

LEARNING AND PROBLEM SOLVING

Tom M. Mitchell

Computers and Thought Lecture

Computer Science Department
Rutgers University
New Brunswick, NJ 08903

Abstract

One mark of intelligence is the ability to improve one's problem-solving abilities with experience. Modelling this kind of learning is an important goal for AI, both because it is bound to lead to general insights on the nature of intelligence, and because of its practical importance for developing high-performance problem solving programs. We discuss some recent progress in the area of learning problem-solving expertise, as well as some directions in which research in this area is headed.

1 Introduction

Probably everyone who has thought imaginatively about computers has wondered at one time or another how they might be organized to learn. Artificial Intelligence (AI) researchers are no exception, and Machine Learning has always been part of the focus of AI. Through the years the direction and intensity of research in machine learning have varied considerably, ranging from an interest in adaptive neural nets and perceptrons, to a search for general mechanisms for learning from examples, to a more recent interest in acquisition and revision of problem-solving strategies, and in knowledge-guided learning.

Today we are seeing a resurgence of interest in machine learning. One reason for this may be that the field of AI has reached a certain level of understanding of general problem solving frameworks, as well as a certain level of understanding of simple learning methods, and that it is simply a good time to try combining these. We now understand certain kinds of problem solving in terms of search, and some of the ways in which knowledge can be used to replace search. We are beginning to see ways in which programs can reason explicitly about their own structure, capabilities, and goals in order to control problem solving and to guide learning. We also understand certain methods by which programs can learn general concepts and rules from examples, and how to apply these methods to acquire specific kinds of knowledge.

A second reason for increased interest in learning may be more pragmatic: recent progress in the area of knowledge-based expert systems has shown that high-performance problem solving programs can be developed in specialized task areas, and that the power of these systems follows largely from the task-specific knowledge they are given. A major impediment to developing such systems lies in the difficulty of constructing large knowledge bases that are complete enough and correct enough to produce high levels of performance. With some expert systems now containing thousands of inference rules, the significance of this knowledge acquisition bottleneck has grown. This has led to the development of semi-automated aids for

knowledge acquisition (e.g., [Davis 81, Politakis 82]), and to experiments in automated learning of inference rules for expert systems (e.g., [Buchanan 78, Michalski 80]). As the field of Expert Systems matures through the coming decade, we can expect to see an increased interest in learning methods, and their application to knowledge acquisition for expert systems.

One interesting prospect is that as expert systems become available to large user communities, we may find ways of allowing these systems to learn through their normal interactions with their human users. For example, in expert systems applications providing *interactive* problem solving aids (e.g., an intelligent aid for VLSI circuit design which resides inside an interactive circuit editor, giving helpful advice where possible), we would like to develop methods by which the system can learn new knowledge (e.g., new circuit design tactics) by observing the actions of its human problem-solving partner. Such a system might better be called an *apprentice system* than an expert system—it would initially act as a helpful assistant in problem solving, while building up knowledge gleaned from its experience working with human expert problem solvers, until it acquired enough knowledge of the task area to become an expert itself. Given such a system that could be duplicated a thousandfold, and distributed to a large user community, the potential of this approach to knowledge acquisition seems difficult to overstate.

The main purpose of this paper is to review some recent progress in learning, especially in the area of acquiring problem solving expertise, and to try to see the directions in which the field is headed. We first trace some of the issues and progress in this field, by tracing the evolving design of a particular learning program, called LEX, which learns problem-solving heuristics. The evolution of the design of LEX, and an analysis of its current limitations, reflect the evolution of our own understanding of learning in problem solving. While this paper uses LEX to illustrate many of its points, the developments and insights that come out of this research effort have parallels in (and draw a great deal upon) many other research efforts in the field. Following the discussion of LEX, we consider some general lessons and research trends that follow from current work on learning problem solving expertise.

Such current work includes a number of research efforts that have focused on the acquisition of problem-solving expertise, both by humans and by computers. Research such as [Brazdil 78, Neves 78, Lenat 82, Langley 81] has led to computer programs that learn various aspects of problem-solving expertise in specific task areas. In addition, work by psychologists [Anderson 83, Larkin 79, Anzai 79] has studied the improvement of problem

solving performance in humans, and the difference between novice and expert approaches to problem solving in several task domains. Work such as [Amarel 71, Korf 80, Mostow 81, Simon 77] has focused on how problems are formulated and reformulated to reduce their complexity.

The remainder of this paper is organized as follows: Section 2 describes the LEX program, which learns problem solving heuristics for solving integral calculus problems. LEX illustrates how a program can learn useful heuristics for solving problems within a specific task domain, by proposing and solving practice problems in that domain. Its initial design shows how syntactic methods for generalizing from examples can form the basis for a system that learns useful heuristics, but it also shows some fundamental limitations of this approach to learning. Section 3 discusses a major change to the design of LEX--the introduction of a very different, goal-directed approach to learning heuristics. This approach leads to surprisingly powerful methods for forming justifiable generalizations from single training instances, and for altering the representation language in which learned heuristics are to be described. Section 4 then considers some general lessons and research trends that arise from a variety of recent research efforts on learning and problem solving.

II Learning Problem Solving Heuristics in LEX

LEX is designed to learn problem solving heuristics by experimentation. It begins with a heuristic search problem-solver, without the heuristics. That is, it is given a set of about 50 operators for solving problems in integral calculus, and it learns a set of heuristics which constitute a strategy for using these operators to solve problems. Whereas each of the given operators contains a set of preconditions characterizing problems to which that operator *can* validly be applied, learned heuristics must characterize the more restrictive subclass of problems to which the operator *should* be applied. Thus, LEX faces a problem very similar to that faced by freshmen calculus students: it is told the allowable integral transformations, and it must then develop a strategy for using these transformations to solve problems.

Below is a typical example of a heuristic that LEX has learned:

$$\int x \operatorname{trig}(x) dx \Rightarrow \text{Apply Integration by parts} \\ \text{with } u = x \\ \text{and } dv = \operatorname{trig}(x) dx$$

In English, this heuristic means that "IF the current integral is the product of x and some trigonometric function of x , THEN try using integration by parts, and bind the operator arguments u and dv to the indicated subexpressions". All of the heuristics learned by LEX are of this form, relating a general class of problem states to a recommended problem solving operator.

LEX is thus a program that learns general concepts (i.e., "the class of problems for which operator X is useful"), from specific training instances (i.e., examples of specific problems for which operator X was found to be useful or useless). These learned concepts become the left hand sides of the learned heuristics. The problem of learning general concepts from specific examples has received a great deal of attention by many researchers (e.g., [Winston 75, Hayes-Roth 74, Michalski 73, Vere 78, Mitchell 78]) over the past decade, and the initial approach to learning heuristics in LEX is based on such earlier work.

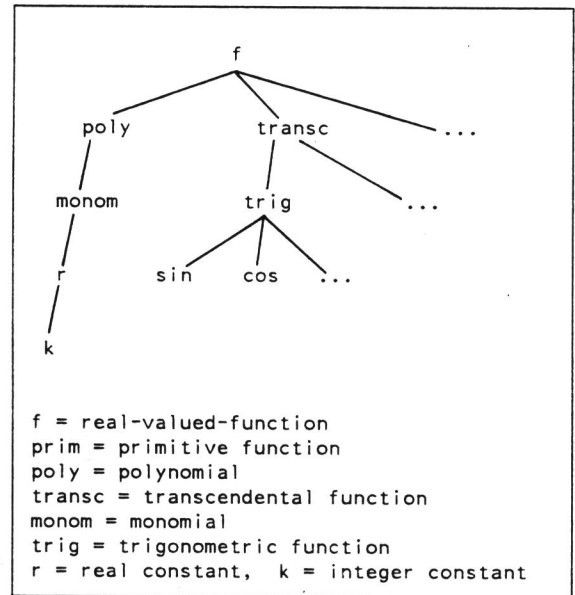


Figure II-1: A Portion of the Generalization Hierarchy

In order for a program to learn a heuristic, or any other kind of knowledge, it must have some representation language in which that knowledge can be stated. LEX describes the left hand sides of its learned heuristics in a language built up from primitive terms such as "polynomial", "integer", "real number", etc., which form a generalization hierarchy as shown in figure II-1. The choice of terms to include in this *generalization language* is crucial, since it is this language which determines the range of concepts that LEX can describe, and therefore learn. The strong dependence of learning capabilities on the initial choice of representation is a longstanding and fundamental problem in learning. Section 3 describes some recent progress on this problem, and a method by which LEX is able to add new terms, such as "odd integer", to its generalization language.

