6

LEARNING BY EXPERIMENTATION: ACQUIRING AND REFINING PROBLEM-SOLVING HEURISTICS

Tom M. Mitchell
Paul E. Utgoff
Rutgers University

Ranan Banerji
St. Joseph's University

ABSTRACT

This chapter concerns learning heuristic problem-solving strategies through experience. In particular, we focus on the issue of learning heuristics to guide a forward-search problem solver, and describe a computer program called LEX, which acquires problem-solving heuristics in the domain of symbolic integration. LEX acquires and modifies heuristics by iteratively applying the following process: *(i)* generate a practice problem; *(ii)* use available heuristics to solve this problem; *(iii)* analyze the search steps performed in obtaining the solution; and *(iv)* propose and refine new domain-specific heuristics to improve performance on subsequent problems. We describe the methods currently used by LEX, analyze strengths and weaknesses of these methods, and discuss our current research toward more powerful approaches to learning heuristics.

6.1 INTRODUCTION

Efforts to build powerful, specialized, heuristic problem solvers have met with increasing success over the past decade. However, identifying and encoding the domain-specific heuristics necessary for high performance of these systems is a painstaking, difficult process. As the complexity of a heuristic

program grows, it becomes increasingly difficult for the system builder to predict how the addition of a particular new heuristic or operator will affect overall system performance. In response to this problem, there has been increased interest over the past several years in developing semi-automated and fully-automated methods to help construct expert heuristic problem solvers [Waterman, 1970; Davis, 1981; Buchanan, 1978; Politakis, 1979] (See also Chapter 7 of this book). At the same time, in the Cognitive Psychology literature there have been several attempts to model acquisition of problem-solving skills in humans [Anzai, 1979; Neves, 1978] (See also Chapter 7 of this book).

The research presented here is directed toward devising methods by which heuristic problem-solving programs improve their problem-solving expertise through experience, by generating selected problems in the domain, solving them, and learning by analyzing their solutions. As part of this research we have designed and constructed a computer program, called LEX, that incorporates general methods for discovering domain-dependent problem-solving heuristics.

The organization of this chapter is as follows. The learning problem considered by LEX is described, followed by a discussion of the methods employed by the current system. This includes methods for *(i)* solving practice problems, *(ii)* performing the *credit assignment* task of isolating appropriate and inappropriate search steps, *(iii)* proposing and generalizing heuristics, and *(iv)* generating new practice problems with which to experiment. The final sections of the chapter discuss augmenting the system by giving it knowledge to conduct detailed analyses of problem solutions. This knowledge can be used to provide strong guidance for the generalization process, and to generate new terms in the language with which heuristics are described. Some of the material from this chapter is drawn from a collection of previously published articles, including [Mitchell, 1981; Mitchell, 1982a; Mitchell, 1982b; Utgoff, 1982].

6.2 THE PROBLEM

LEX begins with a heuristic search problem solver without the heuristics. It is given a set of operators for solving problems in symbolic integration, and it learns a set of heuristics that recommend in which situations the various operators should be applied. Whereas each operator given to LEX contains a set of preconditions that characterize a class of problem states to which that operator *can* validly be applied, learned heuristics characterize the more restrictive subclass of problem states to which the operator *should* be applied; that is, the subclass of problem states for which application of the operator leads to an acceptable solution. Heuristics are learned by *generalizing from examples* of problem states to which the operator is applied in solving practice problems. These training examples are generated by the program, by proposing, solving, and analyzing practice problems.

LEX operates in the domain of symbolic integration. It solves integration

problems by searching through a space of mathematical expressions containing indefinite integrals. The operators for traversing the search space are the standard rules of integration (for instance, integration by parts) as well as transformations that characterize algebraic equivalence of expressions (such as the associative and distributive laws). The problem-solving goal is to derive a problem state that contains no integrals.

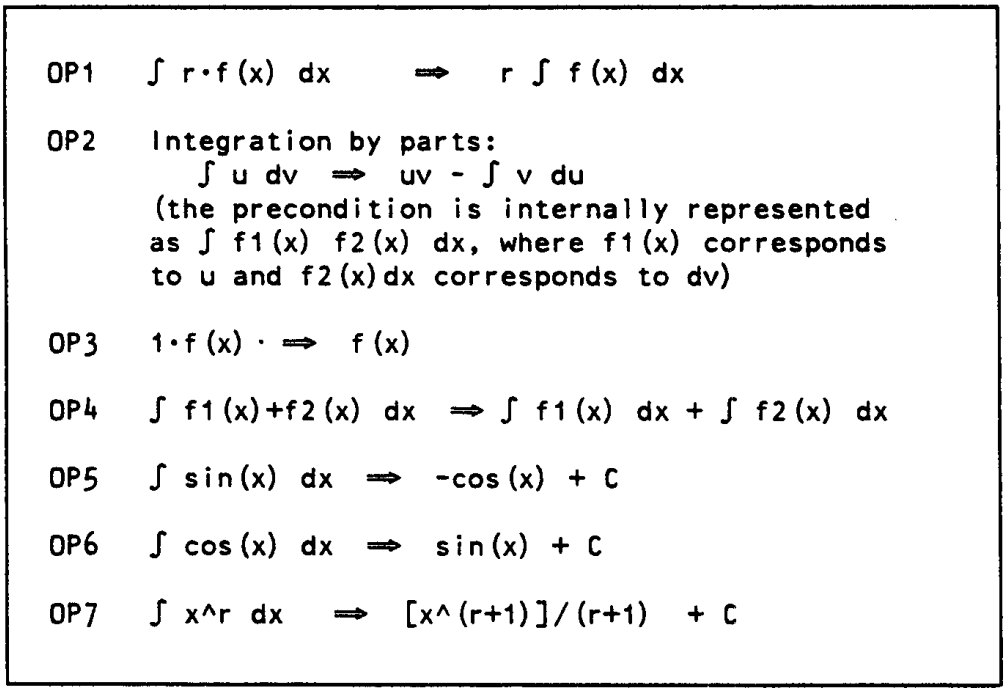

Figure 6-1: Some of the operators for symbolic integration.

Over 40 problem-solving operators are currently provided to LEX, some of which are shown in Figure 6-1. Each operator is interpreted as follows: If the general pattern on the left hand side of the operator is found within the problem state, then that pattern may be replaced by the pattern specified on the right hand side of the operator. For example, op1 indicates that if the problem state contains a subexpression of the form "$\int$ r·f(x) dx" (here "r" stands for any real number, and "f(x)" for any function of x), then that subexpression may be rewritten with the real number outside the integral.

In addition to its problem solver, representation for problem states, and problem-solving operators, LEX also begins with a language for describing heuristics. Each heuristic learned by LEX is of the form:

> IF the current problem state matches the applicability condition P,
> THEN apply operator O, with variable binding B.

Thus, the generalization task that LEX faces is that of determining the appropriate applicability condition, P, for each heuristic. Learning this applicability condition corresponds to learning the concept "situations in which operator O should be applied, with variable binding B."

The language for describing generalizations, or applicability conditions, of heuristics is based on a grammar for algebraic expressions containing indefinite integrals. The sentences derivable by this grammar are the expressions that form legal problem states. The sentential forms derivable by the grammar constitute legal generalizations. Briefly, the grammar contains non-terminal symbols that correspond to classes of functions (for example, trigonometric, polynomial) and classes of operators (such as function composition, multiplication, integration). These can be combined to form generalized algebraic expressions. Figure 6-2 shows this grammar in the form of a hierarchy. Each node in the hierarchy represents some substring of a sentential form, and each edge corresponds to a rule in the grammar.

