A Formalization of Explanation-Based Macro-operator Learning

Prasad Tadepalli
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3202
(tadepalli@cs.orst.edu)

Abstract

In spite of the popularity of Explanation-Based Learning (EBL), its theoretical basis is not well-understood. Using a generalization of Probably Approximately Correct (PAC) learning to problem solving domains, this paper formalizes two forms of Explanation-Based Learning of macro-operators and proves the sufficient conditions for their success. These two forms of EBL, called "Macro Caching" and "Serial Parsing," respectively exhibit two distinct sources of power or "bias": the sparseness of the solution space and the decomposability of the problem-space. The analysis shows that exponential speedup can be achieved when either of these biases is suitable for a domain. Somewhat surprisingly, it also shows that computing the preconditions of the macro-operators is not necessary to obtain these speedups. The theoretical results are confirmed by experiments in the domain of Eight Puzzle. Our work suggests that the best way to address the utility problem in EBL is to implement a bias which exploits the problem-space structure of the set of domains that one is interested in learning.

1 Introduction

Explanation-Based Learning (EBL) treats learning as improving the efficiency of a problem solver [Dejong and Mooney, 1986, Mitchell *et al.*, 1986]. The standard EBL systems start with complete and correct, if inefficient, problem solvers. Learning involves taking a set of examples, i.e., problem-solution pairs, as input and producing an efficient problem solver as output. The examples provide information about the problem distribution and eliminate the need to search for solutions.

While there are satisfactory formal models of empirical learning based on variants of Probably Approximately Correct (PAC) learning [Valiant, 1984, Haussler, 1990], EBL systems are not theoretically well-understood. There are some practical problems in EBL which can be directly attributed to the lack of adequate theoretical understanding. Firstly, there is no clear definition of "success" for EBL systems. Because of this reason there hasn't been a rational method to decide when to stop learning. Any satisfactory formalization of EBL should specify what it means to succeed and should provide an effective method to recognize success. Secondly, it is known that EBL systems do not always lead to performance improvement with learning. After learning a large number of rules, they might face what is called the "utility problem," i.e., the problem of complexity of using the learned knowledge [Minton, 1990]. Since the utility problem can far outweigh the reduction in search due to learning, a successful formalization should take this into account and characterize the sufficient conditions for a guaranteed performance improvement.

This paper introduces a variation of the formal framework for performance improvement learning of [Natarajan and Tadepalli, 1988]. We call this framework Probably Approximately Correct (PAC) Problem Solving. Using this framework, we analyze two forms of macro-operator learning, namely, Macro Caching and Serial Parsing. One of the main results of this paper is an explication of the biases exhibited by these two forms of EBL. In particular, Macro Caching and Serial Parsing implement two distinct biases: (a) the sparse solution space bias and (b) the macro table bias. The sparse solution space bias says that most problems in the domain can be solved by a small number of operator sequences. The macro table bias says that the solution to a problem can be constructed by serially composing a set of macro-operators that solve a series of subproblems. We show that when a domain and the hypothesis space of the learning system satisfy these biases, it leads to an exponential speedup in problem solving. Somewhat surprisingly, the analysis also reveals that it is not necessary to compute the preconditions of macro-operators in order to obtain this speedup. The theoretical predictions of our analysis are confirmed by an implementation of Serial Parsing in the domain of Eight Puzzle.

The main contribution of this paper is a successful integration of EBL work with PAC learning. It also extends Korf's work on macro-operator learning [Korf, 1985] to incremental learning by observation. Our work also suggests that the utility problem in EBL can be solved by building learning programs that implement biases which exploit the domain structure.

The rest of the paper is organized as follows: Section 2 introduces PAC Problem Solving. Sections 3 and 4 describe Macro Caching and Serial Parsing, respectively. Section 5 presents experimental results in the Eight Puzzle domain. Section 6 discusses the related work, and the final section summarizes the contributions.

2 Probably Approximately Correct Problem Solving

In order to formally analyze EBL, we need to precisely define what it means for EBL, or any other speedup learning method, to succeed. The problem definition in [Mitchell *et al.*, 1986] is not sufficient for our purposes, because it does not address the issue of learning from multiple examples. This section presents a definition which draws from the earlier formal frameworks presented in [Natarajan and Tadepalli, 1988].