A. Initial Design of LEX

LEX learns heuristics by generating and solving practice problems, then analyzing the resulting solution trace. Its design is based on four distinct modules, corresponding roughly to the four major components of learning systems suggested in [Smith 77]. These modules are shown in figure II-2, summarized below, and described more fully in the following subsections.

1. **The Problem Solver** utilizes whatever operators and heuristics are currently available, to attempt to solve a given practice problem within allocated time and space resources. Its output is a solution to the given problem, along with a detailed trace of the search performed in attempting to solve the problem.
2. **The Critic** analyzes the search tree generated by the Problem Solver, to produce a set of positive and negative training instances from which heuristics will be inferred. Positive instances correspond to search steps along the best solution path, and negative instances correspond to search steps leading away from this solution path.

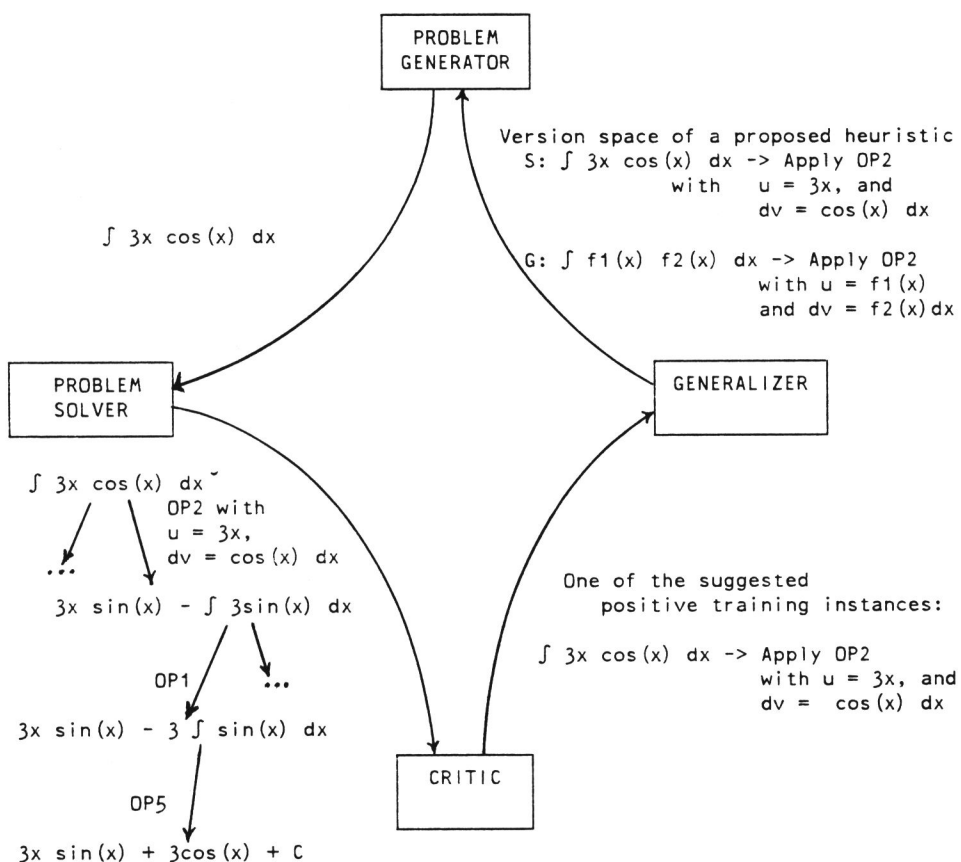


Figure II-2: The Learning Cycle in LEX.

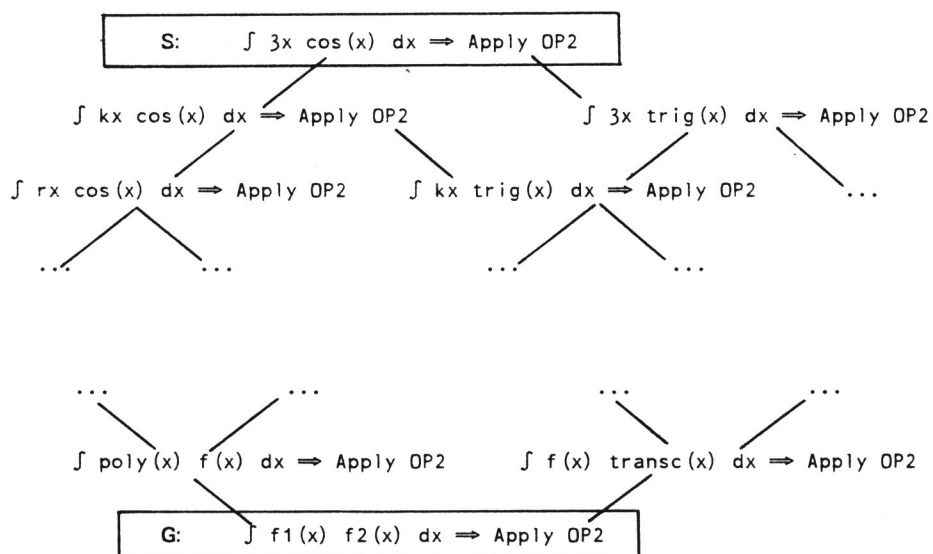


Figure II-3: Representing a Version Space.

3. **The Generalizer** proposes and refines general heuristics intended to produce more effective problem solving behavior on subsequent problems. It formulates heuristics by generalizing from the training instances provided by the Critic.
4. **The Problem Generator** generates the next practice problem for the system. It must generate practice problems that will be informative (i.e., they will lead to training data useful for proposing and refining heuristics), yet easy enough to be solved using existing heuristics.

These modules work together to propose and incrementally refine heuristics, from training instances collected over several propose-solve-criticize-generalize learning cycles. Any system that learns concepts incrementally must have some way of characterizing its partial knowledge of those concepts that are not yet completely formulated, and some way of determining when a given concept is fully learned. By an "incompletely learned concept", we mean one for which the available training information does not yet fully determine the correct description of the concept. For LEX, the ability to characterize what it does *not* know about incompletely learned heuristics, is important (a) to the Problem Solver, which must use these incompletely learned heuristics in trying to solve problems, (b) to the Generalizer, which must revise partially learned heuristics as new training information becomes available, and (c) to the Problem Generator, which must choose practice problems that will lead to refinements of partially learned heuristics.

B. Describing Incompletely Learned Heuristics

LEX describes each partially learned heuristic by characterizing the range of *all plausible descriptions of that heuristic*; that is, the range of all describable generalizations of the heuristic which are consistent with the observed training instances. We call this set of alternative descriptions (or versions) of the heuristic the *version space* of the heuristic. Figure II-3 illustrates schematically the version space of a particular incompletely learned heuristic.

Given a set of training instances for some heuristic, and the generalization language with which heuristics are to be described, one could in principle describe the version space of the heuristic by listing all possible generalizations consistent with the data. But there are usually far too many plausible descriptions of a heuristic for this to be feasible. Fortunately, a much more compact representation of the version space is possible: Any version space can be completely characterized by storing only its maximally specific and maximally general members. Here we say that a heuristic H1 is more specific than heuristic H2, provided the preconditions of H1 are more strict than those of H2, and provided the two heuristics make the same recommendation. This "more-specific-than" relation partially orders the space of possible heuristics, and provides the basis for an efficient representation for version spaces.