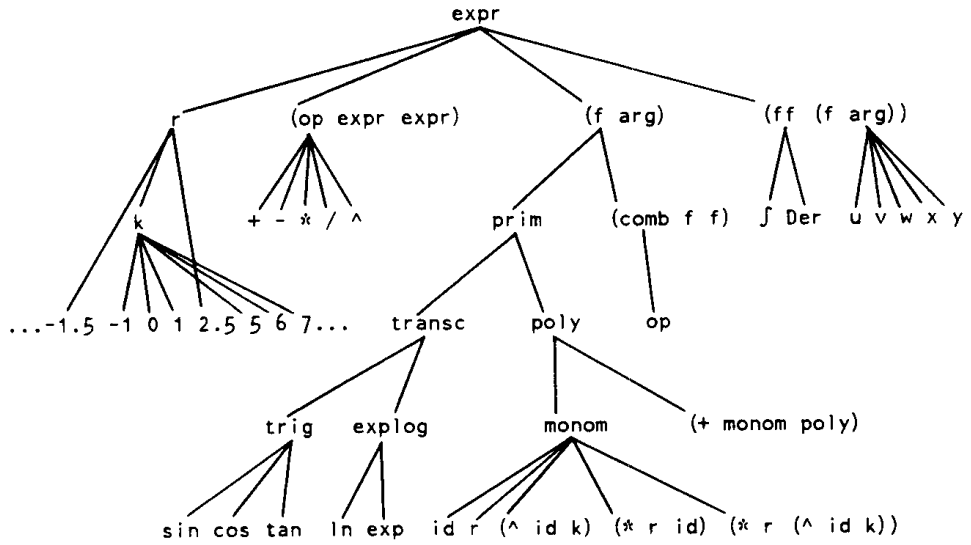

Figure 6-2: A grammar for a concept description language for symbolic integration.

Below is an example of the kind of heuristics that LEX can describe and learn. This heuristic may be interpreted as "*If* the current problem state contains an integrand which is the product of x and any transcendental function of x, *Then* try integration by parts, with u and dv bound to the indicated subexpressions."

$\int$ x transc(x) dx $\Rightarrow$ op2 (Integration by parts),
with u = x
and dv = transc(x) dx

The language used to describe applicability conditions of heuristics deter-

mines, to a great extent, the range of heuristics that can be learned by the system. In the current system, this language is fixed. Section 6.4 discusses an approach to dynamically altering the language when necessary.

6.3 DESIGN OF LEX

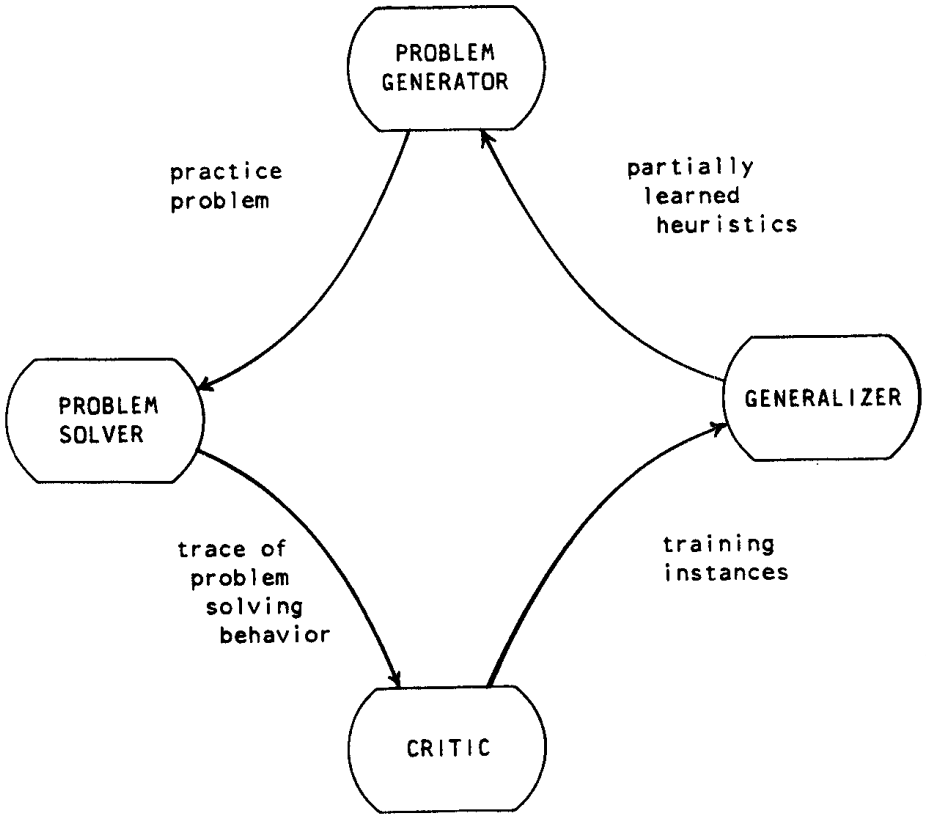

Figure 6-3: The major components of LEX.

LEX is based on four program modules, as shown in Figure 6-3. These modules are summarized below, and described in more detail in the following subsections.

1. **Problem Solver**— This module utilizes whatever operators and heuristics are currently available, to solve a given practice problem. The output of this module is a solution to the given problem, along with a detailed trace of the search performed in attempting to solve the problem.
2. **Critic**—This module analyzes the search performed by the Problem Solver. The output of this module is a set of positive and negative training instances from which heuristics will be inferred. Positive instances correspond to desirable search steps executed in solving the problem, whereas negative instances correspond to undesirable steps.

3. **Generalizer**—This module 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. **Problem Generator**—This module generates practice problems to be considered by the other modules. It attempts to generate practice problems that will be informative (that is, problems that will lead to training data useful for proposing and refining heuristics), yet easy enough to be solved using existing heuristics.

6.3.1 Representing Incompletely-learned Heuristics

LEX learns heuristics incrementally, requiring many positive and negative training instances before converging to a final definition of any given heuristic. Therefore, at any given stage in the system's development, there are typically many partially-learned heuristics whose exact description is underdetermined by the data, knowledge, and assumptions currently held by the system. It is essential that the system have a way of describing what the system *does* and *does not* know about each such partially-learned heuristic. This information is important *(i)* to the Problem Solver, which must use the partially-learned heuristics in trying to solve problems, *(ii)* to the Generalizer, which must revise partially-learned heuristics as new training data become available, and *(iii)* to the Problem Generator, which must choose practice problems that will lead to refinements of partially-learned heuristics.

LEX represents each partially-learned heuristic by representing the range of *all alternative plausible descriptions of the heuristic*. A description is considered plausible if it applies to all the known positive instances associated with the heuristic, but to none of the negative instances. Thus, for each partially-learned heuristic, we refer to the set of all plausible descriptions of the heuristic as the *version space* of the partially-learned heuristic, relative to the observed instances and the language in which heuristics are described.

While, in principle, the version space of a partially-learned heuristic could be represented by listing all of its members, there are typically far too many plausible descriptions of a heuristic for this to be feasible. Fortunately, a much more compact method for representing version spaces is possible. Any version space can be represented compactly by storing only its maximally-specific and maximally-general elements, according to the following definition of "more specific".

Heuristic H1 is **more specific than or equal to** heuristic H2 if and only if both of the following conditions hold:

1. The applicability condition of H2 matches every instance matched by the applicability condition of H1 (that is, the applicability condition of H1 is more specific than or equal to the applicability condition of H1).
2. In each case where both H1 and H2 apply, their recommendations are identical (that is, they recommend the same operator and the same binding of operator arguments).

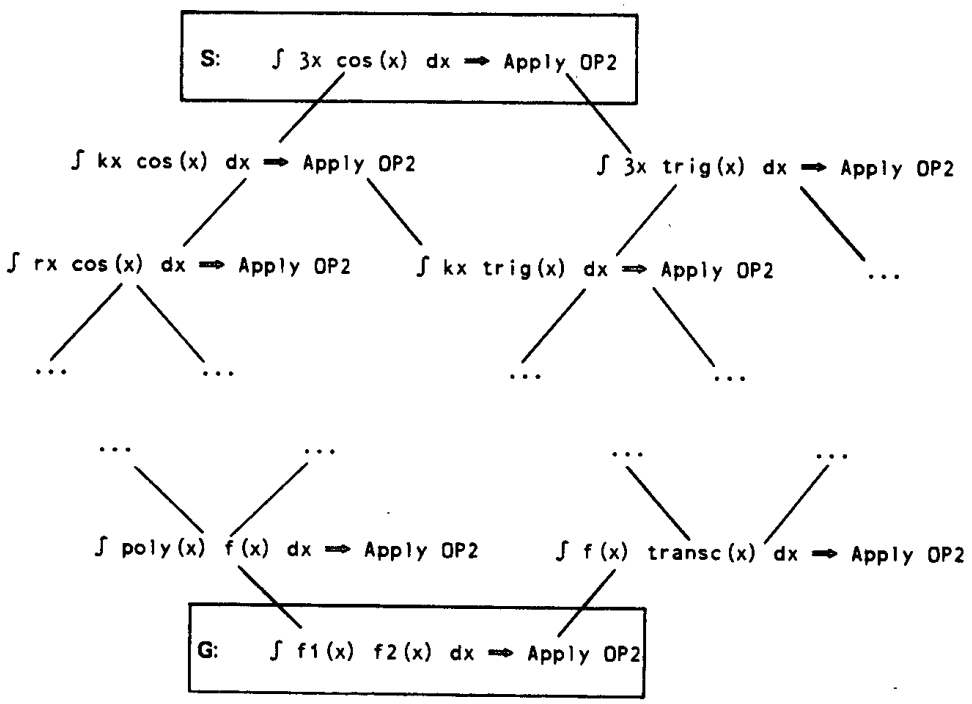

Figure 6-4: Representing a version space.

