1967), and Klein and Kuppin (1970)

For regular grammars, Fu (1975) has applied a particular kind of distributional analysis based on the idea of the *formal derivative* of a string. The formal derivative of a string s is the set of strings

$$D_s L = \{t \mid \text{the string } st \text{ is in the language } L\},$$

that is, all of the strings t that follow s in the given language L in sentences where s is at the beginning of the sentence.

Formal derivatives can be employed to construct regular grammars in a straightforward way. Imagine that we have a grammar G, and we are in the process of generating a sentence. Suppose that, so far, we have generated the string sU, where U is a nonterminal and s is a terminal string. If we take formal derivatives for every string sa that appears in the sample (where a is a single terminal symbol), we can create new nonterminals for each distinct formal derivative. We can add the productions

$$\begin{aligned} U &\rightarrow aV_1 \\ U &\rightarrow bV_2 \\ &\vdots \\ U &\rightarrow mV_k \end{aligned}$$

to the grammar, G, where $V_1, V_2, \ldots, V_k$ correspond to the formal derivatives of $sa, sb, \ldots, sm$. The effect of this construction is to group together all of the strings in the formal derivative of sa, for example, and place them in the sublanguage for V_1. We can construct the entire grammar G by initially taking s to be the null string and U to be the start symbol.

The chief difficulty of distributional methods is that some definition of *similar contexts* is needed so that strings that appear in similar contexts can be grouped into the sublanguage for a new nonterminal symbol. Problems

can also arise when one string is in two different sublanguages and therefore appears in different contexts. The word *program*, for example, can be both a noun and a verb.

Another approach to constructive inference of grammars is to look for repetition in the sample and model it as a recursive production. This method is rarely sufficient in itself to construct the whole grammar, but it can be used in combination with other methods. Consider, for example, the set of training instances $\{a, aaa, aaaa\}$. A reasonable grammar to infer has the productions $S \rightarrow a$ and $S \rightarrow Sa$ and generates all possible strings of repeated as.

To employ this repetition heuristic, it is helpful to know the properties of repetition for different kinds of grammars. For regular grammars, iteration always takes the form of repeated choice of a string without reference to any other strings. However, for context-free languages, repetition can be more complicated. One important theorem about context-free languages (called the $uvxyz$ theorem) states that if a sufficiently long string $uvxyz$ is in the language, then so is the string uv^kxy^kz as well; that is, v and y are repeated an equal number of times. This can be represented by a *self-embedding* production of the form $X \rightarrow VXY$. Solomonoff (1964) and Maryanski (1974) describe inference methods based on searching for double cycles of the uv^kxy^kz variety. Once a possible cycle is found, it can be tested by consulting an informant.

Refinement Methods

Refinement methods formulate a hypothesis grammar and then refine it by applying simplification heuristics or by gathering new training instances. Knobe and Knobe (1977), for example, present an algorithm that creates an initial hypothesis grammar, G, and then enters a refinement cycle in which it repeatedly accepts a new grammatical string, refines G to include the string, and generalizes and simplifies G. The initial grammar includes a distinct nonterminal for each of the terminal symbols. In the course of the algorithm, these nonterminals are generalized by merging. The basic learning cycle proceeds as follows:

Step 1. Accept a grammatical string (i.e., a positive training instance) and attempt to parse the string with the current grammar, G. If the parse succeeds, repeat step 1; otherwise, go to step 2.

Step 2. Compute a list of partial parses and sort it according to generality. (A *partial parse* is a string of terminals and nonterminals in which parts of the original training string have been partly parsed into nonterminals; the more general partial parses are shorter, since most of the sentence has been successfully parsed.) Hypothesize the production $S \rightarrow P$, where S is the start symbol and P is the most general partial parse. (This allows the training instance to be parsed successfully.) Use the modified grammar to generate a test

sentence, and ask the informant if the test sentence is grammatical. If it is, go to step 3; otherwise, try the next most general partial parse, and repeat until a sufficiently specific production has been found.