Consider as an example, the version space shown in figure II-3. This figure illustrates several of the members of a particular version space, the "more-specific-than" relations among them, the most specific member (marked S) and the most general member (marked G). Thus, while there are very many members of the version space of figure II-3, S and G completely characterize the version space by delimiting the range of alternatives in the general-

to-specific partial ordering.

The remainder of this section summarizes the methods used by the four modules of LEX, in formulating and refining heuristics. This discussion centers around the example shown in figure II-2, which illustrates one learning cycle of LEX. This figure shows a proposed practice problem, part of the resulting search tree generated by the Problem Solver, one of the training instances produced by the Critic, and the sets S and G computed by the Generalizer to describe the resulting proposed heuristic.

C. The Problem Solver

The Problem Solver accepts as input a problem to be solved, along with a resource limit on the cpu time and memory space that it may expend in attempting to solve that problem. It uses a simple forward-search strategy, guided by whatever heuristics are available during the current propose-solve-criticize-generalize cycle. At each problem solving step, the heuristics are consulted. If none apply, then the lowest cost node in the search tree is expanded (cost here is measured by the cpu time expended to reach the node). If a heuristic applies to some node in the current search tree, then its recommendation is followed.

One interesting aspect of the Problem Solver is its effective use of incompletely learned heuristics to guide the search. Given the version space of some incompletely learned heuristic, what does it mean to say that the heuristic applies to a given problem state? It may be the case that some of the alternative generalizations in the version space of the heuristic match the problem state, while others do not. In other cases, all of the alternative generalizations may match the problem state. In this latter case there is really no question that the heuristic recommendation should be followed. Even though the heuristic is only incompletely learned, the applicability of the heuristic to this particular problem state is fully determined (i.e., it does not matter which of the alternative heuristic descriptions turns out to be correct, since they all apply to the problem state in question). This situation can be detected by the fact that all of the members of the S boundary of the version space will match the problem state. Similarly, the case where no members of the version space apply to the problem state is easily detected by the fact that no member of the G boundary matches the problem state. When some, but not all members of the version space match a problem state, the Generalizer estimates the proportion that do apply, and uses this number to resolve conflicts among competing heuristic recommendations.

D. The Critic

After a solution has been determined, the Critic faces the task of assigning credit (or blame) to individual search steps for their role in leading to (or away from) a solution. Figure II-2 illustrates part of the search tree generated by the Problem Solver for a given practice problem, and one of the associated positive training instances produced by the Critic. The positive instance shown in this figure corresponds to the first step along the path to the solution. In general, the criterion that the Critic attempts to apply is that each step along the lowest cost solution path is a *positive instance*, and each step leading away from this path is a *negative instance*.

An interesting issue here is that the Critic cannot be certain that the (heuristic) Problem Solver has found the lowest cost solution to the problem. Of course it could check whether any lower cost solution exists, simply by expanding every possible search path up to the cost of the

solution that has been found. But in practice, the search trees are quite large, and this is unrealistic. The Critic does, however, use the Problem Solver to explore selected portions of the search space which were not expanded during the original search for a solution. Before labeling a search step leading away from the solution path as a negative instance, the Critic calls the Problem Solver and has it search for a solution below that search step, giving it a resource allocation based on the resources spent in finding the original solution. Although in principle the Critic can still make errors in labelling search steps as positive or negative, we have found that it rarely produces incorrect training instances in practice.

E. The Generalizer

The Generalizer is the heart of the learning mechanism in LEX. It must generalize from the individual positive and negative training instances supplied by the Critic, in order to formulate general heuristics to aid in subsequent problem solving. This section describes the initial design of the LEX Generalizer, which is based on the candidate elimination method for revising version spaces. This approach to generalizing from examples was first developed as part of the Meta-DENDRAL program [Mitchell 78] for learning general rules of mass spectroscopy. A related approach of maintaining general and specific boundaries during generalization was suggested by [Young 77].

The basic idea behind the candidate elimination method used by LEX is to consider the version space of *all* possible generalizations of the first positive training instance, then to eliminate from consideration any generalization that is found to be inconsistent with subsequent training instances. As more training instances are considered, the version space will shrink, and eventually converge to a single description of the heuristic which is consistent with all of the observed data*. This candidate elimination approach to learning revises the specific and general boundary sets of the version space, *S* and *G*, to update its description of the heuristic that is being learned.

Proposing a new heuristic: When the Generalizer is given a new positive instance to which no heuristic applies, it creates a new heuristic by defining the initial sets *S* and *G* to bound the version space for that heuristic. The set *S* is initialized to the very specific version of the heuristic that applies *only* to the current positive training instance (this is the most specific possible version consistent with the single observed training instance). The set *G* is initialized to the version of the heuristic that suggests the operator will prove useful in *every* situation where it can validly be applied; that is, it is initialized to the given precondition of the operator recommended by the heuristic. Thus, in the example of figure II-2 *G* is initialized to the version whose precondition is the precondition for OP2. Here, $\int f_1(x) f_2(x) dx$ is the precondition of OP2, and stands for the product of any two functions of *x*. In the same figure, *S* is initialized to the generalization that matches only the positive training instance from which the heuristic is formed.

At this point, *S* and *G* delimit a broad range of alternative versions of the proposed heuristic, corresponding to *all* the generalizations expressible in the given language that are consistent with this single training instance. Figure II-3 illustrates this version space. As subsequent positive instances are considered, *S* becomes more general to include newly observed instances in which

OP2 is found to be useful. Likewise, as subsequent negative instances are considered, *G* becomes more specific in order to exclude negative instances in which OP2 may validly be applied, but in which it does not lead to an acceptable solution path. Thus, the range of alternative plausible versions of the heuristic delimited by *S* and *G* will narrow as new information is acquired through subsequent practice problems, and the uncertainty regarding the correct description of the heuristic is thereby reduced.

Version Space of Heuristic from Figure II-3:

S: $\int 3x \cos(x) dx \rightarrow$ Apply OP2 with
 $u = 3x$, and
 $dv = \cos(x) dx$

G: $\int f_1(x) f_2(x) dx \rightarrow$ Apply OP2 with
 $u = f_1(x)$, and
 $dv = f_2(x) dx$

New Training Instances:

Positive training instance:

$\int 3x \sin(x) dx \rightarrow$ Apply OP2 with
 $u = 3x$, and
 $dv = \sin(x) dx$

Negative training instance:

$\int 3x \sin(x) dx \rightarrow$ Apply OP2 with
 $u = \sin(x)$, and
 $dv = 3x dx$

Revised Version Space:

S: $\int 3x \text{ trig}(x) dx \rightarrow$ Apply OP2 with
 $u = 3x$, and
 $dv = \text{trig}(x) dx$

G:

g1: $\int \text{poly}(x) f_2(x) dx \rightarrow$ Apply OP2 with
 $u = \text{poly}(x)$, and
 $dv = f_2(x) dx$

g2: $\int f_1(x) \text{ transc}(x) dx \rightarrow$ Apply OP2
 with $u = f_1(x)$, and
 $dv = \text{transc}(x) dx$

Figure II-4: Revising the Version Space of a Heuristic.

Refining incompletely learned heuristics: Whenever a new training instance is obtained, the version space is refined by eliminating from consideration any member that is inconsistent with the that instance. In the current example, suppose the next practice problem leads to the positive and negative training instances shown in figure II-4. The effect of these training instances is shown in the figure. The most specific version, *S*, of the heuristic is generalized just enough to allow it to apply to the new positive training instance. Here $\text{trig}(x)$ replaces $\cos(x)$ so that the heuristic will apply to integrals containing *any* trigonometric function of *x*. The program determines this revision by first noting that the term $\cos(x)$ in the old *S* prevents that generalization from applying to the new instance, then determining the next more general term (see

*See [Mitchell 78] for a discussion of the conditions necessary to assure convergence.

the generalization hierarchy in figure 11-1) that can be substituted in order to include this new instance**

The general boundary of the revised version space of figure 11-4 has also been updated, in response to the new negative training instance. In this case, the old general boundary is moved to the least more specific generalization which does not match the new negative instance. For instance, g_1 is derived by replacing $f_1(x)$ (which represents "any function") by the next more specific *acceptable* expression, "poly". Because there are alternative ways in which the old G can be specialized, the result here is that the new G set contains two maximally general versions, g_1 and g_2 (note neither is more general than the other). These two are the most general descriptions of the heuristic consistent with the three observed training instances.