We will refer to the maximally-specific members of a version space as the subset S of the version space, and to the maximally-general (minimally-specific) members of the version space as the subset G. LEX represents the version space of each partially-learned heuristic by storing the subsets S and G of that version space, as illustrated in Figure 6-4. In this figure, some of the members of a particular version space are shown, with the more-specific-than relationship among them indicated. While there are very many plausible heuristic descriptions in this version space, the (singleton) sets S and G completely determine the version space by the following rule: a heuristic description is contained in the version space if and only if it is both *(i)* more specific than or equal to some member of G, and *(ii)* more general than or equal to some member of S.

This representation and use of version spaces for generalizing from ex-

amples has been used previously in the META-DENDRAL program for inferring rules of mass spectroscopy, and is described more fully in [Mitchell, 1978] and [Mitchell, 1982a]. In [Mitchell, 1978] a more formal definition of version spaces is given, along with proofs that the algorithm for incrementally updating the sets S and G is correct.

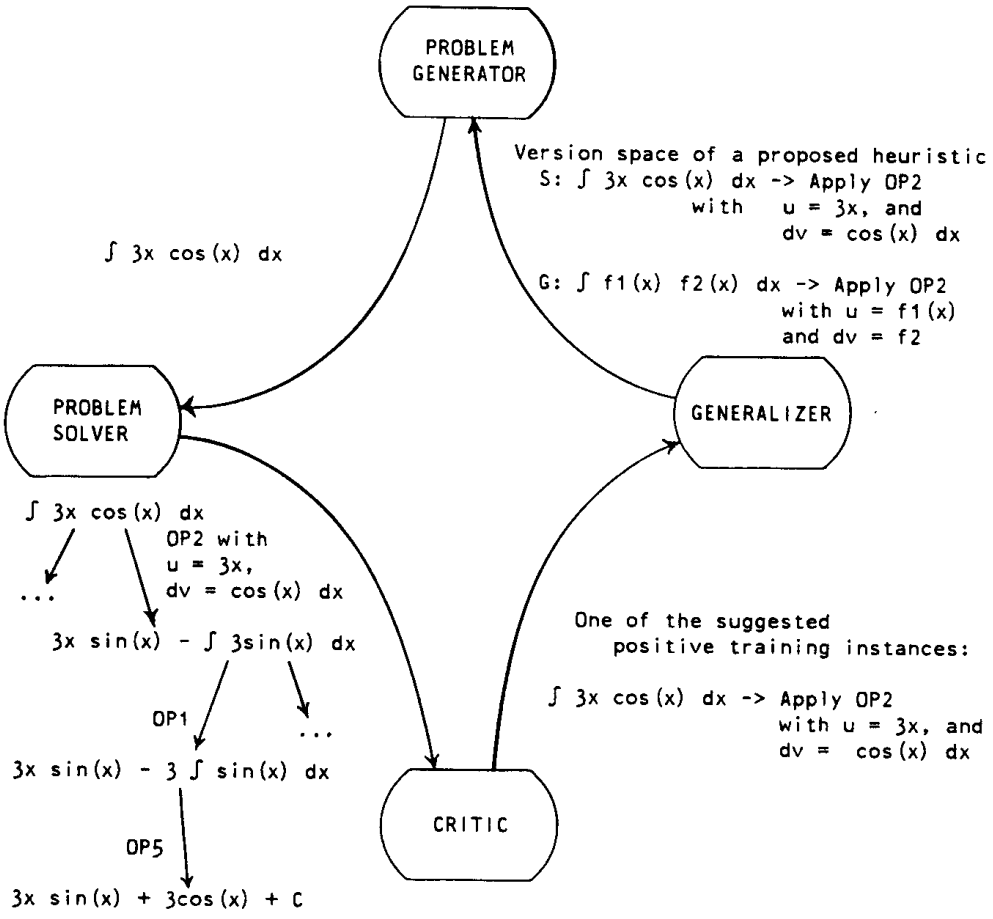

Figure 6-5: The learning cycle in LEX.

The remainder of this section presents the methods used by the four modules of LEX, in formulating and refining heuristics. The discussion centers around the example shown in Figure 6-5, which illustrates one particular practice problem considered by LEX, and the resulting version space of one heuristic. This figure shows the 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.

6.3.2 The Problem Solver

The Problem Solver uses a forward-search strategy guided by whatever heuristics are available during the current propose-solve-criticize-generalize cycle. 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. If the problem is not solved within the allocated resources, the Problem Solver stops and waits for a new problem.[1] Unsolved problems do not lead to any learning, because the credit assignment strategy of the Critic depends upon knowing the problem solution.

The Problem Solver generates a search tree, repeatedly choosing a node to expand and an operator with which to expand it, as shown below.

DO UNTIL problem is solved **OR** resource allocation is expended

BEGIN

IF no heuristics are applicable to any open node

THEN expand the lowest cost open node, using any applicable operator

ELSE IF exactly one heuristic applies to exactly one open node,

THEN execute the step recommended by that heuristic,

ELSE follow the recommendation of one of the applicable heuristics, choosing that heuristic which applies with the highest estimated degree of match (see explanation below).

END.

Here, the "cost" of a node refers to the sum of CPU time expended for each step leading from the root of the tree to that node. An open node refers to any node in the search tree with at least one applicable operator that has not yet been applied. The notion of "estimated degree of match" of a heuristic to a node is introduced to allow using partially-learned heuristics in a reasonable fashion. Notice that for a given partially-learned heuristic and search node, it is possible that some of the alternative plausible descriptions of the heuristic will match the node while others will not. Because of this we define the *degree of match* of a partially-learned heuristic to a given node as the proportion of the members of its version space that match the node. Because the degree of match is difficult to compute exactly, it is estimated by the proportion of members in the union of S and G that match the given problem state.

The ability of the Problem Solver to use partially-learned heuristics to con-

---

[1] LEX makes no distinction between problems that are unsolvable in principle, and those that are solvable in principle but unsolvable within the given resource limits.

trol search is important in allowing it to solve problems that will provide additional training data. In experiments with LEX, it has typically been the case that the majority of available heuristics are only partially learned. Even so, it is quite common that a partially-learned heuristic will apply to a particular node with a degree of match of 1. In such cases, even though the exact identity of the heuristic is not yet determined, the applicability of the heuristic to this particular node is fully determined (that is, it does not matter which of the alternative heuristic descriptions is correct, since they all apply to the node in question). The ability to distinguish such cases from those in which there is ambiguity regarding the heuristic recommendation is an important capability in the Problem Solver's use of partially-learned heuristics.

6.3.3 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. The Critic examines the detailed search trace recorded by the Problem Solver, and selects certain search steps to be classified as positive or negative training instances for forming general heuristics. Each training instance corresponds to a single search step; that is, the application of a single operator to a given problem state, with a particular binding of operator arguments.

Figure 6-5 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 there corresponds to the first step along the path to the solution.