The key difference between purely empirical learning and EBL is that an EBL system is also provided with a "domain theory," which, we assume, is in the form of a set of goals and operators. However, this domain theory is too inefficient to use directly to solve problems. The operationality constraint of EBL [Mitchell *et al.*, 1986] may be viewed as defining a hypothesis space of potential efficient problem solvers, one of which is the target problem solver. In this paper, we view the problem of performance improvement as learning an efficient, approximate problem solver from the domain theory and the example solutions of some target problem solver in the hypothesis space.

Define a *problem domain* D to be the tuple $\langle S, G, O\rangle$, where

- S = A set of *states*.
- G = A set of *goals*, where each goal is described by a procedure that recognizes a set of states in S.
- O = A set of *operators* $\{o_1, o_2, \ldots\}$, where each o_i is a total function[1] from S to S.

The combination of goals G and the operators O is called the *specification* of D. Let us denote the result of applying an operator o on a state s by $o(s)$.

A *problem* is a pair $\langle s, g\rangle$, where $s \in S$ is a state and $g \in G$, a goal. A problem $\langle s, g\rangle$ is *solvable* if there is a sequence of operators $\beta = (o_1, \ldots, o_d)$, and a sequence of states $(s_0, \ldots, s_d)$, such that (a) $s = s_0$,(b) for all i from 1 to d, $s_i = o_i(s_{i-1})$, and (c) s_d satisfies the goal g.

In the above case, β is a *solution sequence* of $\langle s, g\rangle$, and d is the *length* of the solution sequence β. The maximum of lengths of description of all s_i, $i \in 0 \ldots d$ is called the *step length* of the solution sequence.

Notice that our domain specification is not as "explicit" as typical EBL programs require them to be. The operators need not be described in the STRIPS formalism, and goals need not be logical formulae. In fact, they need not be declaratively represented at all, but may be described by procedures whose run time is reasonably bounded. Thus, our learning framework requires the learning techniques to be more independent of the operator representation than the standard EBL techniques. This allows choosing the operator representation which is best suited to the domain rather than being constrained by the needs of the learning technique.

Secondly, unlike the standard EBL approaches, our goals and operators are not parameterized. At first glance, this appears to seriously limit the power of EBL. Somewhat surprisingly, this is not the case. This is because, for most domains, the maximum number of possible instantiations of any parameterized operator (or macro-operator) must be bounded by a polynomial factor so that the cost of instantiating it is not exorbitant. This means that every parameterized operator (or macro-operator) can be replaced with at most a polynomial (in the length of the state description) number of non-parameterized operators. To take a domain like chess as an example, every parameterized operator (e.g., move pawn) can be replaced with $n \times n$ non-parameterized operators (e.g., move pawn from $e2$ to $e4$), where n is the length of the state description, i.e., the number of squares on the chess board. Since, as our later definitions show, our formalization ignores all polynomial factors in sample size, learning time, and problem solving time, our results do not change whether the operators are parameterized or not. A parameterized model was considered and similar results were proved in [Tadepalli, 1990]. For simplicity of exposition, this paper considers only non-parameterized operators.

A *problem solver* f for D is a deterministic program that takes as input a problem, $\langle s, g\rangle$, and computes its solution sequence, if such exists.

A hypothesis space H is a set of problem solvers.

An *example* for a domain D is a pair $\langle\langle s, g\rangle, \beta\rangle$, where β is a solution sequence of $\langle s, g\rangle$.

A *meta-domain* M is any set of domains.

A *learning algorithm* for M is an algorithm that takes as input the specification of any domain $D \in M$ and some number of examples of problem solving with a target problem solver in H and computes as output an approximation for a problem solver for D.

The learning protocol is as follows: First, the domain specification is given to the learner. The teacher then selects a problem distribution from a set of allowed distributions and a target problem solver from the hypothesis space. The learning algorithm has access to a routine called SOLVED-PROBLEM. At each call, SOLVED-PROBLEM randomly chooses a problem in the input domain, solves it using the target problem solver, and returns the example (the $\langle$*problem solution*$\rangle$ pair). The learning algorithm must output an approximate problem solver with a high probability after seeing a reasonable number of examples. The problem solver need only be approximately correct in the sense that it may fail to produce correct solutions with a small probability.