Refinement of version space continues this way, as subsequent training instances become available. New positive instances force the S boundary to become more general, while new negative instances force the G boundary to become more specific. Eventually, these boundaries converge to the heuristic description shown below.

$\int rx \operatorname{transc}(x) dx \Rightarrow$
Apply OP2 with $u = rx$, and $dv = \operatorname{transc}(x) dx$

F. The Problem Generator

Once a practice problem has been solved and analyzed, and the resulting training data has been used to propose and refine heuristics, the Problem Generator must propose a new practice problem. This module is responsible for focusing the system's efforts on useful activity, by choosing useful experiments. The Problem Generator is based on two tactics for producing useful practice problems.

The first problem generation tactic is to produce problems that will allow refining existing, incompletely learned heuristics. This is accomplished by selecting a partially learned heuristic, then generating a problem state that matches some, but not all, of the members of the version space of that heuristic. For example, consider the partially learned heuristic described by the version space in figure 11-3. The problem state $\int 3x \sin(x) dx$ matches some, but not all, of the alternative generalizations in this version space, and is therefore a useful problem to attempt to solve. By solving the problem, LEX will find out whether or not the heuristic should cover this problem state. If the answer is yes, a positive instance will be produced for this heuristic, and the S boundary of the version space will be generalized. If the answer is no, a negative instance will be produced, and the G boundary of the version space will be specialized.

The second tactic for problem generation is to create a problem that will lead to proposing a new heuristic. This is accomplished by looking for pairs of operators whose preconditions (left hand sides) intersect, but for which there is no current heuristic. When such a pair is found, a practice problem is generated to which they both apply. When this problem is considered by the Problem Solver,

Critic, and Generalizer, a new heuristic will be proposed which will be useful in selecting between the two operators. For example, the practice problem $\int 0 \cdot \cos(x) dx$ will lead to a heuristic to choose between the operator which suggests moving the real numbers outside the integral sign, and the operator which suggests multiplying through by zero.

As described above, LEX is able to learn heuristics by exploring its problem domain, generating practice problems, solving them, producing training instances, then generalizing from them. How good is LEX at learning, and how good does it become at integral calculus? We have found that when given *by hand* a reasonable sequence of practice problems, LEX can improve its performance by a few orders of magnitude [Mitchell 83] as measured by the number of search nodes generated on a set of test problems. But the Problem Generator does not do nearly as well at providing useful practice problems as we do. While this module does generate problems that lead to useful heuristics, it generates an inordinate number of unsolvable problems, and suffers from the fact that it has no ability to track the changing level of expertise of the Problem Solver. We have therefore begun rethinking the design of the Problem Generator, considering how it might be given the ability to track the Problem Solver's capabilities, and how it might acquire and use knowledge of which problems are easy and difficult to solve. Further discussion of the initial design of LEX can be found in [Mitchell 83].

As is often the case in experimental computer science research, the most important results to come out of experimenting with LEX were not the specific measurements of improvement in problem solving performance, but rather the intuitions we gained regarding weaknesses in the design, and changes in viewpoint of where the interesting issues lie. Section 3 discusses the most significant of the changes in our viewpoint: a fundamental change in the approach to the generalization problem. Section 4 then discusses several other general points that follow from our experience with LEX and from other recent research in the area of learning.

III Giving the Learner a Goal

The heart of the design of LEX is its Generalizer, based on a candidate elimination approach to searching a large space of candidate generalizations for those consistent with the observed training instances. While LEX has learned useful heuristics in this way, there are certain fundamental problems with its approach, and with *any* of the approaches to generalization that are based solely on finding syntactic similarities among training instances.

The problem is that there really is very little justification for any inductive leap that the program makes. For example, when LEX observes the two positive training instances $\int 3\sin(x)dx$, and $\int 5\sin(x)dx$, it leaps to the conclusion that 3 and 5 can be generalized to "any integer". But what is the basis for this inductive leap? What basis does the generalizer have for assuming that the correct generalization is "any integer", rather than "any prime integer", or "any integer between 2 and 17", or "any integer except 251"? Really there is none. The Generalizer leaps to "any integer" simply because it doesn't know any of these more specific notions. To be safe, we could have it consider all of these intermediate concepts, and lots of others, such as "any integer except 83". But this is clearly futile—the system then could never generalize to "any

**Although the disjunction " $\cos(x)$ OR $\sin(x)$ " would be a more specific generalization than "trig", this disjunction is not currently in the generalization language, and therefore cannot be stated by the program. Of course if this disjunction were defined a priori as a separate term in the language, then it would be considered by the Generalizer.

integer", since it would have to first rule out all of these more specific alternatives.

The fact is that we really don't want LEX to consider generalizing to "any integer except 982". We know that generalizations like this one would probably be useless for describing heuristics for integral calculus. In the end, we really could argue why such concepts were irrelevant, and why other notions like "any integer" and "any integer except 0" might be important. How do we know? Because we know that the *learning goal*, or *purpose* of the Generalizer in LEX is to form heuristics for integral calculus, and we know the problem solving operators in the system, and we know that heuristics are meant to suggest problem solving steps that lead to solutions, and we know how to reason about such things to convince ourselves that certain generalizations just aren't going to be relevant.

The problem is that while we knew all of these things, LEX did not. The Generalizer in LEX really had no knowledge that its goal was to learn the concept "a general description of those problem states for which operator X leads to a solution". Without at least this knowledge of its learning goal, LEX cannot justify any of its inductive leaps, or—more importantly—reason about the basis on which two positive training instances should be considered similar.

Notice that while the Generalizer does not know what it is trying to learn, the Critic in some sense does. The Critic is responsible for labeling search steps as positive or negative instances of the heuristic to be learned. The criterion that it uses for labelling (i.e., that positive instances are those search steps that lead to a solution) is the learning goal. But the Critic "knows" the learning goal only in the sense that it has procedures which apply this criterion to label training instances. What LEX really needs is a *declarative* description of its learning goal, so that it can reason about its purpose. The remainder of this section summarizes our more recent research [Mitchell 82, Utgoff 82, Utgoff 83] on giving LEX explicit knowledge of its learning goal.

We have found that giving LEX declarative knowledge of its learning goal leads to two very important capabilities. First, it provides the basis for generalizing from a single positive training instance, based on an analysis of how that training instance satisfies the criterion used by the Critic in labelling positive instances. Second, it provides the basis for discovering new syntactic terms, such as "odd integer", which must be added to the generalization language in order to describe heuristics which cannot otherwise be expressed. The following subsections describe the way in which the learning goal is represented, and the way in which it leads to these two new capabilities***. In this discussion, we refer to the new version of LEX which is able to reason about its goal as LEX2.

A. Describing the Learner's Goal

The learning goal, or credit assignment criteria is described to LEX2 in predicate calculus, as shown below. While we have experimented with two alternative learning goals (see [Mitchell 82]), to simplify this discussion we assume that the goal is to learn heuristics that recommend problem solving steps leading to any solution (rather than the minimum cost solution). Thus, any problem solving step that applies some operator, *op*, to some problem state,

state, is a positive instance, provided that *state* is not already a goal state, and provided that applying *op* to *state* results in either a goal state or a solvable state. The predicate calculus statement of this, given to LEX2, is as follows

$$\begin{aligned} \text{PosInst}(\text{op}, \text{state}) \Leftrightarrow \\ \sim \text{Goal}(\text{state}) \\ \wedge [\text{Goal}(\text{Apply}(\text{op}, \text{state})) \vee \text{Solvable}(\text{Apply}(\text{op}, \text{state}))] \end{aligned}$$

Here, *PosInst* describes the learning goal. The predicate *Goal* is the predicate for recognizing solved problems (e.g., expressions with no integrals). LEX2 has the knowledge that the *Goal* predicate is satisfied by any problem state that matches "*f(x)*" (e.g., any state that does not contain an integral sign). The function *Apply* applies operators to states, and *Solvable* is the predicate that tests whether a state can be transformed to a *Goal* state with the available operators (*Solvable* has its own predicate calculus definition in LEX2). Thus, the *PosInst* predicate is defined in enough detail that LEX2 can expand the definition to show how (prove that) *PosInst* is satisfied by any given positive instance.