The criterion used by the Critic to produce training instances may be summarized as follows:

1. The Critic labels as a *positive instance* every search step along the lowest cost solution path found. Here, the cost of a solution is taken to be the sum of the execution times of all operators applied along the solution path.
2. The Critic labels as a *negative instance* every search step that *(i)* leads away from a node on the lowest cost solution path found, to a node not on this path, and *(ii)* when its resulting problem state is given anew to the Problem Solver, leads either to no solution or to a higher cost solution. Here a solution is considered higher cost if its cost is more than a certain factor times the cost of the lowest cost known solution (currently this factor is set to 1.15). The resource allocation given to the Problem Solver in this case is equal to the resources spent in obtaining the known solution.

Notice that the Critic is not infallible. It is possible for the Critic to produce positive training instances that are not on the minimum cost solution path, but are rather on the lowest cost solution path found by the Problem Solver. Also, it is possible for the Critic to label as negative a search step that is in fact part of the true (but never discovered) minimum cost solution path. Both

kinds of errors can arise because the heuristic Problem Solver is not assured of finding the minimum cost solution. Criterion 2*(ii)* above is included in order to reduce the likelihood that such errors will occur. Here, the Critic reinvokes the Problem Solver, giving it a problem state associated with a potential negative instance, in order to explore a portion of the problem space that may not have been sufficiently considered during the solution of the original problem. If the Problem Solver is unable to find an appropriate solution from the given state within the specified resource limits, the confidence that this is a negative instance is increased. If the Problem Solver finds a lower cost solution when it is reinvoked, this new solution is used in determining positive training instances. Of course, the only completely error-free strategy for labeling training instances requires a full breadth-first or uniform-cost search, which is usually prohibitively time consuming.

The Critic typically produces between two and twenty training instances from each solved problem, depending upon the length of the problem solution and the branching factor of the search (the search trees produced by the Problem Solver typically contain from a few to a few hundred search nodes). We have found empirically that even though the Critic cannot guarantee correct classifications, it rarely produces incorrect training instances. We have also found that in a significant number of cases, when the Critic calls the Problem Solver to consider a possible negative instance (see criterion 2*(ii)* above) an improved solution is found. For example, in one run of LEX for a sequence of 12 training problems, this occurred 4 times. In those cases in which the Problem Solver does not find the best solution during its first attempt, the cause is usually a misleading recommendation by an incompletely-learned heuristic.

6.3.4 The Generalizer

The Generalizer considers the positive and negative training instances supplied by the Critic within the current learning cycle, in order to propose and refine heuristics to improve problem-solving performance. The generalization problem faced by this module is one of learning from examples. Given a sequence of training instances corresponding to search steps involving a given operator, the generalization problem here is to infer the general class of problem states for which this operator will be useful, along with the range of appropriate bindings for operator variables.

The Generalizer describes the version space for each proposed heuristic, by computing the sets S and G that delimit the plausible versions of that heuristic. For example, Figure 6-5 shows a positive training instance associated with op2 as input to the Generalizer. The output of the Generalizer in this case is a version space corresponding to a partially-learned heuristic, and represented by the (singleton) sets S and G shown in Figure 6-5. This partially-learned heuristic is proposed on the basis of the single training instance shown, and will be refined as subsequent instances become available. Below, we describe the procedures for proposing and refining problem-solving heuristics in LEX.

Proposing a new heuristic—When the Generalizer is given a new positive instance, it determines whether any member of the version space of any current heuristic applies to this instance. If not, a new heuristic is formed to cover the positive instance. This is the case in the example of Figure 6-5. In forming a new 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). 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 being recommended. Thus, in the example of Figure 6-5, G is initialized to the version whose precondition is the precondition for op2. Here, $\int$ f1(x) f2(x) dx represents the integral of the product of any two real functions of x, and corresponds to the precondition $\int$ u dv as it is stated in the system's generalization language.

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. 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.

Refining incompletely-learned heuristics—If the Generalizer finds that an existing heuristic applies to a newly-presented positive or negative instance (that is, if its degree of match to the instance is nonzero), then that heuristic is revised by eliminating from its version space any version that is inconsistent with this training instance. In the current example, the next practice problem that is considered is $\int$ 3x sin(x) dx (the following section explains why). The solution to this problem leads to both a positive and a negative training instance for the heuristic from Figure 6-5. Figure 6-6 shows these two new training instances, and the way in which they lead to a refinement of the version space of this heuristic. In the revised version space shown there, the most specific version, S, of the heuristic has been generalized just enough to allow it to apply to the new positive training instance. Here 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. It then consults the grammar for expressing heuristics (shown in Figure 6-2) to determine the next more

```
Version Space of Heuristic

        S:  ∫  3x cos(x) dx -> Apply OP2 with
                               u = 3x, and
                               dv = cos(x) dx

        G:  ∫  f1(x) f2(x) dx -> Apply OP2 with
                               u = f1(x), and
                               dv = f2(x)dx
```

```
New Training Instances:

    Positive training instance:

                ∫  3x sin(x) dx -> Apply OP2 with
                                   u = 3x, and
                                   dv = sin(x) dx

    Negative training instance:

                ∫  3x sin(x) dx -> Apply OP2 with
                                   u = sin(x), and
                                   dv = 3x dx
```

```
Revised Version Space:

        S:   ∫ 3x trig(x) dx -> Apply OP2 with
                                u = 3x, and
                                dv = trig(x) dx

        G:
          g1:  ∫ poly(x) f2(x) dx -> Apply OP2 with
                                     u = poly(x), and
                                     dv = f2(x) dx

          g2:  ∫ f1(x) transc(x) dx -> Apply OP2
                                     with u = f1(x), and
                                     dv = transc(x) dx
```

Figure 6-6: Revising the version space of a heuristic.

general term that can be substituted in order to include this new instance.[2]

The general boundary of the revised version space of Figure 6-6 has also been altered so that it does not apply to the new negative training instance. In this case, there are two maximally-general versions (g1 and g2) of the heuristic consistent with the three observed training instances. Here, "poly(x)" refers to any polynomial function of x, and "transc(x)" denotes any transcendental function of x. As with revising the set S, revisions to G depend upon the generalization language being used. For instance, g1 is computed by replacing f1(x) (which represents "any real-valued function") by the next more specific acceptable expression. Notice in the hierarchy of Figure 6-2, this expression is "poly".

As subsequent training instances are considered, this partially-learned heuristic is further refined, and S and G converge to the heuristic description shown below. Notice that this description is contained in the version space represented in Figure 6-6, since it is more general than the S boundary set and more specific than the G boundary set of the version space.

$\int$ rx transc(x) dx $\Rightarrow$ apply op2 with u = rx, and dv = transc(x) dx

Although the Generalizer attempts to form a single conjunctive heuristic for each operator known to the system, sometimes it is not possible to cover all the positive instances and exclude all the negative instances with a single conjunctive generalization. The Generalizer deals with learning disjunctions in the following straightforward manner: if a positive instance associated with operator O is not consistent with any current heuristic that recommends operator O, then it proposes a new heuristic (that is, disjunct) for operator O that covers this instance. This new heuristic will be updated by all subsequent negative instances associated with operator O, and by any subsequent positive instances associated with operator O and to which at least some member of its version space applies. This technique for learning disjunctive concepts is similar to several described previously (for example, [Mitchell, 1978; Iba, 1979; Vere, 1978]).

How effective is the Generalizer at producing useful heuristics? One way to answer this question is to measure the improvement in problem-solving performance due to learned heuristics. In one experiment that illustrates typical behavior of LEX, a sequence of twelve hand-selected[3] training problems was presented to the Problem Solver, Critic, and Generalizer, and performance of the Problem Solver was measured at various stages in the training sequence. Perfor-

---

[2]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.

[3]At the time that this experiment was conducted, we had not implemented the Problem Generator module.