Step 3. Generalise and simplify the grammar by applying some of the merging and substitution heuristics described below.

The third step of generalization and simplification is important, because it is in this step that the new production $S \rightarrow P$ is integrated into the grammar and connected to existing production rules. Many different simplification and generalization techniques have been developed by various researchers. We survey a number of these here.

Generalization by disjunction. One important simplification technique is to apply disjunction (see above) to replace two similar strings s and t, which appear on the right-hand sides of productions, by a single nonterminal. There are two basic heuristics for deciding whether s and t are similar: *internal similarity* and *external similarity*. The internal-similarity heuristic compares the sublanguages generated by s and t. If the sublanguages are similar, the heuristic proposes that s and t are similar and should be disjoined. The external-similarity heuristic, on the other hand, compares the contexts in which s and t appear. As in the constructive technique of distributional analysis, if s and t appear in similar contexts, the heuristic recommends that they be disjoined. There are many important special cases of these heuristics:

1. *Heuristics based on internal similarity.* The first internal-similarity heuristic is *subsumption*. If the language generated by s is a superset of the language generated by t, then s and t should be disjoined. This often occurs when s is a single nonterminal, X, and the rule $X \rightarrow t$ is among the productions for X in the grammar.

   If s and t are both single nonterminals, X and Y, a second internal heuristic can be applied. This heuristic compares the right-hand sides, u and v, of production rules of the form $X \rightarrow u$ and $Y \rightarrow v$, to see if they are similar. If they are, X and Y can be merged.

   A third internal-similarity heuristic is *k-tail equivalence*. Two strings s and t are *k-tail equivalent*, for some nonnegative integer k, if the sets of strings of length k or less that they generate are the same. Thus, s and t are judged similar if the short strings that they generate are the same. This heuristic can be applied by choosing a value for k and merging groups of nonterminals that are k-tail equivalent. As k gets small, this heuristic causes more generalization.

2. *Heuristics based on external similarity.* The one heuristic for external similarity is to look at productions in which s and t appear on the right-hand side of productions. If s and t appear in similar contexts within the productions, they can be disjoined. Various special cases of this heuristic have been used, including the case in which s and t are both single nonterminals.

Hypothesizing iteration. As with constructive methods, if productions such as $X \rightarrow a$ and $X \rightarrow aa$ are present, a recursive production $X \rightarrow Xa$ can be introduced.

Shorthand substitution. When a string s appears many times on the right-hand side of productions, it is often good to create a new nonterminal, A, replace all occurrences of s by A, and add the production $A \rightarrow s$ to the grammar. This simplifies the grammar without modifying the language that it generates. The advantage of the simplification is that it is easier to apply the various merging heuristics to a simplified grammar.

The k-tail heuristic was employed by Biermann and Feldman (1970) in the inference of regular grammars. Various of the other heuristics are employed by Klein and Kuppin (1970), Evans (1971), Knobe and Knobe (1977), and Cook and Rosenfeld (1976). Cook and Rosenfeld are concerned with stochastic grammars and use their heuristics to simplify grammars with a hill-climbing procedure based on a numerical-complexity measure.

Semantics-based Methods

The fourth basic approach to grammatical inference employs semantic constraints to guide the search for plausible grammars. Most of this work has centered on language acquisition by children. The child is given positive examples of sentences and is assumed to know the meanings of individual words in isolation. Furthermore, the situation in which the sentence was uttered, and, thus, some idea about its overall meaning, is assumed to be known by the child. In most work, no negative examples are provided, nor is an informant available. This is because most research in psychology (e.g., Brown and Hanlon, 1970) has found that children receive little or no feedback concerning the grammaticality of the sentences they utter. Pinker (1979) discusses the work of several researchers who have studied grammatical inference under these assumptions, including Anderson (1977) and Hamburger and Wexler (1975).