B. Analyzing Training Instances to Guide Generalization

This section sketches how the declarative representation of the credit assignment criterion, *PosInst*, is used by LEX2 to produce a justifiable generalization of a heuristic based on analysis of a single training instance. This is discussed in more detail in [Mitchell 82]. The key idea here is that by analyzing how the observed positive instance satisfies the *PosInst* predicate, it is possible to determine a sufficient condition for satisfying *PosInst*. This condition is then reexpressed as a justifiable generalization of the given positive instance.

As an example, suppose that the system has just produced the problem solution shown in figure III-1, and that the generalizer is now considering the positive training instance corresponding to the first search step along the solution path shown. If no heuristic yet exists for the operator (OP1) associated with this search step, the empirical generalization method described earlier will produce the following version space for the new heuristic:

$$S: \int 7 x^2 dx \Rightarrow \text{use OP1}$$

$$G: \int r f(x) dx \Rightarrow \text{use OP1}$$

The following discussion shows how analysis of this training instance leads to additional information for refining the above version space by generalizing the *S* boundary. The trace of this analysis, shown below, is broken into four main stages: (i) Generate an explanation that shows how the current positive instance satisfies *PosInst*, (ii) Extract from this explanation a sufficient condition for satisfying *PosInst*, (iii) Restate the sufficient condition in terms of the generalization language (i.e., the language for describing left-hand sides of heuristics), as restrictions on various problem states in the solution tree, and (iv) Propagate the restrictions on various problem states through the solution tree, and combine them into a generalization that corresponds to a sufficient condition for assuring *PosInst* will be satisfied. The result of these four stages is a justifiable generalization of the given training instance. The process itself appears to be general enough to apply to learning of search heuristics in domains outside integral calculus.

***Both of these extensions to LEX are now implemented to the stage where they have been run through the major portions of several examples. Neither is fully implemented at this point.

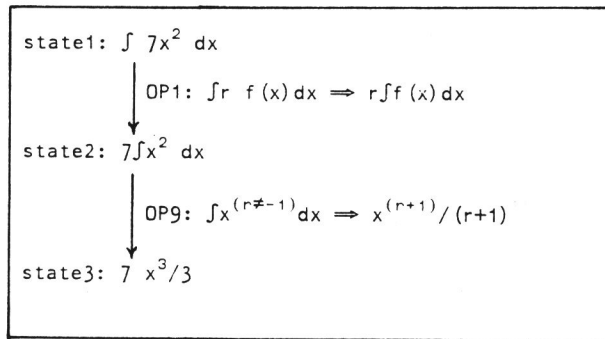


Figure III-1: A Solution Tree

Stage 1: Produce an explanation of how the current training instance satisfies PosInst. By instantiating the definition of PosInst for the positive instance in question, a proof is produced that PosInst(OP1, State1). The result of this process is an And/Or proof tree, which we call the *explanation tree* for the training instance. For example, the explanation tree for the positive training instance <OP1, State1> is shown figure III-2. Note that nodes in the *explanation tree* correspond to statements about various problem states and operators in the associated *solution tree*. The explanation tree in this example indicates that <OP1, State1> is a Positive instance because (1) State1 is not a Goal state, and (2) by applying OP9 to the state resulting from the positive instance step, it is possible to reach a goal state. This explanation tree thus indicates essential features of the solution tree which allow the training instance to satisfy PosInst.

Stage 2: Extract a sufficient condition for satisfying PosInst. If the explanation tree is viewed as a proof that PosInst is satisfied by the current training instance, then any set of nodes that satisfy this And/Or tree yield a sufficient condition for satisfying PosInst. In the current example, if the tip nodes of the explanation tree are selected, then the resulting sufficient condition is "Posinst will be satisfied for any state s as long as s is not a goal state, and one can reach a goal state from s by applying to OP1 followed by OP9." The result of this stage, expressed in predicate calculus, is:

$$(\forall s) \text{PosInst}(\text{OP1}, s) \Leftarrow \neg \text{Goal}(s) \wedge \text{Goal}(\text{Apply}(\text{OP9}, \text{Apply}(\text{OP1}, s)))$$

Stage 3: Restate the sufficient condition by restating each clause as some generalization which must match some state in the solution tree. In the current example, the sufficient condition can be restated as "Posinst will be satisfied for any state s as long as (1) s matches the generalization " $\int f(x)dx$ " (i.e., it is not a goal state), and (2) the result of applying OP1 followed by OP9 produces a state that matches " $f(x)$ " (i.e., a goal state)." In order to reexpress the sufficient condition in this way, LEX2 requires knowledge of the correspondence between various predicates (e.g., Goal) and statements in the generalization language. For example, it requires the knowledge that if state s Matches " $f(x)$ ", then s will satisfy the predicate Goal****. In predicate calculus notation, the result of this step is:

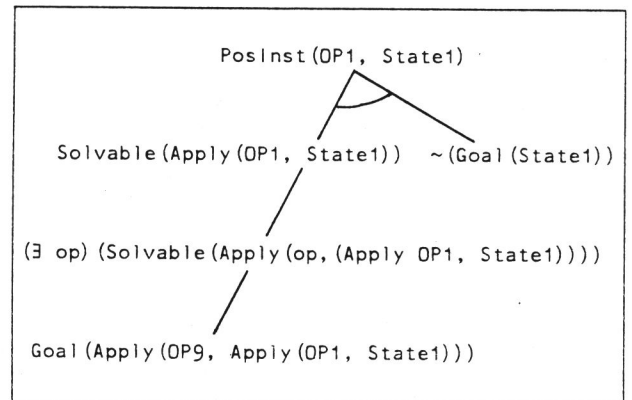


Figure III-2: An Explanation Tree for PosInst(OP1, State1)

$$(\forall s) \text{PosInst}(\text{OP1}, s) \Leftarrow$$

$$(\text{Match}(\int f(x)dx, s) \wedge \text{Match}(f(x), \text{Apply}(\text{OP9}, \text{Apply}(\text{OP1}, s))))$$

Stage 4: Propagate the restrictions on various problem states through the solution tree to determine the corresponding restrictions on the training instance under consideration. Given restrictions on some problem state, x , in the solution tree, the final step is to propagate these restrictions back through the solution tree to deduce the corresponding constraints on an earlier problem state. This back propagation of restrictions is necessary in order to restate the sufficient condition on PosInst in terms of a generalization that applies to the training instance. In the current example, the result of this back propagation (plus a straightforward simplification step) is the following:

$$(\forall s) [\text{PosInst}(\text{OP1}, s) \Leftarrow \text{Match}(\int r x^{(r*-1)} dx, s)]$$

Finally we have a sufficient condition for PosInst(OP1, s) stated as a generalization that must match s . Thus, from an analysis of how the training instance satisfies the PosInst predicate, the system is able to extract the essential characteristics of the positive training instance—those characteristics that are sufficient to assure that OP1 should be recommended. This sufficient condition can be used to initialize the S boundary of the version space for this heuristic, as shown below:

$$S: \int r x^{(r*-1)} dx \Rightarrow \text{Apply OP1}$$

$$G: \int r f(x) dx \Rightarrow \text{Apply OP1}$$

In giving the Generalizer a declarative definition of its goal, we have given it the ability to analyze a single positive training instance, and to produce a justifiable generalization of the instance based on this analysis. The generalization produced is justified in the sense that it constitutes a logically sufficient condition for satisfying the PosInst

****While this knowledge was given beforehand to the system, it seems that the system should be able to learn such facts itself, by applying the same mechanisms that it uses to learn the correspondence between the predicate PosInst, and the generalization language.

predicate*****. In general this analysis can be very difficult, and the result of the analysis will rarely be both a sufficient and a necessary condition (i.e., the correct generalization of the heuristic). As a result, it is still necessary to collect multiple training instances, to analyze them, and to incrementally formulate the correct generalization of the concept. The version space representation of partially learned concepts is again useful for supporting incremental learning. The generalized positive training instances should be used to update the S boundary of the version space, in the same way that the positive instances are used in LEX.