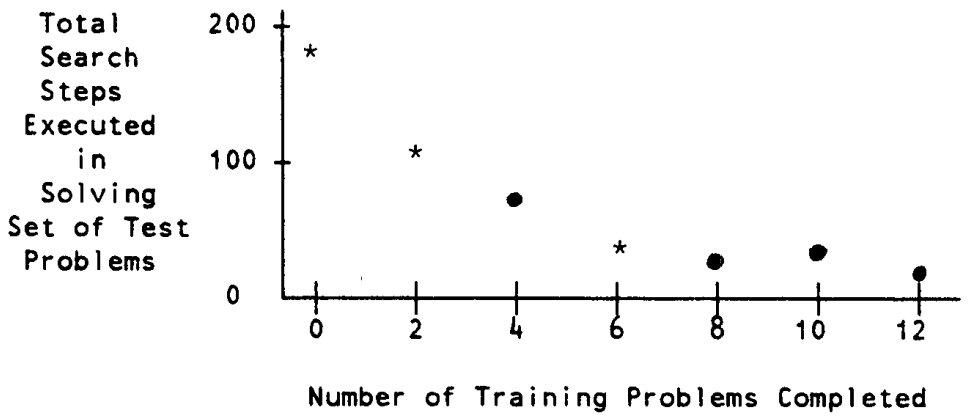

Figure 6-7: Performance Results

mance was measured by testing the Problem Solver on a set of five test problems before any training had occurred, and again after every second training problem. The five test problems were different from the set of twelve training problems, though the two sets were chosen to be similar enough that learned heuristics would be relevant to the test problems. This experiment is reported in greater detail in [Mitchell, 1981], and is summarized in Figure 6-7.

Fourteen heuristics were formed by LEX during this training session, covering thirteen of the 32 operators available to the system at that time. Twelve of these fourteen heuristics remained incompletely learned at the end of the training sequence (that is, their version space still contained multiple plausible descriptions of the heuristic).

Figure 6-7 shows the improvement in problem-solving performance (roughly two orders of magnitude) for this experiment, as measured by the total number of search steps required in attempting to solve the five test problems. At certain points during the training, the Problem Solver could not solve all five test problems within the given resource allocation.[4] Such points are shown as a "*" in Figure 6-7, and the number of search steps recorded in those cases is the number of steps executed before the solution attempt was aborted. While the exact values of the points on this curve would be different for different sets of training and test problems, the general form of the curve is quite repeatable, given reasonable test problems and a well-chosen sequence of training problems.

In addition to observing that problem-solving performance improved significantly using the learned heuristics, it is interesting to note that problem-solving performance did not improve monotonically as a function of training. In particular, while all five test problems could be solved following the fourth train-

[4]The Problem Solver was allowed four CPU minutes and 800,000 cons cells per test problem, running in RUCILISP on a DEC2060.

ing problem, only four of the test problems could be solved after the sixth training problem. This phenomenon was due to the proposal of new, partially-learned heuristics that led the Problem Solver to consider new (and not very useful) branches of the search in one of the test problems. Subsequent training refined these heuristics and the Problem Solver became able again to solve (this time more efficiently) all five test problems by the completion of the eighth training problem.

6.3.5 The Problem Generator

After 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. Its task is very different from that of a teacher of symbolic integration, or an outside trainer in most work on learning from examples. In contrast to an expert teacher, the Problem Generator must choose appropriate practice problems *without knowing* the heuristics that it is trying to teach. While the Problem Generator lacks this important information, it has other information that an expert teacher may not have: very detailed knowledge about the learner's current state (including knowledge of alternative versions of heuristics under consideration). As a result of these characteristics, the experimentation strategy of the Problem Generator is based primarily on generating problems designed to eliminate known ambiguities in LEX's heuristic knowledge.

The major criteria for generating problems are *(i)* to generate training problems whose solutions will provide informative new training data, and *(ii)* to generate training problems that can be solved using the available operators and current set of heuristics. The current implementation of the Problem Generator is based mainly on the first of these considerations, and consists of two different problem generation tactics.

The first problem generation tactic is to produce problems that will allow refinement of existing, partially-learned heuristics. This is done 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 at the top of Figure 6-6. 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. As it turns out, this problem leads to both a positive and a negative instance (corresponding to different bindings of operator arguments), and both version space boundaries are refined as shown in Figure 6-6.

How does the Problem Generator create a problem that matches part of a given version space? It begins by selecting a single member, s1, of the S boundary, and a more general member, g1, of the G boundary. (In the version space at the top of Figure 6-6 both boundary sets happen to be singleton sets.) It then creates, as follows, a problem state that matches g1, but does not match s1. One term in the generalization s1 is selected (in this case cos(x)), and the corresponding term in g1 is found (in this case f2(x)). The generalization hierarchy (see Figure 6-2) is then examined to determine a sibling of the term from s1, that is more specific than the corresponding term from g1. In this case, sin(x) is a sibling of cos(x) that is more specific than f2(x). This sibling is then substituted into s1, and the resulting generalization is fully instantiated to produce a problem state that matches g1, but not the original s1. In the current example, this leads to the problem state $\int 3x \sin(x)\, dx$. Notice that if the term 3x were chosen, rather than cos(x), as the basis for forming a new problem state, the new problem might instead be $\int 7x \cos(x)\, dx$. Furthermore, both of these terms could be replaced to produce the problem state $\int 7x \sin(x)\, dx$. Because of the need to create a problem that can be solved, the Problem Generator attempts to create a problem that is very similar to the most recently encountered positive instance for the heuristic. Therefore, only a single term from s1 is altered, and the resulting generalization is instantiated to correspond as closely as possible to the most recently encountered positive instance (a known solvable problem).

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 intersect, but for which there is no current heuristic. Should a problem be encountered for which both operators apply, a heuristic will be needed to choose which of the two to apply. For example, consider op1 and op3 from Figure 6-1. The intersection of the preconditions of these operators is $\int 1 \cdot f(x)\, dx$; that is, both op1 and op3 will apply to any problem that matches this applicability condition. This applicability condition is therefore instantiated to produce a specific problem state (such as $\int 1 \cdot \cos(x)\, dx$) which is then output by the Problem Generator. When the Problem Solver, Critic, and Generalizer consider this problem, a new heuristic will be proposed which will be useful in selecting between op1 and op3 in cases where they are both applicable.

The current Problem Generator incorporates the above two tactics for creating practice problems, and can employ any of several strategies for determining which tactic to apply at any given step. One such experimentation strategy is to apply the first tactic (refine an existing heuristic) whenever possible, and to apply the second tactic only when the first cannot be applied (for example, when the system begins operation and has no heuristics at all). While we have not yet done extensive testing of this module, it has been used to generate sequences of practice problems that lead to useful heuristics. The main observations that have come out of our preliminary experiments with this module are given below.

- It will be useful to extend the other system modules so that they can take

into account the reason why the current problem has been suggested, and focus their activity accordingly. For example, if a problem is suggested in order to refine a particular heuristic, then the Problem Solver and Critic should be sure to consider the search steps that become training instances for that heuristic, and the Critic might allocate greater resources to obtain a reliable classification of that training instance.

- While the tactics described above are generally successful at creating informative problems to consider, they are not always successful at creating *solvable* problems. Some problems that are generated are simply not solvable with the set of operators known to the system. Other generated problems are solvable in principle, but cannot be solved within the allocated CPU time and space resources, using existing heuristics. In our initial experiments, more than half the generated problems turned out to be solved by the Problem Solver. Both of the current tactics produce a generalization which can be instantiated in any fashion to produce an informative problem. The instantiation is then controlled by a single heuristic: try to create a problem state that is as similar as possible to a previously-solved problem. More reliable methods for creating solvable instances of problems may require that the system have (or acquire) more appropriate knowledge about the characteristics of solvable problems.
- It may be useful to introduce a new tactic that produces problems that are guaranteed to be solvable, by beginning with a goal state, then applying inverses of the known operators to produce a problem state with a known solution. While the solution produced along with the problem will not necessarily be the optimal solution, it will provide an upper bound on the cost of the optimal solution. For this tactic to be useful, there must be a way of selecting sequences of operators that produce informative as well as solvable problems.
- There are also interesting questions to be considered regarding global strategies for exploring the problem domain. For example, should the Problem Generator focus first on refining existing heuristics, and then suggest problems that lead to new heuristics? Or is it better to build up a more broad set of heuristics, focusing at each step on problem types for which no heuristics yet exist, leaving refinement of these heuristics until a broad set of incompletely-determined heuristics are proposed?