Anderson's Language Acquisition System (LAS) attempts to learn a context-free grammar for English from training instances that include a representation of the meaning of each sentence. The Human Associative Memory (HAM; Article XI.E2) network notation is used to represent these sentence meanings. Learning proceeds in a cycle similar to that of Knobe and Knobe (1977): A sentence and its meaning are input, and LAS attempts to parse the sentence. If the parse fails, the grammar is extended according to some refinement heuristics so that the training sentence can be parsed and assigned the correct meaning. One such heuristic adds a word to a sublanguage—for example, it adds *chair* to the sublanguage for (noun)—when the word is located at a place in the HAM net similar to the place of other words in the sublanguage. This is a special case of the general heuristic that the structure of the semantic representation is reflected in the structure of the syntax of the language. A

more sophisticated version of this heuristic is the *graph deformation condition*, which states that branches in the HAM represention of the sample sentence are not allowed to cross. This heuristic rules out certain parses that would result in an ill-formed HAM structure. Anderson also employs one syntactic heuristic: Two nonterminals are merged if they have similar sublanguages.

The work of Hamburger and Wexler (1975) is more theoretical in nature and is concerned with showing that transformational grammars (see Chomsky, 1965) are learnable. In their model, the learner is repeatedly given a sentence and its meaning, where the meaning is represented as a deep-structure parse tree (based on a deep-structure context-free grammar). The learner must find a set of transformation rules that succeed, for each sample sentence, in converting the deep structure into the given sentence. Hamburger and Wexler are proponents of Chomsky's nativist theory of language acquisition, which asserts that people have built-in limits and biases that provide essential constraints for the language-learning process. Consequently, their model of language learning includes several factors that limit the complexity of possible transformations.

Given these limits, Hamburger and Wexler show that the desired set of transformations can be learned by a program as follows. As each training instance (a sentence and its deep structure) is received, the learner tries to transform the deep structure into the surface sentence by applying its current set of transformations. If this succeeds, the learner goes on to the next input example. If not, the learner randomly adds, deletes, or alters a transformation and goes on. This method will work as long as the learner does not repeat transformation rules known to be incorrect. Plainly, this learning procedure is not practical, but it does demonstrate that learning transformation rules under these assumptions is possible.

Conclusion

The expressiveness of grammars for use in AI knowledge representation is somewhat limited, so interest in the difficult problem of grammatical inference is also correspondingly limited in the AI community. This is especially so because of the impractical nature of many of the grammatical-inference systems developed thus far. However, future work on the problem may yield more powerful inference systems, and an understanding of past work may well be helpful in research on related learning problems.

References

We have surveyed here the motivations, limitations, and methods of grammatical inference. More detailed surveys of grammatical inference in the context of cognitive psychology are given in Pinker (1979) and Reeker (1976).

Surveys of grammatical inference for use in syntactic pattern recognition are given in Fu (1974, 1975), Biermann and Feldman (1972), and Gonzalez and Thompson (1978).

STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

DEPARTMENT OF COMPUTER SCIENCE

Telephone
415 497-2273

December 3, 1982

Mr. Cundiss
Defense Technical Information Center
DDA
Cameron Station
Alexandria, VA 22314

Dear Mr. Cundiss:

In response to our telephone conversation on Thursday, December 2, 1982, I spoke with one of the authors of "Learning and Inductive Inference." He assured me that there were no missing pages in the reports you received. Apparently, the sequences of pages you reported as missing were intentionally deleted from the text by the authors. Unfortunately, the pages in the report were not subsequently renumbered, and so some confusion has ensued.

I have included two copies of this report. The author assured me that they are both complete. I am sorry for the inconvenience.

Please do not hesitate to call me in the future if difficulties arise. Thank you for your interest in CS Reports.

Sincerely,

Kathryn Berg
Publications Coordinator

RE: Pages deleted by authors
513 thru 564, 572 thru 588,
591 thru 600.

BIBLIOGRAPHY

Abbott, R. 1977. The new Eleusis. Available from author: Box 1175, General Post Office, New York, NY 10116.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. 1974. *The design and analysis of computer algorithms.* Reading, Mass.: Addison-Wesley.

Anderson, J. R. 1977. Induction of augmented transition networks. *Cognitive Science* 1:125-157.

