

LEARNING PROBLEM-SOLVING HEURISTICS THROUGH PRACTICE

Tom M. Mitchell
Paul E. Utgoff
Bernard Nudel
Ranan Banerji*

Computer Science Department, Rutgers University
New Brunswick, NJ 08903

ABSTRACT

We are interested in automatic improvement of problem solving strategies through practice.** A computer program called LEX is described, which acquires problem solving heuristics in the domain of symbolic integration. LEX learns heuristics from a sequence of presented training problems by (1) using available heuristics to solve the practice problem, (2) analyzing the steps it performed in solving the problem, and (3) proposing and refining domain-specific heuristics designed to improve its performance on subsequent problems. We describe the methods used by LEX, results obtained by the program, and strengths and weaknesses in the current system.

I INTRODUCTION

Efforts to build powerful, specialized, heuristic problem solvers have met with increasing success over the past decade. 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 designer to predict how the addition of a particular new heuristic or operator will affect overall system performance [1], [2]. In response to this problem, there has been increased interest over the past several years in developing semi-automated and automated methods to help construct expert heuristic problem solvers [3], [1], [4], [5], [2]. At the same time, in the Cognitive Psychology literature there have been several attempts to model acquisition of problem solving skills in humans [6], [7], [8], [9].

Our research is directed toward devising methods by which heuristic problem solving programs may improve their problem solving expertise through experience. As part of this research we have designed and constructed a computer program, called LEX, that incorporates *domain-independent methods* to discover *domain-dependent problem solving heuristics*. In [10] we reported a preliminary design for LEX. In this paper we report on the initial implemented system, which is able to learn useful heuristics to improve its performance at solving problems in symbolic integration. We discuss several strengths and weaknesses of the methods used by LEX, which have become evident from our initial experiments with the system.

*Computer Science Department, Temple University, Philadelphia, PA.

**Past and current support for this research has been provided by the National Science Foundation under grant MCS80-08889, and the National Institute of Health under grant RR-64309.

II THE PROBLEM

Given a set of operators for solving problems in symbolic integration, LEX learns a set of control heuristics, each of which recommends applying a particular operator to some general class of problem states. While 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 subclass of problem states to which the operator *should* be applied; that is, the subclass of problem states for which the operator leads to a solution. Preconditions for heuristics are learned by generalizing from examples of problem states to which the operator is applied in solving practice problems.

Thus, LEX begins with a heuristic problem solver, without the heuristics. In the domain of symbolic integration, the problem space is a space of mathematical expressions involving indefinite integrals. The operators supplied to LEX correspond to the usual rules of integration (e.g., integration by parts) as well as transformations that characterize algebraic equivalence of expressions (e.g., the associative law). The problem solving goal is to derive a problem state that contains no unsolved integrals.

OP1	$\text{INT } r f(x) dx \implies r \text{INT } f(x) dx$
OP2	$0 + f(x) \implies f(x)$
OP3	Integration by parts: $\text{INT } u dv \implies uv - \text{INT } v du$ (the precondition is internally represented as $\text{INT } f_1(x) f_2(x) dx$, where $f_1(x)$ corresponds to u and $f_2(x) dx$ corresponds to dv)
OP4	$\text{INT } f_1(x)+f_2(x) dx \implies \text{INT } f_1(x) dx + \text{INT } f_2(x) dx$
OP5	$\text{INT } \sin(x) dx \implies -\cos(x) + C$
OP6	$\text{INT } \cos(x) dx \implies \sin(x) + C$
OP7	$\text{INT } x^r dx \implies x^{r+1}/(r+1) + C$

Figure II-1: Some of the Operators for Symbolic Integration

Some of the over 30 operators currently defined in LEX are shown in figure II-1. Notice that some of these operators correspond to solutions to primitive integrals, some to problem transformations, and some to rules of algebraic equivalence.

In addition to being given a problem solver, representation for problem states, and problem solving operators, LEX also begins with a language with which to describe heuristics that it is to learn. Each learned heuristic is of the form:

IF the current problem state matches
the generalization P,

THEN apply operator O, with variable binding B.