6.4 NEW DIRECTIONS: ADDING KNOWLEDGE TO AUGMENT LEARNING

The current LEX system, as described in the previous section, is able to learn useful problem-solving heuristics in the domain of symbolic integration, by generalizing from self-generated examples. There are several features of the design of LEX that have an important impact on its capabilities. The ability to represent incompletely-learned heuristics is crucial; to the Problem Solver that

must use these partially-learned heuristics in order to solve additional practice problems to obtain additional training data; to the Generalizer that must refine these heuristics; and to the Problem Generator that must be able to consider alternative plausible descriptions of a heuristic in order to suggest an informative practice problem. The ability of the Critic to produce reliable training instances is also crucial to system performance. In spite of the heuristic nature of the Critic's credit assignment method (following from the fact that only part of the search space is explored by the Problem Solver), the Critic in fact performs quite well in producing reliable classifications of training instances. Its ability to call the Problem Solver in a controlled manner to explore selected portions of the search space is important to increasing the reliability of its classifications of training instances. The Generalizer's use of the version space method for generalizing from examples is also a major feature of LEX, which gives it the capability to incrementally converge on heuristics consistent with a sequence of training instances observed over the course of many practice problems.

While LEX is able to learn useful heuristics, it also has significant limitations. One of the most fundamental difficulties is that learning is strongly tied to the language used to describe heuristics—the system can only learn heuristics that it can represent in the provided language. It is difficult to manually select an appropriate language before learning occurs, and LEX often fails to converge on an acceptable heuristic for a given set of training instances, simply because it does not have the appropriate vocabulary for stating the heuristic. For example, we have found that the addition of terms such as "odd integer" and "twice integrable function" to the language shown in Figure 6-2, would allow LEX to describe (and therefore learn) heuristics that it cannot currently represent. This constraint imposed by a fixed representation language is one of the most fundamental difficulties associated with this and some other approaches to learning from examples.

A second deficiency of LEX is its failure to take advantage of an important source of information for chosing an appropriate generalization: analysis of *why* a particular search step was useful in the context of the overall problem solution. By analyzing the role of a particular search step in leading to a problem solution, it is sometimes possible for humans to determine a very good general heuristic after observing only a single training instance. If LEX were to conduct such an analysis, it would converge much more quickly on appropriate heuristics, possibly with less sensitivity to classification errors by the Critic.

In this section, we describe our current research toward giving LEX new knowledge and reasoning capabilities to overcome the above limitations. In particular, we consider how knowledge about heuristic search and about the intended purpose of learned heuristics could allow LEX to *(i)* derive justifiable generalizations of heuristics via analysis of individual training instances, and *(ii)* respond to situations in which the vocabulary for describing heuristics is insufficient to characterize a given set of training instances. More detailed discussions of this material can be found in [Mitchell, 1982b] and [Utgoff, 1982]. The

kind of knowledge considered here, regarding the intended purpose of learned heuristics, is one kind of meta-knowledge that can be useful in acquiring problem-solving strategies. The importance of meta-knowledge in acquiring problem-solving strategies is also discussed in other chapters of this book, such as Chapters 9 and 12.

6.4.1 Describing the Learner's Goal

In order to reason about *why* a given training instance is positive, and to determine which features of the training instance are relevant, it is necessary that the system have a definition of the criterion by which the instance is labeled as positive (that is, the criterion that determines the goal of its learning activity). LEX is intended to learn heuristics that lead the Problem Solver to minimum cost solutions of symbolic integration problems. This goal is implicit in the credit assignment procedure used by the Critic, which attempts to classify individual search steps as positive or negative according to this criterion. While this criterion is currently defined procedurally within the Critic, it is not defined declaratively, and the system therefore cannot reason symbolically about its learning goal. Here we present a declarative representation of this credit assignment criterion, then discuss in subsequent subsections how this knowledge provides the starting point for analyzing training instances, and extending the vocabulary of the language for describing heuristics.

To simplify the examples and discussion here, we assume a slightly modified credit assignment criterion, for which the goal of LEX is to learn heuristics that recommend problem-solving steps that lead to *any* solution (rather than the minimum cost solution). In this case, any search step that applies some operator, *op*, to some problem state, *state*, is a positive instance, provided it satisfies the predicate PosInst defined as follows:

PosInst(op, state) $\Leftrightarrow$
  $\sim$Goal(state) $\wedge$ [Goal(Apply(op, state)) $\vee$ Solvable(Apply(op, state))].

Here, Goal is the predicate for recognizing solution states, Apply is the function for applying 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 is defined as follows:

Solvable(state) $\Leftrightarrow$
  ($\exists$ op) [Goal(Apply(op, state)) $\vee$ Solvable(Apply(op,state))]

6.4.2 Analyzing Training Instances to Guide Generalization

This section suggests how the declarative representation of the credit assignment criterion, PosInst, could be used by LEX to produce a justifiable generalization of a heuristic based on analysis of a single training instance. The key idea here is that by analyzing *why* the observed positive instance is classified as positive, in the context of the overall problem solution, it is possible to deter-

mine a logically sufficient condition for satisfying PosInst. Such an analysis leads to a *justifiable* generalization of the heuristic, that follows from the credit assignment criterion, together with knowledge about search and the representation of operators and problem states. This process is related to the process of operationalizing advice, as discussed by Mostow in Chapter 12 of this book and by [Hayes-Roth, 1980]. The particular method for analyzing solution traces is a generalization of the method of solution analysis presented in [Fikes *et al.*, 1972].

As an example, suppose that the system has just produced the problem solution tree shown in Figure 6-8, and the generalizer is now considering the first step along the solution path as a positive training instance for a heuristic that is to recommend op1. Assuming no heuristic yet exists for op1, the empirical generalization method described earlier will produce the following version space for the new heuristic:

S: $\int 7\ (x^2)\ dx \Rightarrow$ use op1

G: $\int r\ f(x)\ dx \Rightarrow$ use op1

In this example, analysis of how this training instance satisfies the credit assignment criterion will lead to additional information for refining the above version space of alternative hypotheses. The trace of this analysis is broken into four main stages, which attempt to determine some property of the integrand in the training instance which is *sufficient* to assure that the credit assignment criteria will be met. This sufficient condition for satisfying PosInst can then be used to further generalize the S boundary of the version space for this heuristic. The four main stages are *(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 (that is, the language of applicability conditions for 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.

Stage 1: Produce an explanation of how the current training instance satisfies PosInst. This explanation is produced by instantiating the definition of PosInst for the positive instance in question. By determining which disjunctive clauses in the definition of PosInst are satisfied by the current training instance, and then by further expanding those clauses by instantiating predicates to which they refer, a proof is produced that PosInst(op1, State1). The result of this stage is an And/Or proof tree, which we shall call the *explanation tree* for the training instance. The tip nodes in the explanation tree are known to be satisfied because of the observed solution tree to which the training instance belongs. This explanation tree indicates how the training instance satisfies PosInst, and forms the basis for generalization by inferring sufficient conditions for satisfying PosInst.

The explanation tree for the positive training instance <op1, State1> is

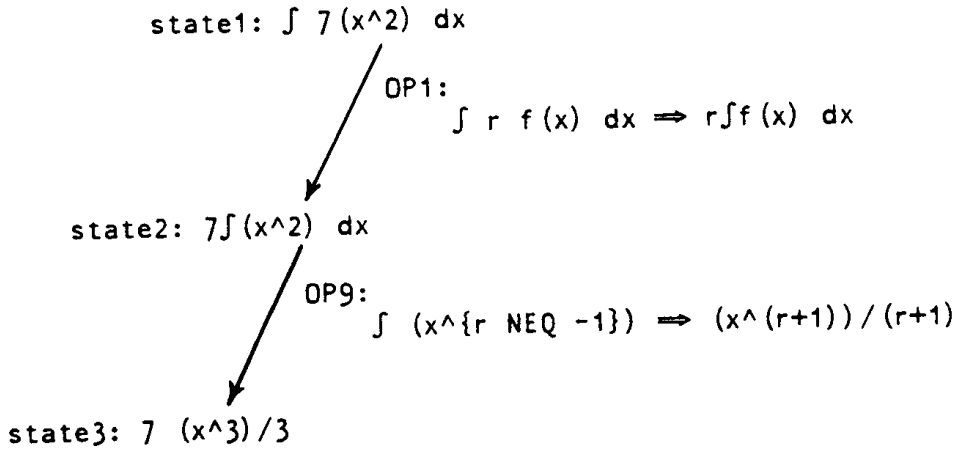

Figure 6-8: The solution tree for example 1.