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

Barr, A., Bennett, J., and Clancey, W. 1979. Transfer of expertise: A theme for AI research. Rep. No. HPP 79 11, Heuristic Programming Project, Stanford University.

Biermann, A., and Feldman, J. 1970. On the synthesis of finite-state acceptors. AI Memo 114, Computer Science Dept., Stanford University.

Biermann, A., and Feldman, J. 1972. A survey of results in grammatical inference. In S. Watanabe (Ed.), *Frontiers of pattern recognition.* New York: Academic Press.

Brown, R., and Hanlon, C. 1970. Derivational complexity and order of acquisition in child speech. In J. Hayes (Ed.), *Cognition and the development of language.* New York: Wiley, 11-53.

Buchanan, B. G., and Mitchell, T. M. 1978. Model-directed learning of production rules. In D. A. Waterman and F. Hayes-Roth (Eds.), *Pattern-directed inference systems.* New York: Academic Press, 297-312.

Buchanan, B. G., Mitchell, T. M., Smith, R. G., and Johnson, C. R., Jr. 1977. Models of learning systems. In J. Belzer, A. G. Holzman, and A. Kent (Eds.), *Encyclopedia of computer science and technology* (Vol. 11). New York: Marcel Dekker, 24 51.

Carnap, R. 1950. *Logical foundations of probability.* Chicago: University of Chicago Press.

Chomsky, N. 1957. *Syntactic structures.* The Hague: Mouton.

Chomsky, N. 1965. *Aspects of the theory of syntax.* Cambridge, Mass.: MIT Press.

Cook, C. M., and Rosenfeld, A. 1976. Some experiments in grammatical inference. In J. C. Simon (Ed.), *Proceedings of the NATO Advanced Study Institute on Computer Oriented Learning Processes.* Leyden, The Netherlands: Noordhoff.

Date, C. J. 1977. *An introduction to database systems* (2nd ed.). Reading, Mass.: Addison-Wesley.

Davis, R. 1976. Applications of meta level knowledge to the construction, maintenance, and use of large knowledge bases. Rep. No. STAN-CS-76-564, Computer Science Dept., Stanford University. (Doctoral dissertation. Reprinted in R. Davis and D. B. Lenat (Eds.). 1980. *Knowledge based systems in artificial intelligence.* New York: McGraw-Hill.)

Davis, R. 1978. Knowledge acquisition in rule-based systems: Knowledge about representations as a basis for system construction and maintenance. In D. A.

Waterman and F. Hayes-Roth (Eds.), *Pattern-directed inference systems.* New York: Academic Press, 99–134.

Dietterich, T. G. 1979. The methodology of knowledge layers for inducing descriptions of sequentially ordered events. Rep. No. UIUC-DCS 80 1024, Computer Science Dept., University of Illinois, Urbana.

Dietterich, T. G. 1980. Applying general induction methods to the card game Eleusis. *AAAI 1,* 218–220.

Dietterich, T. G., and Michalski, R. S. 1979. Learning and generalization of characteristic descriptions: Evaluation criteria and comparative review of selected methods. *IJCAI 6,* 223–231.

Dietterich, T. G., and Michalski, R. S. 1981. Inductive learning of structural descriptions: Evaluation criteria and comparative review of selected methods. *Artificial Intelligence* 16:257–294.

Dietterich, T. G., and Michalski, R. S. In press. Discovering sequence generating rules.

Doyle, J. 1980. A model for deliberation, action, and introspection. Tech. Rep. AI-TR-581, AI Laboratory, Massachusetts Institute of Technology. (Doctoral dissertation.)

Duda, R. O., and Hart, P. E. 1973. *Pattern classification and scene analysis.* New York: Wiley.

Evans, T. G. 1971. Grammatical inference techniques in pattern analysis. In J. T. Tou (Ed.), *Software engineering* (Vol. 2). New York: Academic Press, 183–202.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. 1972. Learning and executing generalised robot plans. *Artificial Intelligence* 3:251–288.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2:189–208.