Definition 1 *Formally, an algorithm A is a learning algorithm for a meta-domain M in a hypothesis space H with respect to a set of problem distributions Γ, if for any domain $D \in M$, any choice of a problem distribution P in Γ, and any target problem solver $f \in H$,*

1. *A takes as input the specification of a domain $D \in$*

[1] This can be extended to partial functions by assuming that any operator, when applied to a state in which it is not applicable, will lead to a "dead state."

M, an error parameter ϵ, and a confidence parameter δ,

2. *A may call SOLVED-PROBLEM, which returns examples $\langle x, f(x)\rangle$ for D, where x is chosen with probability $P(x)$ from $S \times G$. The number of oracle calls of A and its running time must be polynomial in the maximum problem size n, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and the length of its input.*
3. *For all $D \in M$ and distributions $P \in \Gamma$, with probability at least $(1-\delta)$ A outputs a program f' that approximates f in the sense that $\Sigma_{x \in \Delta} P(x) \leq \epsilon$, where $\Delta = \{x | f' \text{ fails on } x \text{ while } f \text{ succeeds}\}$.*
4. *There is a polynomial R such that, for a maximum problem size n, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, maximum length l and maximum step length r of any solution output by SOLVED-PROBLEM, and an upper bound t on the running times of programs in D on inputs of size n, if A outputs f', the run time of f' is bounded by $R(n, l, r, t, \frac{1}{\epsilon}, \frac{1}{\delta})$.*

This framework is very similar to that introduced in [Natarajan and Tadepalli, 1988]. The main difference is the idea of the hypothesis space which is not present in [Natarajan and Tadepalli, 1988]. Another difference is that it uses a less powerful SOLVED-PROBLEM oracle than the USEFUL oracle of [Natarajan and Tadepalli, 1988] which is capable of generating optimal solutions for any problem. We also insist that SOLVED-PROBLEM must solve all problems using the same target problem solver chosen from the hypothesis space.

The first two conditions in Definition 1 require that the learning algorithm must terminate and output the problem solver within reasonable computational time limits. The third condition requires that with probability at least $1-\delta$, the learning algorithm should output an approximate problem solver for the domain. The problem solver is approximate in the sense that it is allowed to fail to find a solution with a probability less than ϵ while the target problem solver succeeds when tested on problems chosen according to the same distribution that was used in training. Note that the problem solver might find any correct solution and not necessarily the same solution that the target problem solver would have found. This is one reason why this framework is not subsumed by that of PAC learning of functions described in [Natarajan, 1989]. The final condition is to ensure that the problem solver output by the learning algorithm scales reasonably well.

We call this framework Probably Approximately Correct (PAC) Problem Solving.

3 Macro Caching

In this section we describe Macro Caching and characterize the sufficient conditions for it to be a learning algorithm for a meta-domain.

We define a macro-operator (or a macro) to be any sequence of operators. Macro Caching consists of verifying that an operator sequence solves the problem by executing it (the "explanation" step of EBL) and storing the entire operator sequence from the start to the goal state in the example as a single macro. Unlike the standard implementations of EBL, our algorithms do not rely on precondition computation. Thus, a macro may only be viewed as a *potential* solution sequence, which must be tested at the problem solving time. The problem solver works by applying each macro to the current state and testing whether it succeeds. If some macro succeeds in achieving the goal, it returns the macro; otherwise it fails.

The idea is to pick enough examples so that after extracting the macros from these examples, one is reasonably certain that the learned problem solver is probably approximately correct. One difficulty remains, however. The number of examples necessary for successful learning depends on the the number of necessary macros, and we do not a priori know this number. Hence, a different approach called "stochastic testing," used in [Angluin, 1988], is adapted to our problem. The idea is to stochastically determine whether learning is complete by testing the program on randomly selected examples. In this "on-line" model, the program is tested as it is being trained, and the learning is terminated as soon as it succeeds on enough number of consecutive random examples. The testing scheme, described in Table 1, has a slightly better bound on the number of test examples than that of [Angluin, 1988].

Input domain specification $D \in M$, error parameter ϵ,
    confidence parameter δ
Test-counter := 0
$i := 1$
While *Test-counter* $\leq \frac{1}{\epsilon}\{2\ln(i+1) + \ln\frac{1}{\delta}\}$ do
    Let SOLVED-PROBLEM() return
        ⟨problem $\langle s, g\rangle$, solution β⟩
    If $\langle s, g\rangle$ is solvable by the current problem solver
      Then
        Increment the *Test-counter* by 1.
      Else Begin
        Reset the *Test-counter* to 0.
        Update the problem solver by learning
          from ($\langle s, g\rangle$, β)
        $i := i + 1$
        End;
End While;
Output current problem solver;

Table 1: Stochastic testing scheme.