This goal-directed learning is a kind of "learning by being told", to use Winston's terminology [Winston 75]. That is, LEX2 is told the concept that it must learn -- it is told the definition of the PosInst predicate. In giving the Generalizer this goal, we have changed the fundamental nature of the generalization problem: Without knowledge of this goal, the Generalizer was forced to make inductive leaps to guess the classification of unobserved instances. With knowledge of this goal, it already knows (in principle) the classification of all the instances. But while PosInst is a *valid* definition of the concept to be learned, it is not a very *useful* one. In particular, to determine whether a given instance is an instance of the concept defined by PosInst, one must expand the definition of PosInst, which essentially involves solving the given integral! Thus, the learning problem has become one of converting this valid definition into a form in which it is more useful for recognizing instances of the concept. This kind of problem is what several researchers [Hayes-Roth 80, Mostow 81, Keller 83] have called operationalization problems, and this work is related to their ideas, as well as to the ideas of Winston [Winston 81] and deJong [deJong 82], who also discuss methods for generalizing from single training instances.

C. Automatically Extending the Vocabulary for Describing Heuristics

As discussed earlier, a major difficulty in learning from examples is the reliance of the learner on some a priori chosen set of primitive terms for representation. In particular, if the learner's generalization language does not contain the right primitive vocabulary, then some concepts will not be describable, and hence will not be learnable. One major attraction of the kind of goal-directed generalization described in the previous section, is that it provides a basis for introducing new vocabulary terms into the generalization language of LEX2. This section summarizes how such new terms can be formed, and is based on work described in [Utgoff 82, Utgoff 83].

As an example, consider the solution path shown in figure III-3, and the positive training instance corresponding to the first step of this solution path. If this positive training instance is observed, together with the positive training instance $\int \cos^5(x) dx$, and the negative training instance $\int \cos^6(x) dx$, then LEX will be unable to produce a heuristic that matches these two positive instances, and excludes the negative instance. The problem here is that LEX does not have any term in its the generalization language, which includes both 5 and 7 while excluding 6.

*****It would be interesting to consider the dual of this -- producing generalized negative instances whose negation would correspond to a necessary condition for satisfying the learning goal.

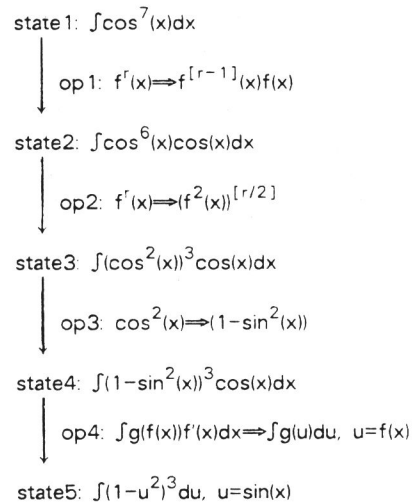


Figure III-3: Solution Path for $\int \cos^7(x) dx$

In this case, the kind of goal-directed learning presented in the previous section can lead to the generation of the new term "odd integer", to be added to the generalization language. This new term is defined as a by-product of analyzing the positive instance shown in the figure. As before, this goal-directed analysis leads first to a set of constraints on various nodes in the search tree, which characterize why the training instance is positive. These constraints are then propagated through the problem-solving operators in the search tree to determine which features of the training instance were necessary to satisfy these statements. *It is during this propagation and combination of constraints that new descriptive terms are suggested.*

To see how this works in the case shown in figure III-3, suppose that the first three stages of analysis determine that PosInst is satisfied by <OP1, state1> because state5 is a polynomial raised to some integer power*****. By the propagation of this constraint, step by step, back through the sequence of operators in the solution tree, it can be determined that the corresponding constraint on State1 is that it must match an expression of the form $\int \cos^c(x) dx$ where c is constrained to satisfy the predicate "real(c) $\wedge$ integer($(c-1)/2$)". This predicate is the definition of "odd integer".

Where do such new terms come from? The system had to know about the operations of "division" and "subtraction", as well as the predicates "integer" and "real", in order to create its definition of oddness. And it had to combine these in a very specific way in order to create the definition of oddness, rather than one of the millions of other useless definitions. The operations "subtraction" and "division" arise from the definitions of operators OP1 and OP2 used in the solution tree of figure III-3. In the process of backpropagating the constraint on state5, step by step, through the sequence of operators in the given solution tree, LEX2 is led to combine these operators with

*****The system could have concluded this by expanding the definitions of PosInst and Solvable, and then by applying the knowledge that any polynomial raised to an integer power is Solvable.

the constraints that it propagates, in the particular way that leads to the definition of odd integer.

Once a new term such as "odd integer" has been created in this fashion, it is desirable to add it to the generalization language. But in order to do this, it is necessary to determine where it fits in the generalization hierarchy. For example, there is no easy way for the system to determine that "odd integers", according to its new definition of them, is a specialization of "integers". This task of assimilating new terms into the generalization hierarchy can require considerable domain specific knowledge itself (e.g., the knowledge that the integers are closed under subtraction).

It is interesting to compare this approach to generating new terms with the approach in AM [Davis 81], Eurisko [Lenat 82] and BACON [Langley 83]. AM creates notions like "prime numbers", by combining more primitive predicates and operations such as "integers" and "divisors--of". BACON creates notions such as the "voltage" of a battery, by combining primitive terms such as "current" and "conductance". In both of these cases, the search for new terms is a heuristically driven, generate and test search. In AM, for example, there are very many possible ways of combining primitive operations and predicates. In contrast, the goal-directed generation of new terms in LEX2 involves very little search -- the particular combination of predicates and operations that defines the new term follows directly from the constraints produced by the solution analysis, and from the sequence of operators through which these constraints must be propagated. Much work remains to be done in exploring both of these methods for constructing and assimilating new primitive terms.

IV Lessons for Machine Learning

This section summarizes some of the lessons that follow from a variety of recent work on learning and problem solving. Some of these lessons are illustrated by our experiences with LEX, while others derive from related research on topics such as learning of heuristic strategies, problem assimilation and reformulation, and use of meta-representations in guiding problem solving and learning.

For systems that learn incrementally, an explicit representation for incompletely learned concepts is important. Any system that learns concepts over some period of time must have some way to keep track of what it does and does not yet know about incompletely learned concepts. In LEX and in the Meta-DENDRAL program [Buchanan 78] it has been useful to represent incompletely learned concepts in terms of their version space. The benefits of this representation include (1) It allows the system to use an incompletely learned concept to classify new instances as positive, negative, or undecidable based on current knowledge. (2) It provides a basis for generating instances whose training classification will allow refining the concept. For example, the LEX Problem Generator generates practice problems on this basis. (3) It provides a sound basis for incremental generalization methods, either empirical or goal-directed.

By allowing the learner to have explicit knowledge of its learning goal, it is possible to employ goal-directed learning methods which are more powerful than empirical, syntactic methods. Goal-directed approaches to generalization, such as that discussed in the previous section, differ from empirical, syntactic approaches in a fundamental way: Whereas empirical approaches to generalization must make inductive leaps to guess the

classification of unobserved instances, goal-directed approaches begin with knowledge of how (in principle) to classify every possible instance. Thus, in goal-directed generalization, the problem is not so much an inductive inference problem as it is a problem of reexpressing the "correct but unuseable" definition of the concept in a more useable form. Our experience with LEX2 is that the goal-directed analysis of training instances can be quite difficult and expensive***** and will not always produce a strong generalization. Therefore, goal-directed learning in LEX2 remains an incremental process, relying on analyzing and generalizing a number of training instances, and using the information extracted from each training instance to refine the version space description of the partially learned concept. Recent work along a similar vein by several researchers (e.g., [Winston 81, Keller 83, Mostow 81, deJong 82]) suggests that we are beginning to see a broad move toward research on methods in which there is a justifiable basis for generalization.