Fogel, L. J., Owens, A. J., and Walsh, M. J. 1966. *Artificial intelligence through simulated evolution.* New York: Wiley.

Friedberg, R. M. 1958. A learning machine: Part I. *IBM J. Research and Development* 2:2–13.

Friedberg, R. M., Dunham, B., and North, J. H. 1959. A learning machine: Part II. *IBM J. Research and Development* 3:282–287.

Fu, K. S. 1970a. Statistical pattern recognition. In J. M. Mendel and K. S. Fu (Eds.), *Adaptive, learning, and pattern recognition systems.* New York: Academic Press, 35–80.

Fu, K. S. 1970b. Stochastic automata as models of learning systems. In J. M. Mendel and K. S. Fu (Eds.), *Adaptive, learning, and pattern recognition systems.* New York: Academic Press, 393–432.

Fu, K. S. 1974. *Syntactic methods in pattern recognition.* New York: Academic Press.

Fu, K. S. 1975. Grammatical inference: Introduction and survey. *IEEE Transactions on Systems, Man, and Cybernetics* SMC-5:95–111, 409–423.

Gardner, M. 1977. On playing the new Eleusis, the game that simulates the search for truth. *Scientific American* 237:18–25.

Gelernter, H. 1959. Realization of a geometry theorem-proving machine. *Proceedings of an International Conference on Information Processing.* Paris: UNESCO House, 273–282.

Gelernter, H. 1963. Realization of a geometry theorem proving machine. In E. A. Feigenbaum and J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill, 134–152.

Gold, E. 1967. Language identification in the limit. *Information and Control* 16:447–474.

Gonsales, R. C., and Thompson, M. G. 1978. *Syntactic pattern recognition*. Reading, Mass.: Addison-Wesley.

Goodwin, G. C., and Payne, R. L. 1977. *Dynamic system identification: Experiment design and analysis*. New York: Academic Press.

Greiner, R. 1980. RLL 1: A representation language language. Rep. No. HPP-80-9. Heuristic Programming Project, Computer Science Dept., Stanford University.

Greiner, R., and Lenat, D. B. 1980. A representation language language. *AAAI 1*, 165–169.

Hamburger, H., and Wexler, K. 1975. A mathematical theory of learning transformational grammar. *J. Mathematical Psychology* 12:137–177.

Harris, Z. 1964. Distributional structure. In J. Fodor and J. Katz (Eds.), *The structure of language*. Englewood Cliffs: Prentice Hall, 33–49.

Hayes-Roth, F., Klahr, P., Burge, J., and Mostow, D. 1978. Machine methods for acquiring, learning, and applying knowledge. Rand Paper P-6241, Rand Corp., Santa Monica, Calif.

Hayes-Roth, F., Klahr, P., and Mostow, D. 1980. Knowledge acquisition, knowledge programming, and knowledge refinement. Rand Paper R-2540-NSF, Rand Corp., Santa Monica, Calif.

Hayes-Roth, F., Klahr, P., and Mostow, D. 1981. Advice-taking and knowledge refinement: An iterative view of skill acquisition. In J. R. Anderson (Ed.), *Cognitive skills and their acquisition*. Hillsdale, N.J.: Lawrence Erlbaum, 231–253. (Also in Rand Paper P-6517, Rand Corp., Santa Monica, Calif., 1980.)

Hayes-Roth, F., and McDermott, J. 1977. Knowledge acquisition from structural descriptions. *IJCAI 5*, 356–362.

Hayes-Roth, F., and McDermott, J. 1978. An interference matching technique for inducing abstractions. *CACM* 26:401–410.

Hopcroft, J. E., and Ullman, J. D. 1969. *Formal languages and their relation to automata*. Reading, Mass.: Addison-Wesley.

Horning, J. J. 1969. A study of grammatical inference. Rep. No. CS-139, Computer Science Dept., Stanford University.

Hunt, E. B., Marin, J., and Stone, P. J. 1966. *Experiments in induction*. New York: Academic Press.