Definition 2 *A problem solver f for a domain D and a problem distribution P satisfies a* sparse solution space bias *if there is a set of operator sequences m_f such that, on any problem $x \in D$ such that $P(x) > 0$, $f(x) \in m_f$, and $|m_f|$ is bounded by a polynomial Q in the problem size n.*

Intuitively, the sparse solution space bias implies that there is a small number of operator sequences out of which solutions can be selected, so that the simple strategy of remembering all of them and trying them one by one would work well. The small number of operator sequences might be due to (a) the small number of problems that have a non-zero probability in the given problem-distribution (sparseness of problem distribution) or (b) the large number of states that satisfy the goal (high density of goal states), or a combination of both. To give an example of the second case, if a goal in the blocks world is to achieve a state in which there is a clear block on the table, it can be achieved by one of three macros: a null macro, a macro which puts the block being held in hand on table, and another macro which picks up a block on the top of a tower and places it on the table. This works for an exponential number of (in fact, all) initial states because an exponential number of states also satisfy the goal.

Theorem 3 *If H is defined by the set of problem solvers that satisfy the sparse solution space bias for all domains D in M and all distributions P in Γ, then stochastic testing with Macro Caching is a learning algorithm for M in H with respect to Γ.*

Proof (sketch): Notice that, at any stage i, the learner has to succeed on $\frac{1}{\epsilon}(2\ln(i+1)+\ln\frac{1}{\delta})$ randomly chosen test problems for the learner to terminate. At any stage i, the probability that the learner succeeds on all its tests when its macros are not, in fact, adequate to solve a randomly chosen problem with a probability greater than $1-\epsilon$ is less than $(1-\epsilon)^{\frac{1}{\epsilon}(2\ln(i+1)+\ln\frac{1}{\delta})}$, which turns out to be less than $\frac{\delta}{(i+1)^2}$. So, if the program terminates at some stage j, the probability that we have a problem solver whose probability of failure is greater than ϵ is less than or equal to $\sum_{i=1}^{j}\frac{\delta}{(i+1)^2} < \int_{i=0}^{j}\frac{\delta}{(i+1)^2}di < \delta$.

By the sparse solution space bias, the number of macros necessary to solve the domain is bounded by a polynomial $Q(n,t)$. Since the lack of each such macro can cause at most one failure in our algorithm, the number of failures i satisfies $i \leq Q(n,t)$. Hence, the total number of iterations of the while loop, and the total number of examples needed is bounded by $\frac{Q(n,t)}{\epsilon}(2\ln(Q(n,t)+1)+\ln\frac{1}{\delta})$, which is a polynomial. It can be seen that the program runs in polynomial time.•

Macro Caching's main limitation is due to its assumption that a small number of macros can solve most problems in the problem-space. For example, it does not work in domains like Rubik's Cube and Eight Puzzle, where each macro can solve at most one problem, and the problem distribution is not sparse.

4 Serial Parsing

This section describes how EBL can succeed for arbitrary problem distributions by exploiting a problem-space structure called serial decomposability [Korf, 1985].

Here we make the assumption that states are representable as vectors of discrete valued features, $\langle v_1, ..., v_n\rangle$, where the maximum number of values a feature can take is bounded by a polynomial in n.

In Rubik's Cube, the variables are cubie names, and their values are cubie positions. In Eight Puzzle, the variables are tiles, and their values are tile positions. Note that the above assumption makes it difficult to represent domains with relations, e.g., the blocks world.

A domain is *serially decomposable* for a given total ordering on the set of features if the effect of any operator in the domain on a feature value is a function of the values of only that feature and all the features that precede it [Korf, 1985].

Rubik's Cube is serially decomposable for any ordering of features (also called "totally decomposable"). In Eight Puzzle, the effect of an operator on any tile depends only on the positions of that tile and the blank in the original state. Hence Eight Puzzle is serially decomposable for any ordering that orders the blank as the first feature.

Note that serial decomposability is a property of the domain as well as its representation. If Eight Puzzle is represented with positions as variables and tiles as their values, then it is not serially decomposable.