The language of generalizations in the preconditions determines, to a large degree, the range of heuristics that can be learned. In LEX, for the symbolic integration domain, this language of generalizations is based on a grammar for algebraic expressions containing, and contained in, indefinite integrals. The sentences derivable by this grammar are the expressions that form legal problem states. The intermediate sentential forms derivable by the grammar constitute legal generalizations of problem states. Briefly, the grammar contains non-terminal symbols that correspond to classes of functions (e.g., trigonometric, polynomial) and classes of operators (e.g., function composition, multiplication, integration). These can be combined to form generalized algebraic expressions. Figure II-2 shows a portion of 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.

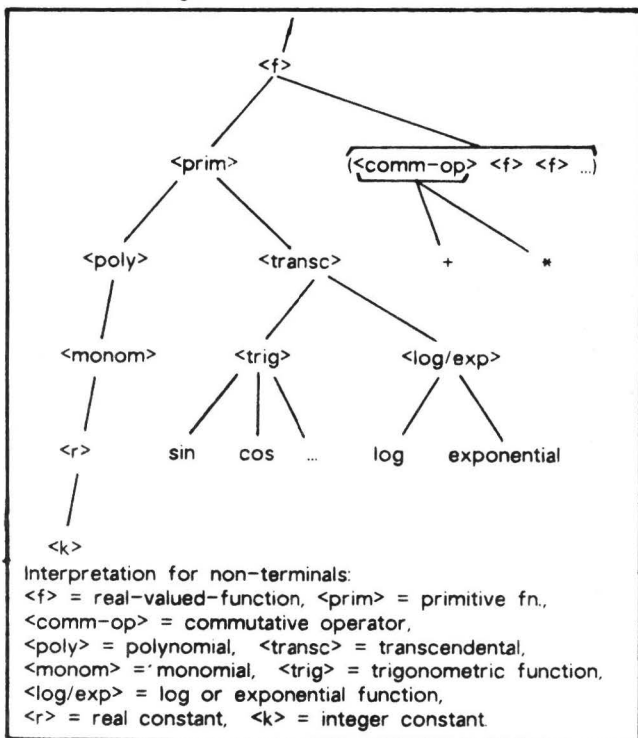


Figure II-2: A Portion of the Grammar for Algebraic Expressions (non-terminals are in angle brackets).

The heuristic H1, shown below, is a heuristic that can be described in this language, and is one of the heuristics that LEX has proposed. In this heuristic, the terms "x" and "transc" are combined by multiplication to form a generalized expression that describes the precondition of the heuristic.

H1:

IF the problem state contains $\text{INT } x \text{ transc}(x) \, dx$,

THEN use Integration by Parts: $\text{INT} u \, dv \Rightarrow uv - \text{INT} v \, du$,
With u bound to x, and dv bound to $(\text{transc}(x) \, dx)$.

LEX has been able to learn and describe useful heuristics in this language. However, the language is insufficient for describing the most useful forms of many heuristics. For example, the above heuristic incorrectly recommends Integration by Parts to solve $\text{INT } x \tan(x) \, dx$. Although this heuristic is still quite useful, a more appropriate heuristic requires using the term "any twice integrable function", which is not in the current language. One of our major current efforts is toward automating the detection and repair of deficiencies in the generalization language.

III DESIGN OF LEX

We decompose the problem of learning heuristics into four tasks. The design of LEX involves four corresponding modules - of which we have currently implemented three. These four modules are related to the four components of learning systems described in [11], and are characterized below.

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.

Critic: This module analyzes the search performed by the Problem Solver in attempting to solve a particular problem. The output of this module is a set of positive and negative training instances, corresponding to useful and wasteful search steps executed in the course of solving a specific practice problem.

Generalizer: This module generalizes from the training instances provided by the Critic, to suggest and refine general heuristics intended to produce more effective problem solving behavior on subsequent problems.

Problem Generator: After a given problem has been solved and analyzed, and the training data derived from this problem has been used to propose and refine heuristics, the Problem Generator must produce a new practice problem. The generated problem should be tractable given the current level of system expertise, yet should be difficult enough that its solution will yield additional informative training data. This module is *not* currently implemented. We mention it here to indicate our plans for the fully implemented system.