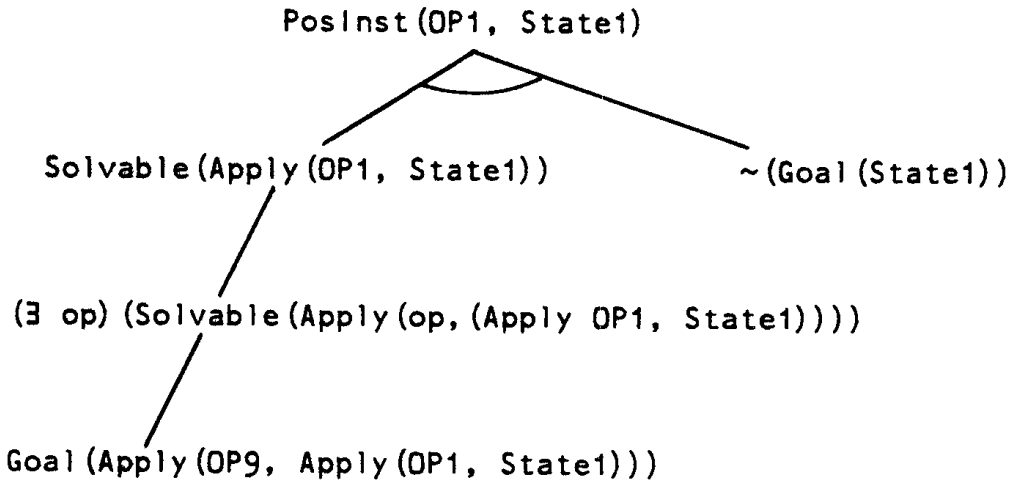

Figure 6-9: The explanation tree for PosInst(op1, State1).

shown in Figure 6-9. Nodes in the *explanation tree* correspond to statements about various problem states and operators in the associated *solution tree*. The explanation tree for the current example indicates that <op1, State1> is a Positive instance because *(i)* State1 is not a Goal state, and *(ii)* by applying op9 to the state resulting from the positive instance step, it is possible to reach a goal state. Subsequent stages of analysis of this explanation tree, shown below, extract this explanation (at an appropriate level of generality), and to restate it in the generalization language in which heuristics are expressed.

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 it is clear that any set of nodes that satisfy this And/Or tree correspond to a sufficient condition for satisfying PosInst. In the current example, for instance, if all the tip nodes of the explanation tree are satisfied by a given state, s, then PosInst will be satisfied by the training instance <op1, s>. In this stage, a set of nodes that satisfy the And/Or tree is selected, and the corresponding sufficient condition for PosInst is formulated by replacing the problem state from the training instance by a universally-quantified variable. In the current example, if the tip nodes of the explanation tree are selected, then the resulting sufficient condition for PosInst may be stated as follows:

$$(\forall s)\ \text{PosInst}(\text{op1}, s) \Leftarrow (\sim\text{Goal}(s) \wedge \text{Goal}(\text{Apply}(\text{op9}, \text{Apply}(\text{op1}, s))))$$

Notice that there are many possible choices of sets of nodes to satisfy the And/Or tree, and correspondingly many sufficient conditions. This choice of nodes is one of the major control issues in the analysis of the training instance. Generally, nodes close to the root of the explanation tree lead to more general sufficient conditions. However, since the sufficient conditions formulated in this stage must be transformed by subsequent stages to statements in the generalization language for heuristics, the choice of covering nodes from the explanation tree must trade off *(i)* the generality of the corresponding sufficient condition, with *(ii)* the loss in generality that is likely when this sufficient condition is transformed into the generalization language for heuristics. As an example, consider the alternative choice of the two nodes at the second level of the explanation tree. This set of nodes leads to the following sufficient condition for PosInst:

$$(\forall s)\ \text{PosInst}(\text{op1}, s) \Leftarrow (\sim\text{Goal}(s) \wedge \text{Solvable}(\text{Apply}(\text{op1}, s)))$$

While this sufficient condition on satisfying PosInst is more general than the earlier sufficient condition, it turns out that this added generality will be lost when attempting to redescribe the sufficient condition in terms of the generalization language. The difficulty in this case stems from the fact that there is no straightforward translation from the predicate "Solvable" to a statement in the generalization language of LEX. In contrast, the sufficient condition corresponding to the tip nodes of the explanation tree involves only the predicate "Goal", which is easily characterized in terms of the generalization language.

Stage 3: Restate the sufficient condition in terms of the generalization language, as restrictions on various problem states involved in the solution tree. In the current example, the sufficient condition corresponding to the tip nodes of the explanation tree can be restated as follows:

$$(\forall s)\ \text{PosInst}(\text{op1}, s) \Leftarrow$$
$$(\text{Match}(\textstyle\int f(x)dx, s) \wedge \text{Match}(f(x), \text{Apply}(\text{op9}, \text{Apply}(\text{op1}, s))))$$

The predicate "Match" corresponds to the matching procedure used to compare applicability conditions, or generalizations, with problem states (that is, it tests whether the applicability conditions are satisfied in the problem state). The first conjunct above expresses the fact that "s" is *not* a Goal state ("s" contains

an integral), and the second conjunct expresses the fact that Apply(op9, Apply(op1, s)) *is* a goal state (it is some expression that does not contain an integral sign). This second conjunct corresponds to a restriction on the state labeled State3 in Figure 6-8.

In general, the goal of this stage is to translate the sufficient condition into a conjunctive set of statements of the form Match(<generalization>, <problem-state>), where <generalization> can be any statement in the generalization language used by the system, and <problem-state> can be any expression that corresponds to a particular problem state in the solution tree for the current example.

The translation of sufficient conditions into the generalization language requires knowledge about the correspondence between the representation language in which the analysis is being done, and the generalization language used to describe heuristics. For instance, in the current example the following knowledge is used in the translation:

$(\forall s)\ \sim Goal(s) \Leftrightarrow Match(\int f(x)dx, s)$
and
$(\forall s)\ Goal(s) \Leftrightarrow Match(f(x)dx, s)$

Unfortunately, some expressions generated by analyzing the explanation tree may have no corresponding expression in the generalization language. For example, in the current LEX generalization language, there is no way of characterizing all "Solvable" functions. In this case, translating the sufficient condition corresponding to the second level nodes in the explanation tree may require further specializing the sufficient condition, by replacing Solvable(x) by sufficient conditions for Solvable. An example of such knowledge is the knowledge that all polynomial integrands are solvable. It is important to note that even if no such knowledge is available, it will always be possible to translate the sufficient condition into some weaker condition describable in the generalization language. This can always be accomplished by using the fact that the solution tree provides at least one problem state which satisfies the predicate, and the problem state is itself describable in the generalization language. Thus, for example, the condition Solvable(Apply(op1, s)) may, if no other relevant knowledge is available, be weakened and replaced by Match($7\int(x^2)dx$, Apply(op1, s)).

Stage 4: Propagate the restrictions on various problem states through the solution tree to determine equivalent conditions on the problem state involved in the current training instance. By examining the definitions of the operators involved in reaching a given state, x, it is possible to propagate restrictions on x 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. This propagation requires using the operators in a way different from the way in which they are used during forward search problem-solving, and is similar to the process of goal regression discussed in the literature on means-ends problem-solving and planning [Nilsson, 1980].

As an example, consider the second expression in the sufficient condition from stage 3: Match(f(x), Apply(op9, Apply(op1, s))). This condition, when back propagated through op9 becomes Match($f(x)\int(x \uparrow (r \neq -1)dx)$, Apply(op1, s)). The new generalization corresponds to the class of problem states which can be transformed using op9 into an expression that satisfies the original condition. Similarly, this new expression can be propagated back through op1 to yield an equivalent condition on State1: Match($\int r(x \uparrow \{r \neq -1\})dx$, s). Thus, the sufficient condition from stage 3 can be restated as:

$(\forall s)$ PosInst(op1, s) $\Leftarrow$
(Match($\int f(x)dx$, s) $\wedge$ Match($\int r(x \uparrow \{r \neq -1\})dx$, s))

Since the second conjunct is more specific than the first, the above expression can be simplified to:

$(\forall s)$ [PosInst(op1, s) $\Leftarrow$ Match($\int r(x \uparrow \{r \neq -1\})dx$, s)]

Finally, we have found sufficient conditions for PosInst(op1, s) which are stated as a generalization that must match State1. While the sufficient condition determined by the above analysis is not the most general sufficient condition possible, it is satisfied by the current training instance and follows naturally from analyzing that instance. If this training instance were the first instance encountered for this particular heuristic, the resulting version space would reflect the extra information extracted from analyzing this instance, as shown below.