We assume that there is a single, fixed goal state described by $\langle g_1, \ldots, g_n\rangle$.[2]

Korf defines a macro table as a set of macros $M_{j,i}$ such that if $M_{j,i}$ is used in a solvable state s where the features 1 thru $i-1$ have their goal values, $g_1, \ldots, g_{i-1}$ and the feature i has a value j, then the resulting state is guaranteed to have goal values $g_1, \ldots, g_i$ for features 1 thru i.

Korf showed that if a domain is serially decomposable and every state reachable from a solvable state is also solvable, then it has a macro table [Korf, 1985]. If a domain is serially decomposable for the feature ordering $\langle 1, \ldots, n\rangle$, then any move sequence that takes a state $\langle g_1, \ldots, g_{i-1}, j, s_{i+1}, \ldots\rangle$ to $\langle g_1, \ldots, g_i, t_{i+1}, \ldots\rangle$ can be used as a macro $M_{j,i}$, since the values of features 1 thru i in the goal state only depend on their values in the initial state, and not on the values of other features.

If a full macro table with appropriately ordered features is given, then it can be used to construct solutions from any initial state without any backtracking search [Korf, 1985].

Definition 4 *A problem solver f satisfies a* macro table bias *for a domain D in M if there is a feature ordering $O = \langle 1, \ldots, n\rangle$ such that, (a) D is serially decomposable for O, and (b) f constructs all its solutions using a macro table M as follows: for each feature i from 1 to n, macros $M_{j,i}$ are successively applied, where j is the value of feature i in the state before applying the macro.*

Since each application of macro $M_{j,i}$ guarantees that the features 1 thru i will have their goal values, any solvable problem is solved in n macro applications by such problem solver. Korf's learning program builds a macro table by exhaustively searching for a correct entry for each cell in the table [Korf, 1985]. Thus, Korf's work might be characterized as "learning by doing." Our method, called Serial Parsing, extends his work to incremental learning of macro tables by observation.

The stochastic testing algorithm now calls Serial Parsing instead of Macro Caching to extract the macros in a solution sequence. For a goal $\langle g_1, \ldots, g_n\rangle$, Serial Parsing proceeds from the beginning, varying i from 1 thru n, applying the operators in the solution to the current state ("explanation" step of EBL), and collecting each operator subsequence that occurs between a state

[2] This can be generalized with parameterization.

Input problem $\langle s^0, g \rangle$,
  where $s^0 = \langle s_1^0, \ldots, s_n^0 \rangle$, and $g = \langle g_1, \ldots, g_n \rangle$,
   and solution $\beta = \langle o_1, \ldots, o_d \rangle$
For each k, let $s^k := o_k(o_{k-1}(\ldots(o_1(s^0))\ldots))$
Operator index $k := 0$
Current feature $i := 1$
Current value (of feature i) $j := s_i^k$
$macro :=$ ""
Do until $k > d$ or $i > n$
  If$\langle s_1^k, \ldots, s_i^k \rangle = \langle g_1, \ldots, g_i \rangle$
   Then Begin
    Let $M_{j,i} := macro$
    $i := i + 1$
    $j := s_i^k$
    $macro :=$ ""
    End Begin;
   Else Begin
    $k := k + 1$;
    $macro := macro.o_k$
    End Begin;
End Do;
Output macro table M

Table 2: Serial Parsing algorithm.

$\langle g_1, \ldots, g_{i-1}, j, \ldots \rangle$, and $\langle g_1, \ldots, g_{i-1}, g_i, \ldots \rangle$, and storing it as the macro $M_{j,i}$. If the domain is serially decomposable for the feature ordering $1 \ldots n$, this is guaranteed to yield correct macros. Note that the structure of the macro table eliminates the need to compute the preconditions for macros.

For example, in Eight Puzzle, let r, l, u, and d represent the primitive operators of moving a tile right, left, up, and down respectively. Macros are represented as strings made up of these letters. For notational ease, features (tiles) are labeled from 0 to 8, 0 standing for the blank and i for tile i. A macro $M_{j,i}$ represents the sequence of moves needed to get the i^{th} tile to the goal position from its current position j, while preserving the positions of all previous tiles including the blank. The positions are numbered by the tile numbers in the goal state, which is assumed to be fixed. The goal state and an example start state along with a solution are shown in Figure 1.