The first three of these modules are implemented in the current version of LEX. The results described here have been obtained by presenting practice problems by hand to the system. Below we describe in greater detail the methods used by these modules, as well as results obtained by the current system.

A. REPRESENTING INCOMPLETELY LEARNED HEURISTICS

Since the Generalizer requires many training instances to converge on an appropriate description of the heuristic(s) for a given operator, the Problem Solver must typically depend upon a set of *incompletely* learned heuristics to guide the search. These heuristics will be refined upon examination of subsequent training data, yet they form the basis for the problem solving necessary to acquire that data.

Although the problem of utilizing partially learned information has received little attention in previous research on learning, it is a central issue in learning by practice. The practice cannot take place unless the system has good methods for representing and utilizing knowledge that has not yet been well-honed by experience.

As an example, suppose that a heuristic is to be proposed to cover a newly encountered positive training instance representing a useful problem solving step (e.g., the application of a given operator to a given problem state). The appropriate generalization required for the new heuristic cannot typically be inferred from the single training instance.

LEX would represent the new, incompletely determined heuristic by representing the set of all describable heuristics that are consistent with this first positive instance. As subsequent training instances are encountered, LEX will refine this represented set of alternative versions of the heuristic, so that at any point it represents exactly *the set of all alternative versions of this heuristic that are consistent with the observed training instances*. We refer to this set of all plausible versions of the heuristic as the **version space** of the incompletely learned heuristic, relative to the observed instances, and the provided language for describing heuristics.

Representing an incompletely learned heuristic by storing explicitly all plausible versions of the heuristic is impractical. However, this version space can be represented efficiently, simply by storing 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:

1. H2 applies to (i.e. its precondition matches) any instance to which H1 applies.

and

2. For any instance to which both H1 and H2 apply, both recommend the same operator and the same binding of operator arguments.

***This definition is similar to that given in [12].

We will refer to the subset of maximally specific members of a version space as the subset S of members of that version space. Similarly, we refer to the subset of minimally specific (i.e., maximally general) members of the version space as the subset G. The version space of each incompletely learned heuristic is represented by computing and storing the sets S and G of that version space. These sets delimit the version space: any representable heuristic is contained in the version space if and only if it is both (1) more specific than or equal to some member of G, and (2) is more general than or equal to some member of S.

The representation and use of version spaces for generalizing from examples has been used previously in the Meta-DENDRAL program for inferring rules of mass spectroscopy, and is described more fully in [13] and [14]. By representing the version spaces of incompletely learned heuristics, LEX represents information critical to (1) converging on the heuristic when subsequent training data become available, (2) generating informative practice problems, and (3) allowing the Problem Solver to make use of incompletely learned heuristics in a reliable manner.