S: $\int r\,[x \uparrow (r \neq -1)]\,dx \Rightarrow$ Apply op1

G: $\int r\,f(x)\,dx \Rightarrow$ Apply op1

6.4.3 Automatically Extending the Vocabulary for Describing Heuristics

One of the most fundamental difficulties associated with current approaches to machine learning is the problem of acquiring an appropriate vocabulary with which to describe learned concepts. Nearly all existing systems assume some fixed vocabulary of terms with which to represent learned concepts (for instance, the LEX terms trigonometric, polynomial, exponential, and so on, as shown in Figure 6-2). In cases where this vocabulary is inappropriate, it will be impossible to describe (and hence to learn) the desired concept. In the LEX system, we have found that there are many cases where the current language for describing heuristics is insufficient to correctly characterize sets of training instances produced by the Critic.

As an example, consider the solution path shown in Figure 6-10, 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 $\int\cos^7(x)dx$, 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 the language in Figure 6-2 for describing heuristics has no term that includes both 5 and 7 while excluding 6.

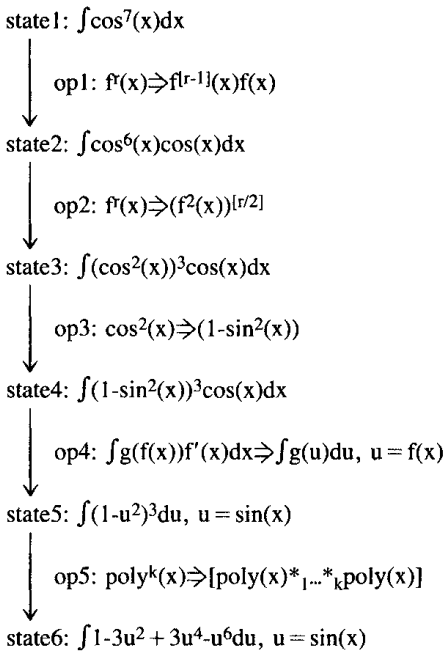

Figure 6-10: Solution path for $\int \cos^7(x)dx$.

In this case, a solution analysis similar to that described in the previous section can lead to the generation of a new term to be added to the language of Figure 6-2. As in the previous case, the solution trace analysis first produces a set of statements about various nodes in the search tree, which characterize why the training instance is positive. These statements 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 may be suggested.*

For example, in the case of the solution path shown in Figure 6-10, suppose that the analysis first determines that the solution path leads to a solution because State6 is of the following form, which we assume satisfies the system's definition of a solvable state.

$$\int poly(x) *_1 \ldots *_k poly(x)dx$$

Then the set of states, X_1, for which application of op5 leads to such a solvable state can be computed as:

$$X_1 \Leftarrow op5^{-1}(\int poly(x) *_1 \ldots *_k poly(x)dx)$$

giving

$$X_1 = \int poly^k(x)dx$$

In turn, we can compute the set of states, X_2, for which application of op4 leads to such a solvable state, as shown below. Here, "range(op4)" indicates the set of all problem states that can be reached by applying op4 to some other problem state.

$X_2 \Leftarrow op4^{-1}(intersection(range(op4),X_1))$.

By this repeated backward propagation of constraints through the solution tree, it can be determined that application of the solution method of Figure 6-10 leads to a solvable state when the initial state (in this case State1) is of the form $\int \cos^c(x)dx$ where c is constrained to satisfy the predicate "real(c) $\wedge$ integer((c-1)/2)", better known as "odd integer". Thus, detailed analysis of the solution path can suggest the need for new predicate terms in the language for describing heuristics. These terms (such as "odd integer") arise from combinations of existing terms, composed in a way that is determined by the particular operator sequence in the solution path being analyzed.

6.5 SUMMARY

The LEX system is an experiment in learning by experimentation. The current system, based on a generator of practice problems, problem solver, critic, and generalizer, indicates that useful problem-solving heuristics can be learned by employing empirical methods for generalizing from examples. It also indicates that more powerful and more general approaches to learning will be needed before practical systems can be built that improve their strategies in significant ways. One way of augmenting empirical learning methods by analytical methods has been discussed, which is based on giving the system the ability to reason about its goals, heuristic search, and the task domain. This research and the research of others (for example, that described in Chapters 8, 9, and 12 of this book) suggests that the addition of such meta-knowledge about the goals, the learner, and the problem-solving methods in the domain, is a promising area for further research.

ACKNOWLEDGMENTS

The LEX system has been developed over the past three years with the aid of several researchers in addition to the authors. We gratefully acknowledge the aid of William Bogdan, who helped implement the Critic; Bernard Nudel, who helped implement the Critic and Problem Solver; and Adam Irgon, who implemented the Problem Generator. Richard Keller has contributed to the newer work on using the intended purpose of heuristics for analyzing training instances. This research is supported by the National Science Foundation under Grant No. MCS80-08889, and by the National Institutes of Health under Grant No. RR-64309.

REFERENCES

Anzai, Y. and Simon, H., "The theory of learning by doing," *Psychological Review*, Vol. 36, No. 2, pp. 124-140, 1979.

Buchanan, B. G. and Mitchell, T. M., "Model-Directed Learning of Production Rules," *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (Eds.), Academic Press, New York, 1978.

Davis, R., "Applications of meta level knowledge to the construction and use of large knowledge bases," *Knowledge-based Systems in Artificial Intelligence*, Davis, R. and Lenat, D. (Eds.), McGraw-Hill, New York, 1981.

Fikes, R. E., Hart, P. E. and Nilsson, N. J., "Learning and executing generalized robot plans," *Artificial Intelligence*, Vol. 3, pp. 251-288, 1972.

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.

Iba, G. A., "Learning disjunctive concepts from examples," Master's thesis, M.I.T., Cambridge, Mass., 1979, (also AI memo 548).

Mitchell, T. M., *Version Spaces: An approach to concept learning*, Ph.D. dissertation, Stanford University, December 1978, (also Stanford CS report STAN-CS-78-711, HPP-79-2).

Mitchell, T. M., Utgoff, P. E., Nudel, B. and Banerji, R., "Learning problem-solving heuristics through practice," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vancouver, pp. 127-134, August 1981.

Mitchell, T. M., "Generalization as Search," *Artificial Intelligence*, Vol. 18, No. 2, pp. 203-226, March 1982.

Mitchell, T. M., "Toward Combining Empirical and Analytic Methods for Learning Heuristics," *Human and Artificial Intelligence*, Elithorn, A. and Banerji, R. (Eds.), Erlbaum, 1982.

Neves, D. M., "A computer program that learns algebraic procedures," *Proceedings of the 2nd Conference on Computational Studies of Intelligence*, Toronto, 1978.

Nilsson, N. *Principles of Artificial Intelligence*, Tioga, Palo Alto, 1980.

Politakis, P., Weiss, S. and Kulikowski, C., "Designing consistent knowledge bases for expert consultation systems", Technical Report DCS-TR-100, Department of Computer Science, Rutgers University, 1979, (also 13th Annual Hawaii International Conference on System Sciences).

Utgoff, P. E. and Mitchell, T. M., "Acquisition of Appropriate Bias for Inductive Concept Learning," *Proceedings of the 1982 National Conference on Artificial Intelligence*, Pittsburgh, August 1982.

Vere, S. A., "Inductive learning of relational productions," *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (Eds.), Academic Press, New York, 1978.

Waterman, D. A., "Generalization learning techniques for automating the learning of heuristics," *Artificial Intelligence*, Vol. 1, No. 1/2, pp. 121-170, 1970.