Given the problem and the solution in Figure 1, Serial Parser breaks down the solution as "*dr* $-$ *rdlu* $-$ *druuldrdlu* $-$ *uldrurdllurd* $-$ *urdl*" and extracts the macros: $M_{5,0}$ = "*dr*"; $M_{2,1}$ = "*rdlu*"; $M_{7,2}$ = "*druuldrdlu*"; $M_{3,3}$ = ""; $M_{4,4}$ = ""; $M_{7,5}$ = "*uldrurdllurd*"; and $M_{7,6}$ = "*urdl*". From many examples the system constructs a table of macros. The problem solver works as described in Definition 4. It fails if the macro table is missing some necessary macro at any point.

The result of this section can now be stated and proved. Unlike Theorem 3, the next theorem is distribution-independent.

Theorem 5 *If (a) H is the set of all problem solvers defined by the macro table bias for domains in M,*

6	1	3			1	2	3
8	4	7			8		4
2	5				7	6	5

Solution: drrdludruuldrdluuldrurdllurdurdl

Figure 1: A training example in Eight Puzzle

(b) the number of distinct feature values in D is bounded by a polynomial function of maximum problem size, and (c) the feature ordering of the target problem solver is input to the learner,
then stochastic testing with Serial Parsing is a learning algorithm for M in H.

Proof (sketch): We already saw that, if at some stage i, stochastic testing terminates, then, with a probability at least $1 - \delta$, it outputs a problem solver which has a probability of success of at least $1 - \epsilon$. We will now see that the number of examples and the running time of the algorithms are polynomially bounded.

Conditions (a) and (b) imply the existence of a macro table, whose size (number of macros in the table) is bounded by a polynomial in the problem-size. Conditions (a) and (c) guarantee that the Serial Parser would learn at least one missing macro from the example whenever the macro-based problem solver fails to solve a problem. Hence, by arguments similar to those of Theorem 3, the examples are bounded by $\frac{|M|}{\epsilon}(2\ln(|M|+1) + \ln\frac{1}{\delta})$, which is a polynomial. It also follows that the learner runs in polynomial time. Since the macro-based problem solver never needs to backtrack, it runs in time polynomial in all the relevant parameters.●

The above theorem shows that Serial Parsing exploits serial decomposability, a problem-space structure which allows it to compress the potentially exponential number of solutions into a polynomial size macro table.

The macro table bias requires that the teacher's solutions (provided by the SOLVED-PROBLEM) are composed from a macro table. This assumption is needed in the proof because Serial Parser can only learn a macro if the solution contains that macro. This assumption may not be true in general because, every problem might have several solutions, some of which may not be obtained by composing macros from a macro table. In fact, this assumption is most likely violated if the teacher gives only optimal solutions, because finding optimal solutions for the $N \times N$ generalization of Eight Puzzle is intractable [Ratner and Warmuth, 1986]. Hence the macros learned from optimal solutions do not fit into macro tables. However, if the learner is allowed to ask queries, i.e., ask the teacher to solve carefully designed problems, it is possible to learn macro tables in polynomial time irrespective of how the teacher solves the problems by designing a problem for each cell in the macro table and storing the solution as a macro.

ϵ	Examples			table	error	CPU
$=\delta$	training	test	total	size	%	msec
0.9	23.1	7.9	31.0	34.4	7.8	7.33
0.8	24.5	9.0	33.5	34.7	4.4	7.45
0.7	21.1	10.4	31.5	34.1	11.6	7.14
0.6	22.4	12.4	34.8	34.6	4.3	7.18
0.5	25.9	15.9	41.8	34.9	1.2	7.18
0.4	28.5	21.0	49.5	35.0	0.0	7.38
0.3	27.6	30.5	58.1	35.0	0.0	7.61
0.2	27.0	54.1	81.1	35.0	0.0	7.48
0.1	30.8	156.4	187.2	35.0	0.0	7.72

Table 3: Experimental Results on Eight Puzzle.

5 Experimental Results

Serial Parsing was implemented in the Eight Puzzle domain. The program SP (Serial Parser) was trained by randomly selecting a solvable problem (with uniform distribution), constructing a solution using a macro table, and giving them to the learner. SP was trained with stochastic testing for several values of ϵ and δ. After each training session, it was also tested on 100 independent random test examples to estimate the error rate. The results are shown in Table 3. Each row represents the averages after training the system 10 times with the same values of ϵ and δ.