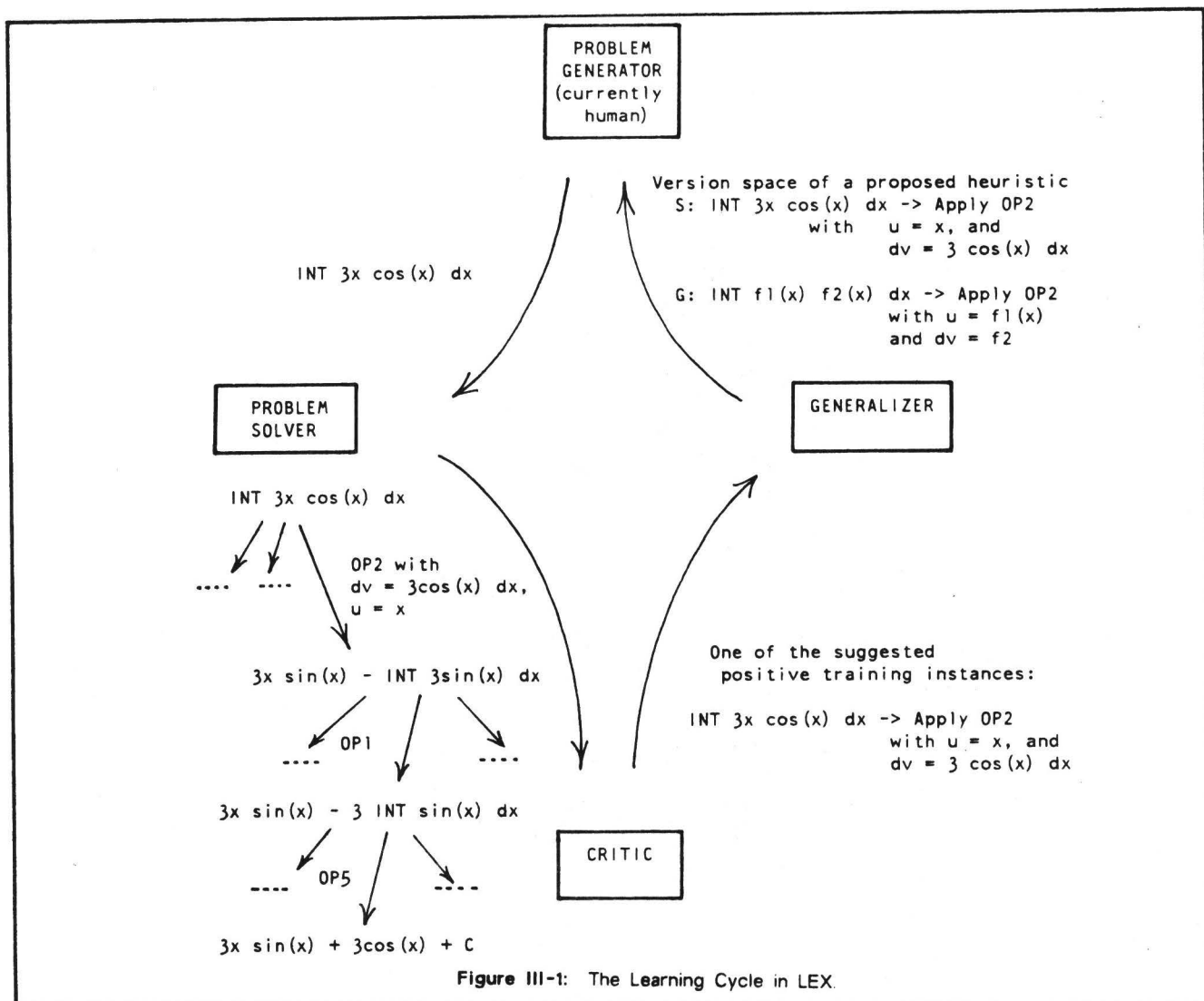
Figure III-1 illustrates one particular practice problem that was considered by LEX. This figure shows the search generated by the Problem Solver, one of the training instances produced by the Critic, and the sets S and G computed by the Generalizer to describe the resulting proposed heuristic. This example is explained in greater detail in the following subsections, which describe the operation of the three modules.

B. THE PROBLEM SOLVER

The Problem Solver uses a forward search strategy guided by whatever heuristics are available in the current solve-criticize-generalize cycle. It accepts 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.

The Problem Solver is faced with solving problems by using a set of incompletely learned heuristics. Because each such heuristic is represented in terms of its version space of alternative descriptions, the "match" of an incompletely learned heuristic to a given problem state may be ambiguous. We define the **degree of match** of an incompletely learned heuristic to a problem state to be the proportion of the members of the version space that match the given problem state. Thus, if all members of the version space match a given problem state, the recommendation of the incompletely learned heuristic is unambiguous, and its degree of match to that problem state is 1. The degree of match of an incompletely learned heuristic to a given problem state is used by the Problem Solver to weight the associated heuristic recommendation. Since the degree of match is difficult to compute exactly, it is estimated by the proportion of the union of the sets S and G that match the given problem state. This estimate seems to work well empirically, and is relatively inexpensive to compute.

The Problem Solver strategy for choosing a node to expand, and an operator with which to expand it, is characterized below. Here, the "cost" of a node refers to the sum of execution costs for each step along the least cost path from the root of the tree to the current node. An open node refers to any node in the search tree with at least one applicable operator that has not yet been applied.