Kelley, K. 1967. Early syntax acquisition. Rep. No. P-3719, Rand Corp., Santa Monica, Calif.

Klein, S., and Kuppin, M. 1970. An interactive heuristic program for learning transformational grammars. *Computer Studies in the Humanities and Verbal Behavior* 3:144–162.

Knobe, B., and Knobe, K. 1977. A method for inferring context-free grammars. *Information and Control* 31:129–146.

Kotovsky, K., and Simon, H. A. 1973. Empirical tests of a theory of human acquisition of concepts for sequential patterns. *Cognitive Psychology* 4:399–424.

Langley, P. W. 1977. Rediscovering physics with BACON.3. *IJCAI 6,* 505–507.

Langley, P. W. 1980. Descriptive discovery processes: Experiments in Baconian science. Rep. No. CS 80 121, Computer Science Dept., Carnegie-Mellon University. (Doctoral dissertation.)

Larson, J. 1977. Inductive inference in the variable valued predicate logic system VL21: Methodology and computer implementation. Rep. No. 869, Computer Science Dept., University of Illinois, Urbana.

Larson, J., and Michalski, R. S. 1977. Inductive inference of VL decision rules. *SIGART Newsletter* 63:38–44.

Lenat, D. B. 1976. AM: An artificial intelligence approach to discovery in mathematics as heuristic search. Rep. No. STAN CS 76 570. Computer Science Dept., Stanford University. (Doctoral dissertation. Reprinted in R. Davis and D. B. Lenat. 1980. *Knowledge-based systems in artificial intelligence.* New York: McGraw-Hill.)

Lenat, D. B. 1977. On automated scientific theory formation: A case study using the AM program. In J. E. Hayes, D. Michie, and L. I. Mikulich (Eds.), *Machine intelligence 9.* New York: Halsted Press, 251–286.

Lenat, D. B. 1980. The nature of heuristics. Rep. No. HPP-80-26. Heuristic Programming Project, Computer Science Dept., Stanford University.

Lenat, D. B., Hayes-Roth, F., and Klahr, P. 1979. Cognitive economy in artificial intelligence systems. *IJCAI 6,* 531–536. (Extended version available as Rep. No. HPP–79–15, Heuristic Programming Project, Computer Science Dept., Stanford University.)

Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., and Lederberg, J. 1980. *Applications of artificial intelligence for organic chemistry: The DENDRAL project.* New York: McGraw-Hill.

Maryanski, F. J. 1974. *Inference of probabilistic grammars.* Doctoral dissertation, Electrical Engineering and Computer Science Dept., University of Connecticut.

McCarthy, J. 1958. Programs with common sense. In *Proceedings of the Symposium on the Mechanization of Thought Processes, National Physical Laboratory* 1:77–84. (Reprinted in M. L. Minsky (Ed.). 1968. *Semantic information processing.* Cambridge, Mass.: MIT Press, 403–409.)

McCarthy, J. 1968. Programs with common sense. In M. Minsky (Ed.), *Semantic information processing.* Cambridge, Mass.: MIT Press, 403–409.

Michalski, R. S. 1969. On the quasi-minimal solution of the general covering problem. *Proceedings of the Fifth International Federation on Automatic Control* 27:109–129.

Michalski, R. S. 1975. Variable-valued logic and its applications to pattern recognition and machine learning. In D. C. Rine (Ed.), *Computer science and multiple-valued logic theory and applications.* Amsterdam: North-Holland, 506–534.

Michalski, R. S. 1980. Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-2:349–361.

Michalski, R. S., and Chilausky, R. L. 1980. Learning by being told and learning from examples: An experimental comparison of the two methods of knowledge acquisition in the context of developing an expert system for soybean disease diagnosis. *International Journal of Policy Analysis and Information Systems* 4:125–161.

Michalski, R. S., and Larson, J. B. 1978. Selection of most representative training examples and incremental generation of VL1 hypotheses: The underlying methodology and the description of programs ESEL and AQ11. Rep. No. 867. Computer Science Dept., University of Illinois, Urbana.