The first column shows ϵ which is set to be the same as δ. The second column shows the total number of examples used until after the last macro was learned. The third column shows the number of examples used for stochastic testing after the last macro was learned. The fourth column shows the total number of examples used. The fifth column shows the total number of macros learned. The sixth column shows the error rate after learning was complete, averaged over 100 independent test problems. The final column shows the CPU msecs for solving a single problem after learning was complete, averaged over 100 problems. (The measurements were taken on a sun SPARCstation 1 with 8MB of real memory.)

The full macro table contains 35 macros. In most cases all 35 macros were learned even when ϵ and δ were set to high values. The number of examples increased with decreasing ϵ and δ, as can be expected. The most important thing to notice is that the system was able to learn the full macro table with approximately 30 training examples. With Macro Caching , $9!/2$ macros and many more examples would have been required for perfect learning!

It is clear from the last column that SP did not suffer from the utility problem. The CPU time was fairly constant with decreasing ϵ and δ. After learning was complete, SP was able to solve any randomly chosen solvable problem in less than 8 msecs of CPU time. The two main reasons for SP's speed are: (a) the macro-based problem solving is linear in the length of the solution, and (b) the operators and goals are represented efficiently and avoid the overhead of the logical representations.

6 Discussion and Related Work

Recently, there have been a few formal frameworks proposed to capture performance improvement learning. For example, [Cohen, 1989] analyzes a "Solution Path Caching" mechanism and shows that organizing the solution sequences of the examples in a tree and restricting the search of the problem solver to this tree improves the performance of the problem solver with a high probability. Solution Path Caching is similar to Macro Caching in that they both can be shown to be polynomial-time learning algorithms for domains defined by sparse solution space bias. By defining learning as producing a polynomial-time problem solver as opposed to simply running faster than the original problem solver, we have more stringent conditions on successful learning in our framework. Hence, Solution Path Caching (like Macro Caching but unlike Serial Parsing) fails to learn (by our definition) in domains like Eight Puzzle.

Greiner and Likuski formalize EBL as adding redundant learned rules to a horn-clause knowledge base to hasten query-processing [Greiner and Likuski, 1989]. This model is extended to recursive domain theories in [Subramanian and Feldman, 1990]. They conclude that learning macro-rules in EBL in such domains is not profitable in general unless strong assumptions are made about the problem distribution. While our result on Macro Caching is consistent with their conclusion, we also show that the structure of the problem-space can be exploited to make learning profitable in other domains.

Serial decomposability is just one example of a problem-space structure. [Etzioni, 1990] describes another kind of problem-space structure called "nonrecursive explanations" which explains the successful performance of PRODIGY in a number of domains. When this structure is present, the size of the explanation of a control heuristic is independent of the solution length. Somewhat surprisingly, Etzioni also found that examples were not the key reason for PRODIGY's successful performance [Etzioni, 1990]. Etzioni's program STATIC matches PRODIGY's performance by statically analyzing the problem-space without using any examples. Thus, even though [Etzioni, 1990] explains why PRODIGY works, it fails to explain the role of examples in EBL-type systems in distribution-independent learning. This paper shows that EBL can gain from the problem-space structure as well as examples. The examples play two roles: first, they provide distribution information that determines which macros are worth learning, and second, they help the learner avoid expensive search for macros.

Serial Parsing is given the order in which subgoals are achieved. In systems like SOAR that successfully learn macros using EBL, the goal ordering is implicitly given by defining the subgoals such that they include one another [Laird *et al.*, 1986]. In [Tadepalli, 1991], a method called Batch Parsing is described which learns the subgoal ordering along with macros for the subgoals. The basic idea here is to learn the macro table column by column, using multiple examples to disambiguate the feature that corresponds to a given column. [Chalasani *et al.*, 1991] describes algorithms that detect serial and total decomposability by experimentation.

One consequence of our formalization is that it blurs the distinction between EBL and empirical learning. To the extent that the output of an EBL method depends on examples, EBL is also an empirical method. We showed that EBL systems also have syntactic biases in that their performance is based on some assumptions about the structure of the target problem solver. The main difference between the "empirical" and the "explanation-based" approaches seems to be that the EBL systems also have a "semantic bias" in that their hypothesis space is also constrained by consistency with their domain theory.