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 this heuristic,

ELSE follow the recommendation of one of the applicable heuristics, choosing that heuristic which applies with the highest estimated degree of match, breaking ties in favor of expanding the highest cost open node.

The main features of the Problem Solver strategy may be summarized as follows: In the absence of heuristics, the problem solver conducts a uniform cost search for a solution. When available heuristics apply to any of the open nodes in the search, their recommendations are followed, weighted by their estimated degree of match.

C. THE CRITIC

After the solution has been determined, the Critic examines the search trace, and selects certain search steps as 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.

Those problem solving steps that lead to the lowest cost solution found, are isolated as positive training instances. Other steps that lead away from the lowest cost solution path are isolated as negative instances - choices that the learned heuristics should not recommend. The output of the Critic is a set of these positive and negative examples. Figure III-1 illustrates 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 (a) leads from a node on the lowest cost solution path found, to a node not on this path, and (b) when its resulting state is given anew to the Problem Solver, leads either to no solution or to a high cost solution. Here a solution is considered high cost if its cost is more than **High-Cost-Factor** times the cost of the lowest cost known solution (for the results reported here **High-Cost-Factor** was 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. More importantly, it is possible for the Critic to label as negative a search step along the true (but not found) minimum cost solution path. Both kinds of errors can arise because the Problem Solver is not assured of finding the minimum cost solution. Criterion 2b 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 labelling training instances requires a full, uniform-cost search [15]. We have found empirically that the current approximate strategy of the Critic does not seem to be a limiting factor in overall system performance at this point.

D. 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 subsequently refine heuristics to improve problem solving performance. The generalization problem faced by this module is one of learning from examples. Given a set 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 appropriate bindings of variables in the operator.

The Generalizer represents the version space for each proposed heuristic for each operator, in terms of the sets *S* and *G* that delimit the plausible versions of that heuristic. For example, figure III-1 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 III-1. 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: If no member of the version space of any current heuristic applies to a newly presented positive instance, a new heuristic is formed to cover this instance. This is the case in the example of figure III-1. In forming a new heuristic, the set *S* is initialized to the very specific version of the heuristic, that applies *only* to the current training instance (this is the most specific describable 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. In the example of figure III-1, *G* is therefore initialized to the version whose precondition is exactly the description of the precondition for OP2. That is, $\text{INT } f1 \ f2 \ dx$ represents the integral of the product of any two real functions of *x*, and corresponds to the precondition $\text{INT } u \ dv$ 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, and consistent with this single training instance. As subsequent positive instances are considered, *S* will become more general to include newly observed instances in which OP2 is found to be useful. Likewise, *G* will become more specific in order to exclude negative instances in which OP2 may validly be applied, but in which it does not lead to a solution. 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 reduced.

Refining incompletely learned heuristics: If an existing heuristic applies to a newly presented positive or negative instance, 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 $\text{INT } 5x \sin(x) \ dx$. The solution to this problem leads to both a positive and a negative training instance for the heuristic from figure III-1. Figure III-2 shows these two new training instances, and the way in which they allow refining 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 $\text{trig}(x)$ replaces $\cos(x)$ so that the heuristic will apply to integrals containing *any* trigonometric function of *x*. Similarly, *k* (denoting any integer constant) replaces the 3 in the specific heuristic. The program determines this revision by first noting that the terms 3 and $\cos(x)$ in the old *S* prevent this generalization from applying to the new instance. It then consults the grammar for expressing heuristics (shown in figure II-2) to determine the next most general terms that can be substituted in order to include this new instance. 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.

Version Space of Heuristic from Figure III-1:	
S:	INT $3x \cos(x) dx \rightarrow$ Apply OP2 with $u = x$, and $dv = 3 \cos(x) dx$
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 $5x \sin(x) dx \rightarrow$ Apply OP2 with $u = x$, and $dv = 5 \sin(x) dx$
Negative training instance:	
	INT $5x \sin(x) dx \rightarrow$ Apply OP2 with $u = \sin(x)$, and $dv = 5x dx$
Revised Version Space:	
S:	INT $kx \text{ trig}(x) dx \rightarrow$ Apply OP2 with $u = x$, and $dv = k \text{ trig}(x) dx$
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 III-2: Revising the Version Space of a Heuristic.