Minsky, M. 1963. Steps toward artificial intelligence. In E. A. Feigenbaum, and J. Feldman (Eds.), *Computers and thought*. New York: McGraw-Hill, 406–450.

Minsky, M. L. (Ed.). 1968. *Semantic information processing*. Cambridge, Mass.: MIT Press.

Minsky, M. L., and Papert, S. 1969. *Perceptrons; an introduction to computational geometry*. Cambridge, Mass.: MIT Press.

Mitchell, T. M. 1977. Version spaces: A candidate elmination approach to rule learning. *IJCAI 5*, 305–310.

Mitchell, T. M. 1978. Version spaces: An approach to concept learning. Rep. No. STAN-CS-78-711, Computer Science Dept., Stanford University. (Doctoral dissertation.)

Mitchell, T. M. 1979. An analysis of generalization as a search problem. *IJCAI 6*, 577–582.

Mitchell, T. M., Utgoff, P. E., and Banerji, R. B. In press. Learning problem-solving heuristics by experimentation. In R. S. Michalski, T. M Mitchell, and J. Carbonell (Eds.), *Machine learning*. Palo Alto, Calif.: Tioga.

Mitchell, T. M., Utgoff, P. E., Nudel, B., and Banerji, R. B. 1981. Learning problem-solving heuristics through practice. *IJCAI 7*, 127–134.

Mostow, D. J. 1981. Mechanical transformation of task heuristics into operational procedures. Rep. No. CS-81-113, Computer Science Dept., Carnegie-Mellon University. (Doctoral dissertation.)

Mostow, D. J. In press. Using the heuristic search method. In R. S. Michalski, T. M. Mitchell, and J. Carbonell (Eds.), *Machine learning*. Palo Alto, Calif.: Tioga.

Mostow, D. J., and Hayes-Roth, F. 1979a. Machine-aided heuristic programming: A paradigm for knowledge engineering. Rep. No. Rand N-1007-NSF, Rand Corp., Santa Monica, Calif.

Mostow, D. J., and Hayes-Roth, F. 1979b. Operationalizing heuristics: Some AI methods for assisting AI programming. *IJCAI 6*, 601–609.

Nii, H. P., and Aiello, N. 1979. AGE (Attempt to Generalize): A knowledge-based program for building knowledge-based programs. *IJCAI 6*, 645–655.

Nilsson, N. J. 1965. *Learning machines*. New York: McGraw-Hill.

Norman, D. A. 1980. Twelve issues for cognitive science. *Cognitive Science* 4:1–32.

Pao, T. W. 1969. A solution of the syntactical induction-inference problem for a non-trivial subset of context-free languages. Interim Rep. No. 69-19, Moore School of Electrical Engineering, University of Pennsylvania.

Pinker, S. 1979. Formal models of language learning. *Cognition* 7:217–283.

Quinlan, J. R. 1979. Induction over large data bases. Rep. No. HPP-79-14, Heuristic Programming Project, Computer Science Dept., Stanford University.

Quinlan, J. R. In press. Inductive inference as a tool for the construction of high-performance programs. In R. S. Michalski, T. M. Mitchell, and J. Carbonell (Eds.), *Machine learning*. Palo Alto, Calif.: Tioga.

Reboh, R. 1981. Knowledge engineering techniques and tools in the PROSPECTOR environment. Rep. No. 243. AI Center, SRI International, Inc., Menlo Park, Calif.

Reeker, L. H. 1976. The computational study of language acquisition. In M. Rubinoff and M. C. Yovits (Eds.), *Advances in computers* (Vol. 15). New York: Academic Press, 181–237.

Rissland, E. L., and Soloway, E. M. 1980. Overview of an example generation system. *AAAI 1*, 256–258.

Rosenblatt, F. 1957. The perceptron: A perceiving and recognizing automaton. Rep. No. 85-460-1, Project PARA, Cornell Aeronautical Laboratory.

Rosenblatt, F. 1962. *Principles of neurodynamics: Perceptrons and the theory of brain mechanisms*. Washington, D.C.: Spartan Books.

Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM J. Research and Development* 3:210–229. (Reprinted in E. A. Feigenbaum, and J. Feldman (Eds.). 1963. *Computers and thought*. New York: McGraw-Hill, 71–105.)

Samuel, A. L. 1967. Some studies in machine learning using the game of checkers. II – Recent progress. *IBM J. Research and Development* 11:601–617.

Shortliffe, E. H. 1976. *Computer-based medical consultations: MYCIN*. New York: American Elsevier.

Simon, H. A. 1979a. Artificial intelligence research strategies in the light of AI models of scientific discovery. *IJCAI 6*, 1086–1094.

Simon, H. A. In press. Why should machines learn? In R. S. Michalski, T. M. Mitchell, and J. Carbonell (Eds.), *Machine learning*. Palo Alto, Calif.: Tioga.

Simon, H. A., and Lea, G. 1974. Problem solving and rule induction: A unified view. In L. Gregg (Ed.), *Knowledge and cognition*. Hillsdale, N.J.: Lawrence Erlbaum, 105–127.

Solomonoff, R. 1964. A formal theory of inductive inference. *Information and Control* 7:1–22, 224–254.

Soloway, E. 1978. Learning = interpretation + generalization: A case study in knowledge-directed learning. Rep. No. COINS-TR-78-13, Computer and Information Sciences Dept., University of Massachusetts, Amherst. (Doctoral dissertation.)

Sussman, G. J. 1973. A computational model of skill acquisition. AI Tech. Rep. 297, AI Laboratory, Massachusetts Institute of Technology. (Doctoral dissertation.)

Sussman, G. J. 1975. *A computer model of skill acquisition*. New York: American Elsevier.

Tsypkin, Y. Z. (Z. J. Nikolic, Trans.). 1973. *Foundations of the theory of learning systems*. New York: Academic Press.

Ullman, J. D. 1980. *Principles of database systems*. Potomac, Md.: Computer Science Press.

van Melle, W. 1980. A domain-independent system that aids in constructing knowledge based consultation programs. Rep. No. 820, Computer Science Dept., Stanford University. (Doctoral dissertation.)

Vere, S. A. 1975. Induction of concepts in the predicate calculus. *IJCAI 4*, 281–287.

Vere, S. A. 1978. Inductive learning of relational productions. In D. A. Waterman and F. Hayes-Roth (Eds.), *Pattern-directed inference systems*. New York: Academic Press, 281 296.

Waterman, D. A. 1968. Machine learning of heuristics. Rep. No. STAN CS 68 118, Computer Science Dept., Stanford University. (Doctoral dissertation.)

Waterman, D. A. 1970. Generalization learning techniques for automating the learning of heuristics. *Artificial Intelligence* 1:121 170.

Wee, W. G., and Fu, K. S. 1969. A formulation of fuzzy automata and its application as a model of learning systems. *IEEE Transactions on System Science and Cybernetics* 5:215 223.

Widrow, B., and Hoff, M. E. 1960. Adaptive switching circuits. In *1960 IRE WESCON Convention Records* 4:96 104.

Wiederhold, G. 1977. *Database design*. New York: McGraw-Hill.

Winston, P. H. 1970. Learning structural descriptions from examples. Rep. No. TR 231, AI Laboratory, Massachusetts Institute of Technology. (Reprinted in P. H. Winston (Ed.). 1975. *The psychology of computer vision*. New York: McGraw-Hill, 157 209.)

Winston, P. H. (Ed.). 1975. *The psychology of computer vision*. New York: McGraw-Hill.

Yovits, M. C., Jacobi, G. T., and Goldstein, G. D. (Eds.). 1962. *Self-organizing systems 1962*. Washington, D.C.: Spartan Books.

Zadeh, L.A. 1979. Approximate reasoning based on fuzzy logic. *IJCAI 6*, 1004 1010.

NAME INDEX FOR CHAPTER XIV

SUBJECT INDEX FOR CHAPTER XIV