7 Conclusions

The main contribution of this paper is an integration of EBL with formal machine learning. The most important result of our analysis is an explication of the biases that allow EBL methods to guarantee performance improvement in the limit. Conditions which were thought necessary for EBL to work, e.g., declarative representation of operators, were found to be not so crucial from our analysis. On the other hand, the structure of the problem-space and the distribution of problems were found to be very important. Our paper also integrated Korf's work on macro-operator learning with the EBL work.

Our work suggests that the best way to solve the utility problem is to implement a bias that exploits the domain structure. In the future, it is worthwhile to investigate the kinds of biases that occur in natural domains and implement them in new learning algorithms. As in empirical learning, this means that the program is then not suitable for domains which do not satisfy this bias. We think that the best way to address this generality issue is to build a variety of learning systems which are appropriate for domains with different problem-space structures.

Acknowledgments

I am indebted to Balas Natarjan for his generous help and advice throughout this work. I thank Prasad Chalasani, Tom Dietterich, Oren Etzioni, Nick Flann, Sridhar Mahadevan, Tom Mitchell, Balas Natarajan, and Armand Prieditis for many fruitful discussions. I also thank them, Barney Pell, and the anonymous reviewers of this paper for their comments. I thank Walter Rudd for his generous support and encouragement.

References

[Angluin, 1988] D. Angluin. Queries and concept learning. *Machine Learning*, 2, 1988.

[Chalasani *et al.*, 1991] P. Chalasani, O. Etzioni, and J. Mount. Detecting and Exploiting Decomposability in Update Graphs. In *2nd Int'l Conference on Principles of Knowledge Representation and Reasoning, KR'91.*

[Cohen, 1989] W. Cohen. Solution path caching mechanisms which provably improve performance. Technical Report DCS-TR-254, Rutgers University, July 1989.

[Dejong and Mooney, 1986] G. Dejong and R. Mooney. Explanation-based learning: A differentiating view. *Machine Learning*, 2, 1986.

[Etzioni, 1990] O. Etzioni. Why prodigy/ebl works. In *Proceedings of AAAI-90*, 1990.

[Greiner and Likuski, 1989] R. Greiner, and J. Likuski. Incorporating Redundant Learned Rules: A Preliminary Formal Analysis of EBL. In *Proceedings of IJCAI-89.*

[Haussler, 1990] D. Haussler. Probably Approximately Correct Learning. In *Proceedings of AAAI-90*, Boston, MA, August, 1990.

[Korf, 1985] R. Korf. *Macro-operators: A weak method for learning. Artificial Intelligence*, 26, 1985.

[Laird *et al.*, 1986] J. E. Laird, P.S. Rosenbloom, and A. Newell. Chunking in Soar: The Anatomy of a General Learning Mechanism. *Machine Learning*, 1, 1986.

[Minton, 1990] S. Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42, 1990.

[Mitchell *et al.*, 1986] T. Mitchell, R. Keller, and S. Kedar-Cabelli. Explanation based generalization: A unifying view. *Machine Learning*, 1, 1986.

[Natarajan and Tadepalli, 1988] B. Natarajan and P. Tadepalli. Two new frameworks for learning. In *Proceedings of Machine Learning Conference*, Ann Arbor, MI, 1988.

[Natarajan, 1989] B. Natarajan. On learning sets and functions. *Machine Learning*, 4, 1989.

[Ratner and Warmuth, 1986] D. Ratner, and M. Warmuth. Finding a shortest solution for the N X N extension of the 15-PUZZLE is intractable. In *Proceedings of AAAI-86*, Philadelphia, PA, 1986.

[Subramanian and Feldman, 1990] D. Subramanian and R. Feldman. The utility of ebl in recursive domain theories. In *Proceedings of AAAI-90*, Boston, MA, August 1990.

[Tadepalli, 1990] P. Tadepalli. *Tractable learning and planning in games.* Ph.D. Thesis, Department of Computer Science, Rutgers University, 1990.

[Tadepalli, 1991] P. Tadepalli. Learning with Inscrutable Theories. In Birnbaum, L. and Collins, G., (eds.) *Machine Learning: Proceedings of International Machine Learning Workshop*, San Mateo, CA: Morgan kaufmann.

[Valiant, 1984] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 11(27), August 1984.