The general boundary of the revised version space of figure III-2 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 $f_1(x)$ (which represents "any real-valued function") by the next more specific *acceptable* expression. Notice in the hierarchy of figure II-2, this expression is "poly".

As subsequent training instances are considered, this partially learned heuristic will be further refined so that S and G converge toward a useful statement of the heuristic, such as:

INT $kx \text{transc}(x) dx \rightarrow$ apply OP2 with $u = x$,
and $dv = k \text{transc}(x) dx$

Although the above example illustrates the basic method used to propose and refine heuristics, the Generalizer allows a slightly more complex method in order to deal with errors in the training data (the Critic is fallible). The method used is an extension to the basic version space approach, originally suggested in [13]. This extension

involves computing several "concentric" version spaces, each one consistent with a smaller proportion of the training instances. We are currently experimenting with the Generalizer to collect data regarding its effectiveness in dealing with data errors. The results reported in this paper do not use this extension.

The Generalizer currently 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 begin a new heuristic (i.e., 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 the version space applies.

IV RESULTS AND DISCUSSION

We have written the LEX program as a means toward developing and testing methods for learning heuristics. The value of this system is as an experimental tool rather than as a problem solver in symbolic integration (much better problem solvers exist in this domain). Here we present empirical results showing the improvement of problem solving performance through practice, and discuss some insights gained through experiments with LEX.

A. RESULTS

We have measured the performance improvement of LEX, on solving a set of 5 test problems (shown in figure IV-3), between stages of training on 12 training problems (shown in figure IV-2). No training was allowed during solution of the test problems. Performance on the test problems was measured before any training had occurred, and again after every second training problem. Two performance measures were considered: (1) total number of search steps required to solve the 5 test problems, and (2) total c.p.u. time spent applying operators in solving the 5 test problems (this excludes the resources spent in matching preconditions of operators and heuristics to problem states). The resource allocation given to the Problem Solver was 240 c.p.u. seconds and 800,000 cons cells to be spent in applying operators (running in RUCILISP on a DEC-2060).

Fourteen heuristics were formed by LEX during this training session, covering thirteen of the 32 operators available (shown in figure IV-1). Twelve of these fourteen heuristics remained incompletely learned at the end of the training sequence (i.e., their version space still contained multiple plausible generalizations of the heuristic).

Figure IV-4 displays the improvement in performance by LEX for this experiment. For both of the performance measures shown, performance improved by roughly two orders of magnitude. In those cases where the Problem Solver could not solve a test problem within the given resource allocation, the data point represents a lower bound on the required effort (the lower bound was taken to be the resources spent before aborting the solution attempt). Each such lower bound data point is shown as an "*" in figure IV-4. A third performance measure not shown here -- total c.p.u. time spent *including* time matching operators and heuristics to problem states -- improved by roughly one order of magnitude for the same training sequence.

```

INT sin(x)dx => -cos(x)
INT xr dx => xr+1 / (r+1)
INT r f(x)dx => r INT f(x)dx
INT x-1 dx => ln(x)
INT exp dx => exp
ef(x) => exp(f(x))
INT ln(x) dx => x ln(x) - x
1*f(x) => f(x)
INT cos(x) dx => sin(x)
INT (f(x)+g(x))dx => INT f(x)dx+INT g(x)dx
INT u dv => uv - (INT v du)
INT r dx => rx
0*f(x) => 0
INT x dx => (x2)/2
0+f(x) => f(x)
f(x) - 0 => f(x)
ln(exp(x)) => x
exp(ln(x)) => x
r1*r2 => evaluated (r1 * r2)
tan(x) => sin(x)/cos(x)
cot(x) => 1/tan(x)
sec(x) => 1/cos(x)
csc(x) => 1/sin(x)
0 * expr => 0
1 * expr => expr
expr - 0 => expr
1 / f(x) => f(x)-1
INT (f(x)/r)dx => (INT f(x) dx)/r
f1 * (f2 * f3) => f2 * (f1 * f3)
f1 + (f2 + f3) => f2 + (f1 + f3)
f1 * (f2 + f3) => (f1 * f2) + (f1 * f3)
r1+r2 => evaluated (r1 + r2)

```

Figure IV-1: Operators Available to LEX.