Goal-directed learning methods can provide a method for discovering new vocabulary terms that are essential for describing (and therefore learning) heuristics. The same goal-directed interpretation of training instances that leads to generalizing from single instances in LEX2 can also lead to discovery of new terms. These terms arise when attempting to analyze the particular sequence of search steps that lead to a solution, together with the constraints on the training instance that allow it to satisfy the criterion for positive instances. This approach seems to generalize to discovery of new terms for describing search heuristics in other task domains as well. It is less clear how to generalize it to tasks other than learning search heuristics. This goal-directed method for generating new terms provides an interesting contrast to methods in AM [Davis 81] and BACON [Langley 83] which employ heuristically-driven generate-and-test strategies for discovering new terms.

Initial methods for learning by experimentation have been developed, and show the need for more sophisticated approaches. LEX employs two problem generation tactics which focus on repairing specific deficiencies in LEX's set of heuristics: (1) suggest a problem to refine an incompletely learned heuristic, and (2) suggest a problem to generate a new heuristic that will choose between alternative problem solving operators. As noted earlier, although the LEX tactics can produce useful practice problems, the Problem Generator suffers from its lack of knowledge of which problems are easy, and which are difficult. Again, much more research is needed to understand useful experimentation strategies. Some issues that have not been addressed include (1) What is a good global strategy for choosing sequences of problems (e.g., should one attempt to refine existing partially learned heuristics before generating new heuristics?), and (2) How might the activities of other modules in the learning system be sensitive to the purpose of the current experiment (e.g., if the Critic knew that the practice problem was generated to obtain the classification of a particular training instance, then it might spend more of its resources to improve its accuracy in classifying that training instance). One general observation should be made regarding the long-term attractiveness of systems that can learn with a minimum of human intervention: If computer cycles will be essentially free and plentiful a decade from now, then spending a cpu

*****We have seen LEX2 generate explanation trees with a few thousand nodes, when considering a more complex learning goal than that described in this paper.

year or two on several computers may not be an unreasonable expense for developing a high-performance problem solving system, so long as the effort does not require a heavy investment of human resources.

The ability to shift the focus of learning is a major requirement for effective learning. LEX failed to evolve into a top-notch problem-solver (e.g., one comparable to MACSYMA) for several reasons. LEX2 addresses some of these difficulties, by changing the empirical generalization problem into a goal-directed learning problem. But there is fundamental limitation in both systems: they both approach learning in problem solving as though it were a matter of augmenting one particular kind of knowledge (e.g., the search heuristics), within an otherwise fixed problem solving framework. In fact, our experience shows that while performance in LEX initially improves as it learns heuristics, past some point new heuristics (even very good ones) fail to produce significant improvements in overall performance. The reason is that the bottleneck that limits system performance no longer lies in making the kinds of decisions for which these heuristics are useful. For example, while the learned heuristics are very useful in determining which operator should be applied to a given problem state, they are not helpful enough in selecting which problem state from the current search tree should be expanded next. Eventually, it is this choice of which node to expand next that limits problem-solving performance. Furthermore, as new heuristics are added, the cost of considering all of them at each problem-solving step becomes very expensive. LEX continues to learn good heuristics, but they do not address what has become the most limiting bottleneck to improved performance. The situation is analogous to trying to optimize a large program by hand. One must constantly search for the current bottleneck within the program, improve the portion of the program which leads to that bottleneck, then isolate the next bottleneck. To be effective, learning systems must have a similar capability to alter their focus of attention. They must be able to monitor their performance at various tasks, isolate bottlenecks which limit performance, formulate learning tasks to improve performance, accomplish them, then move on to the next bottleneck.

These capabilities to focus learning attention, to formulate specific learning tasks, and to pursue them, are as essential to learning in problem solving as the "wandering bottleneck phenomenon" is commonplace to development of large programs. In order to focus learning attention on the current performance bottleneck, the system must have enough of a capability for self-description and self-monitoring that it can know the roles of its component parts, and monitor how well those roles are being fulfilled. When a performance bottleneck is isolated (e.g., the current method of selecting which search node to expand next is weak), the system must have a method for formulating a specific learning task to remove the bottleneck (e.g., learn heuristics to rank search nodes according to the likelihood that they lead quickly to a solution). Note that in formulating this task, the system is formulating an explicit learning goal which may be used as the basis for goal-directed learning of the required knowledge. Thus, systems that formulate their own learning tasks will produce a learning goal as a part of this process. Having formulated a learning task, the system must next isolate a learning schema (e.g., the goal-directed learning schema) and instantiate that schema for the problem at hand.

Recent work on Eurisko [Lenat 82], and an earlier paper on cognitive economy [Lenat 79] provide a number of key ideas on self-monitoring, self-description, and

shifting the focus of learning activity. Eurisko shows how a system can monitor certain aspects of its performance, and use the resulting measurements to focus its attention on specific learning tasks. It uses an ordered agenda to keep track of the many suggested tasks, giving especially high priority to tasks suggested for multiple reasons. Eurisko also shows us how a very uniform representation for a problem solver allows a single learning mechanism to apply across a broad spectrum of learning tasks. The uniform representation here is in terms of structured objects (frames, schema), and the learning mechanism is a set of heuristics that suggest how to create new objects, fill in slots, and create new types of slots. Because of this uniformity, the heuristics which drive learning in Eurisko can apply across a wide variety of learning tasks (e.g., to learning new concepts in number theory, and to learning new heuristics).

Research on problem assimilation, formulation, and reformulation is an essential component of research on learning. When a problem is encountered, we must assimilate and formulate that problem by mapping it onto some problem-solving method, or schema that we understand how to use. For example, if we want to solve a problem using some search method, we must first define the search space and search operators. Work on problem formulation and reformulation is important to progress in learning for two reasons. One reason is that as previous work in this area [Amarel 71] has shown, problem reformulation can lead to large improvements in problem solving performance. It is therefore one of the most significant possible routes toward improving problem-solving expertise. The second reason is that in order for a learner to shift its learning focus to address some new performance bottleneck, it must be able to *formulate its new learning problem* in terms of some learning schema that it possesses. That is, once the learner has identified a problem at the level of stating the goal "learn heuristics to rank problem states according to their likelihood of leading to a solution", it must then formulate a method for solving this problem in terms of some known learning method. Recent work in the general area of problem formulation and reformulation includes [Simon 77, Amarel 71, Korf 80, Hayes-Roth 83, Mostow 81]. This work has shown us several general formulations for problem solving schema, some ideas of how to map problems onto these schema, and some ideas of how to reformulate problems in order to reduce the complexity of the search for their solutions.

Learning to learn might not be very different from learning other forms of problem solving. As we begin to understand that learning is another kind of problem solving (as Simon and Lea [Simon 74] pointed out some years ago), it becomes more clear that systems which learn problem solving expertise might also be able to learn learning expertise. The payoff for this capability is great—a small improvement in learning abilities may lead over time to considerably more effective problem solving, since the impact of this small improvement accumulates over the many times that the learning mechanism is applied. Again, Eurisko provides an example of a system that has a capability to improve its own learning strategy, by virtue of describing this strategy as a set of heuristics open to change by the system itself.

V Conclusion

Overall, it seems that we are beginning to see some pieces of the puzzle of learning. We now understand something about methods for producing justifiable generalizations, about how learning systems can alter their

representations, about the importance of being able to refocus learning activity and formulate learning problems, about the importance of self-description and self-monitoring to support this refocusing, about learning to learn.

Of course there is plenty more that we do not yet understand. One thing seems clear: we need more research groups developing and experimenting with alternative architectures for problem solving and learning. We need to develop an understanding of the space of possible architectures, of what kinds of initial structure and knowledge should be provided in a learning system, and of how much initial flexibility the system requires in order to support learning. The early work on perceptrons lies at one extreme, representing systems with very little initial knowledge. Eurisko provides an interesting data point, as a system with much more initial structure and knowledge, yet with a great deal of flexibility. LEX2 provides a data point as a system with considerable knowledge for guiding learning, but with the fatal flaw that much of its structure cannot be altered at all. There are other data points too, but we still have only a few. And even we humans don't know how to generalize from data that skimpy!

VI Acknowledgments

I wish to thank Saul Amarel, Rich Keller, Doug Lenat, and Lou Steinberg, for providing constructive comments on earlier drafts of this paper. The work on LEX is the result of the efforts of several people over the past three years, notably Paul Utgoff, Richard Keller, Ranan Banerji, Bernard Nudel, William Bogdan, Adam Irgon, and Peter Spool. This research has been supported by the National Science Foundation under grant MCS80-08889, and by the National Institutes of Health under grant RR-64309.

References

- [Amarel 71] Amarel, S.
Representations and Modeling in
Problems of Program Formation.
Machine Intelligence 6, 1971.
- [Anderson 83] Anderson, J. R.
Acquisition of Proof Skills in Geometry.
In Michalski, R. S., Carbonell, J. G. and
Mitchell, T. M. (editors), *Machine
Learning*, . Tioga, 1983.
- [Anzai 79] Anzai, Y. and Simon, H.
The theory of learning by doing.
Psychological Review 36(2):124-140,
1979.
- [Brazdil 78] Brazdil, P.
Experimental Learning Model.
In *Proceedings of the AISB Conference*.
1978.
- [Buchanan 78] Buchanan, B. G. and Mitchell, T. M.
Model-directed learning of production
rules.
In Waterman, D. A. and Hayes-Roth,
F. (editors), *Pattern-Directed
Inference Systems*, . Academic
Press, New York, 1978.
- [Davis 81] Davis, R.
Applications of meta level knowledge to
the construction and use of large
knowledge bases.
In Davis, R. and Lenat, D. (editors),
*Knowledge-based Systems in
Artificial Intelligence*, . McGraw-
Hill, New York, 1981.
- [deJong 82] DeJong G.
Automatic Schema Acquisition in a
Natural Language Environment.
In *Second National Conference on
Artificial Intelligence*, pages
410-413. Pittsburgh, PA, August,
1982.
- [Hayes-Roth 74] Hayes-Roth, F.
Schematic Classification Problems and
their Solution.
Pattern Recognition 6:105-113, 1974.
- [Hayes-Roth 80] Hayes-Roth, F., Klahr, P. and Mostow,
D. J.
*Knowledge acquisition, knowledge
programming, and knowledge
refinement*.
Technical Report R-2540-NSF, The
Rand Corporation, Santa Monica,
CA., May, 1980.
- [Hayes-Roth 83] Hayes-Roth, F.
Using Proofs and Refutations to Learn
from Experience.
In Michalski, R. S., Carbonell, J. G. and
Mitchell, T. M. (editors), *Machine
Learning*, . Tioga, Palo Alto, 1983.
- [Keller 83] Keller, R.
Learning by Operationalizing Concepts.
In *Third National Conference on
Artificial Intelligence*, pages to
appear. Washington, D.C., August,
1983.
- [Korf 80] Korf, R. E.
Toward a Model of Representation
Changes.
Artificial Intelligence 14(1):41-78,
1980.
- [Langley 81] Langley, P.
*Strategy Acquisition Governed by
Experimentation*.
Technical Report CIP No.431, Carnegie-
Mellon Univ., Pittsburgh, PA, 1981.
- [Langley 83] Langley, P., Simon, H. A. and Bradshaw,
G. L.
Rediscovering Chemistry with the
BACON System.
In Michalski, R. S., Carbonell, J. G. and
Mitchell, T. M. (editors), *Machine
Learning*, . Tioga, Palo Alto, 1983.

- [Larkin 79] Larkin, J., McDermott, J., Simon, D., and Simon, H.
Models of Competence in Solving Physics Problems.
Technical Report C.I.P. No. 408,
Carnegie-Mellon University,
Pittsburgh, PA, November, 1979.
- [Lenat 79] Lenat, D. B., Hayes-Roth, F. and Klahr, P.
Cognitive Economy.
RAND technical report N-1185-NSF,
The RAND Corporation, Santa
Monica, CA., June, 1979.
- [Lenat 82] Lenat, D. B.
*The Role of Heuristics in Learning by
Discovery: Three Case Studies.*
Tioga, Palo Alto, 1982.
- [Michalski 73] Michalski, R. S.
Discovering Classification Rules using
Variable-Valued Logic System VL1.
In *Third International Joint Conference
on Artificial Intelligence*, pages
162-172. 1973.
- [Michalski 80] Michalski, R. S. and Chilausky, R. L.
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.
*Policy Analysis and Information
Systems* 4(2), June, 1980.
Special issue on knowledge acquisition
and induction.
- [Mitchell 78] Mitchell, T. M.
*Version Spaces: An approach to concept
learning.*
PhD thesis, Stanford University,
December, 1978.
also Stanford CS report STAN-
CS-78-711, HPP-79-2.
- [Mitchell 82] Mitchell, T.
*Toward Combining Empirical and
Analytical Methods for Inferring
Heuristics.*
Technical Report LCSR-TR-27,
Department of Computer Science,
Rutgers University, 1982.
- [Mitchell 83] Mitchell, T. M., Utgoff, P. E. and Banerji,
R. B.
Learning by Experimentation: Acquiring
and Refining Problem-Solving
Heuristics.
In Michalski, R. S., Carbonell, J. G. and
Mitchell, T. M. (editors), *Machine
Learning*, . Tioga, 1983.
- [Mostow 81] Mostow, D. J.
*Mechanical Transformation of Task
Heuristics into Operational
Procedures.*
PhD thesis, Carnegie-Mellon University,
1981.
- [Neves 78] Neves, D. M.
A computer program that learns
algebraic procedures.
In *Proceedings of the 2nd Conference
on Computational Studies of
Intelligence*. Toronto, 1978.
- [Politakis 82] Politakis, P.
*Using Empirical Analysis to Refine
Expert System Knowledge Bases.*
PhD thesis, Rutgers University, August,
1982.
- [Simon 74] Simon, H. A. and Lea, G.
Problem Solving and Rule Induction: A
Unified View.
In Gregg, L. W. (editor), *Knowledge and
Cognition*, pages 105-127.
Lawrence Erlbaum Associates,
Potomac, Maryland, 1974.
- [Simon 77] Simon, H. A.
Artificial Intelligence Systems that
Understand.
In *Fifth International Joint Conference
on Artificial Intelligence*, pages
1059-1073. Cambridge, Mass.,
1977.
- [Smith 77] Smith, R. G., Mitchell, T. M., Chestek,
R. A. and Buchanan, B. G.
A Model for Learning Systems.
In *Fifth International Joint Conference
on Artificial Intelligence*, pages
338-343. Cambridge, Mass., 1977.
- [Utgoff 82] Utgoff, P. E. and Mitchell, T. M.
Acquisition of Appropriate Bias for
Inductive Concept Learning.
In *Proceedings of the 1982 National
Conference on Artificial
Intelligence*. Pittsburgh, August,
1982.
- [Utgoff 83] Utgoff, P. E.
Adjusting Bias in Concept Learning.
In *Proceedings of IJCAI 1983*.
Karlsruhe, August, 1983.
- [Vere 78] Vere, S. A.
Inductive learning of relational
productions.
In Waterman, D. A. and Hayes-Roth,
F. (editors), *Pattern-Directed
Inference Systems*, . Academic
Press, New York, 1978.
- [Winston 75] Winston, P. H.
Learning Structural Descriptions from
Examples.
In Winston, P. H. (editor), *The
Psychology of Computer Vision*,
chapter 5. McGraw Hill, New York,
1975.
- [Winston 81] Winston, P. H.
*Learning New Principles from
Precedents and Exercises.*
AIM 632, MIT, May, 1981.
- [Young 77] Young, R. M., Plotkin, G. D. and Linz,
R. F.
Analysis of an Extended Concept-
Learning Task.
In *Fifth International Joint Conference
on Artificial Intelligence*, pages
285. August, 1977.