1. INT 4 dx
2. INT cos(y) dy
3. INT 5*(3+4) dx
4. INT 3x dx
5. INT x*x dx
6. INT 3*(x+5) dx
7. INT 0 * ((5+2.1)/sin(x)) dx
8. INT x+(e^x) dx
9. INT x³ dx
10. INT (3+cos(x))*4.1 dx
11. INT x*(3+4) dx
12. INT 5.3*sin(x) dx

Figure IV-2: Sequence of 12 training problems

1. INT x*sin(x) dx
2. INT 3*(4*cos(x)) dx
3. INT 0*(sin(x)+(x³)) dx
4. INT (2*(x⁴))+x dx
5. INT (3*x)*(e^x) dx

Figure IV-3: Set of 5 test problems

B. DISCUSSION

The most useful results we have obtained are not quantitative measurements of system performance, but rather insights gained in developing and running the system. Following are some observations about the performance of LEX that indicate the most significant deficiencies and successes of the initial design.

The ability of the Problem Solver to utilize incompletely learned heuristics is very important in allowing it to solve

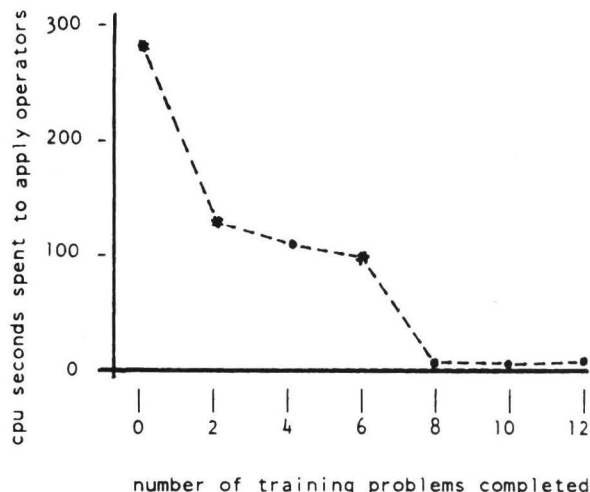


Figure IV-4: Performance Results

problems that will provide additional training data. In our experiments it has almost always been the case that the majority of heuristics available for solving practice problems are only incompletely determined by the data. Without reasonable methods for utilizing these incompletely learned heuristics, the Problem Solver would be faced with unrealistically large searches in solving subsequent practice problems.

The use of learned heuristics by the Problem Solver during training is not without drawbacks, however. Having reached an adequate level of problem solving ability, LEX typically becomes conservative, following its current heuristics while training rather than applying operators for which no heuristics exist. It thus improves slightly by refining known heuristics, but may miss out on larger improvements that would be possible by learning heuristics for new operators.

A still more disconcerting problem is that incompletely learned heuristics can lead LEX to expensive excursions down absurd search paths, preventing solution of the problem within the allocated resources. In the experiment described above, for example, while LEX could solve all five test problems after the fourth training problem, it could only solve four of the test problems following the sixth training problem. This phenomenon was due to its formulation of new, incompletely learned heuristics that prevented the Problem Solver from obtaining the solution it had found using the earlier set of heuristics. We are

currently considering various refinements to the Problem Solver's strategy for utilizing incompletely learned heuristics.

We have noted that the Critic may produce incorrect training instances. In spite of this possibility, it rarely produces instances that are inconsistent with our own intuitions. In a significant number of cases, when the Critic calls the Problem Solver to consider a possible negative instance (see criterion 2.b in the earlier discussion of the Critic) an improved problem solution is found. This occurred in 4 of the 12 training problems in the current experiment. The approximate Critic algorithm seems to work reasonably well, and the Critic does not seem to be a limiting factor in determining overall system performance at this point.

Our initial experiments indicate that the insufficiency of the language for expressing heuristics is an important limitation of our system. In experimenting with the system, and observing it sometimes unable to represent heuristics for which it has training data, we have already been led to extend the language manually. We have tried to understand the reasoning that we go through in coming up with an appropriate extension to the language. We consider the issue of detecting and repairing deficiencies in the generalization language to be a major focus for future work.

One shortcoming of the initial design of the Generalizer is that it fails to take advantage of an important source of information for choosing an appropriate generalization: analysis of the problem solution. As an example of the kind of information available from analyzing the problem solution, consider the example of figure III-1. By considering the role that the indicated positive instance played in reaching the solution, it is reasonable to infer that the appropriate generalization of the heuristic should cover all problem states for which the operator sequence OP2, OP1, OP5 leads to a solution. By reasoning about the definitions of these operators, the system might determine the generalization corresponding to this class of instances (i.e., the precondition for the "macrooperator" [16] composed of this operator sequence). In the example of figure III-1 the value of the constant 3 in the problem does not effect the feasibility or utility of the operator sequence <OP2, OP1, OP5>. Thus, the S boundary of the version space in figure III-1 could be generalized by replacing the constant 3 with the symbol "r" that matches any real constant.

Summary: The initial implementation of LEX has been described via an extended example. Empirical results, showing the improvement in performance with training, have been given. While LEX is able to learn heuristics that lead to improved performance at solving problems of symbolic integration, the methods used by each of the modules of LEX can be characterized as domain-independent methods for learning domain-dependent heuristics.

V ACKNOWLEDGMENTS

We gratefully acknowledge the aid of Mr. William Bogdan, who took part in the implementation of the current system.

REFERENCES

[1] Davis, R. "Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases", Technical report STAN CS-76-552, Stanford University, June 1976.

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

[3] Waterman, D. A. "Generalization learning techniques for automating the learning of heuristics" *Artificial Intelligence*, 1:1/2 (1970) 121-170.

[4] Buchanan, B. G. and Mitchell, T. M., "Model-directed learning of production rules," in *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F., eds., Academic Press, New York, 1978

[5] Mostow, D. J. and Hayes-Roth, F. "Machine-aided heuristic programming a paradigm for knowledge engineering", Technical report Rand N-1007-NSF, The Rand Corporation, February 1979.

[6] Anderson, J. R. "A general learning theory and its application to the acquisition of proof skills in geometry." In *Proceedings of the Workshop on Machine Learning*, Carnegie-Mellon University, Pittsburgh, PA, July, 1980.

[7] Anzai, Y. and Simon, H. "The theory of learning by doing" *Psychological Review*, 36:2 (1979) 124-140.

[8] Neches, R. "Promoting self-discovery of improved strategies." In *Annual conference of the American Educational Research Association*, San Francisco, CA, April, 1979, also report CIP 398, Psychology Department, Carnegie-Mellon

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

[10] Mitchell, T. M., Utgoff, P. E. and Banerji, R. B. "Learning problem-solving heuristics by experimentation" In *Proceedings of the Workshop on Machine Learning*, Carnegie-Mellon University, Pittsburgh, PA, July, 1980.

[11] Smith, R. G., Mitchell, T. M., Chestek, R. A. and Buchanan, B. G. "A model for learning systems" In *IJCAI-5*, Cambridge, Mass., 1977, 338-343.

[12] Vere, S. A., "Inductive learning of relational productions," in *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F., eds., Academic Press, New York, 1978.

[13] Mitchell, T. M. "Version Spaces: An approach to concept learning", Technical report Ph.D. thesis, Stanford University, December 1978, also Stanford CS report STAN-CS-78-711, HPP-79-2

[14] Mitchell, T. M. "Generalization as search" *Artificial Intelligence*, (1981), to appear

[15] Nilsson, N. J. *Problem-Solving Methods in Artificial Intelligence*, McGraw-Hill, 1971.

[16] Fikes, R. E., Hart, P. E. and Nilsson, N. J. "Learning and executing generalized robot plans." *Artificial Intelligence*, 3 (1972) 251-288.