# Learning to Solve Problems from Exercises

Prasad Tadepalli
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR 97331-5501
(541)-737-5552
(tadepall@eecs.oregonstate.edu)

October 13, 2008

**Abstract**

It is a common observation that learning easier skills makes it possible to learn the more difficult skills. This fact is routinely exploited by parents, teachers, text book writers, and coaches. From driving, to music, to science, there hardly exists a complex skill that is not learned by gradations. Natarajan's model of "learning from exercises" captures this kind of learning of efficient problem solving skills using practice problems or exercises (Natarajan, 1989). The exercises are intermediate subproblems that occur in solving the main problems and span all levels of difficulty. The learner iteratively bootstraps what is learned from simpler exercises to generalize techniques for solving more complex exercises. In this paper, we extend Natarajan's framework to the problem reduction setting where problems are solved by reducing them to simpler problems. We theoretically characterize the conditions under which efficient learning from exercises is possible. We demonstrate the generality of our framework with successful implementations in the Eight Puzzle, symbolic integration, and simulated robot planning domains illustrating 3 different representations of control knowledge, namely, macro-operators, control rules, and decision lists. The results show that the learning rates for the exercises framework are competitive with those for learning from problems solved by the teacher.

**Keywords:** Learning from exercises, speedup learning, PAC learning, macro-operator learning, control rule learning.

# 1 Introduction

Evidence from learning theory suggests that there are serious computational limits to what can be feasibly learned autonomously. And, yet, people seem to be able to effectively learn a variety of skills with little effort. If learning complex skills is computationally hard, one way to circumvent this complexity is by learning simpler skills first and using them effectively in learning more complex skills. Indeed, it appears that this fact about human learning is routinely exploited by teachers, curriculum developers, coaches, and the like. From such mundane skills as driving to creative arts like music and to abstract subjects like mathematics, there hardly exists an area of human expertise that is not taught by gradations. In this paper, we formally explore this idea in the context of machine learning of problem solving skills.

We are interested in *speedup learning*, which is the study of learning systems that improve the computational efficiency of problem solving with experience (Tadepalli & Natarajan, 1996). Rather than viewing speedup learning as decreasing the time needed to solve a problem using the same problem solver, we view it as acquiring control knowledge for a problem solver which has a better asymptotic complexity than the search-based "default problem solver."

Consider the domain of symbolic integration. Given the specification of this domain as a table of integration operators, we have complete information on how to solve any solvable problem. Yet, while we are capable of solving problems of symbolic integration *in principle*, we may not be very efficient at it. We need to try and solve example problems, and gradually build up heuristics that will enable us to solve the problems faster in the future. It also appears that we typically need to solve and study simpler problems first before we are able to solve more complex problems. The knowledge learned from the simpler problems helps us efficiently solve more complex problems and learn from them.

Speedup learning may be "supervised," where the learner is presented with sample problems and their solutions by a teacher (DeJong & Mooney, 1986; Mitchell, Keller, & Kedar-Cabelli, 1986; Shavlik, 1990), or "unsupervised," where the learner is only presented with example problems, which it has to solve on its own (Mitchell, Utgoff, & Banerji, 1983; Minton, Carbonell, Knoblock, Kuokka, Etzioni, & Gil, 1989; Laird, Rosenbloom, & Newell, 1986; Gratch & DeJong, 1992; Tadepalli, 1992). While supervised speedup learning can be efficient, it is burdensome to the teacher, who is typically a human. Unsupervised speedup learning is attractive, since it places the burden of solving the training problems completely on the learner. However, since the learner begins with no or little heuristic knowledge about how to go about solving difficult problems, it is faced with a computationally challenging task. A similar problem exists in *Reinforcement Learning*, where the learner has to rely on trial and error search and delayed feedback to improve its performance (Sutton & Barto, 1998). If the feedback is not frequent or does not truly reflect the long-term benefits to the agent, the learner has to explore for a long time before it succeeds in achieving its final goal.

In previous work, Natarajan introduced the framework of "learning from exercises," which takes a middle course to resolve this dilemma (Natarajan, 1989). Instead of demanding that the teacher should supply solutions to the training problems, in this framework, the teacher is required to provide a set of "exercises" which are at varying levels of difficulty. This is very much similar to the exercises one might find at the end of a text book dealing with say symbolic integration or differential equations. The burden of solving the exercises is entirely on the learner. The learner solves the exercises in the increasing order of difficulty, so that the knowledge learned from the simpler exercises helps it to solve more difficult ones, and eventually the most difficult "natural" problems in the domain.

Exercises play a role similar to tasks and subtasks in hierarchical reinforcement learning and hierarchical planning (Dietterich, 2000; Erol, 1995; Marthi, Russell, & Wolfe, 2007). It has been found that designing suitable task hierarchies improves the efficacy of planning and reinforcement learning in two ways. First, task hierarchies help decompose a hard task into simpler subtasks, which have smaller solution lengths, and hence involve exponentially smaller search. In reinforcement learning, this reduces the delay between successive reward signals, and helps guide the learner in fruitful directions. Indeed, in the absence of such decomposition of large tasks into smaller subtasks or intermediate shaping rewards that implicitly define the subtasks, the learner is condemned to searching an exponentially large space with no hope of ever learning anything useful without external guidance. Second, they help the learner generalize its experience more effectively, since simpler subtasks usually involve fewer constraints on when they are applicable. For

example, in chess, it is easier to learn the conditions under which a knight-fork may be successful, than it is to learn the conditions under which the game may eventually lead to a won. The existence of a suitable task hierarchy is a requirement of the effectiveness of our approach as well as other hierarchical approaches to learning and planning.

We make three major contributions in this paper. First, we formalize the problem of learning from exercises in a more general problem reduction setting compared to the state-space formulation of Natarajan (1989). Second, we give a novel proof for the tractability of learning in this new setting based on Valiant's Probably Approximately Correct (PAC) learning framework (Valiant, 1984). The new proof is simpler and more general than the previous result of Natarajan. Third, we demonstrate the usefulness and generality of this framework through empirical results in three different domains with different hypothesis languages.

Our framework is based on the assumption that there is a natural hierarchy of problem solving stages, which is used to generate exercises. Each problem solving stage is associated with a difficulty level. Exercises that belong to the higher levels of the hierarchy contain those in the lower levels as subproblems and are more difficult. In our problem reduction setting, the solutions to problems are most naturally represented as trees. The nodes of the tree correspond to problems and the children of each node correspond to its subproblems, all of which should be solved to solve the problem. We call such trees solution trees. The theory requires that the exercises in the different stages should be such that the learner must be able to reduce exercises of higher difficulty level to those of less total difficulty in polynomial time using a default problem solver, which is usually based on some kind of search.

We describe an algorithm schema that works as follows. It collects a set of exercises and their difficulty levels using an oracle, and sorts them according to their difficulty. It then calls a routine that solves and generalizes the exercises of each stage in the increasing order of their difficulty. The knowledge learned from the solutions of easier exercises helps solve more difficult exercises by reducing them to problems of less difficulty using the default problem solver. Our main result is to show that if there is such a routine which can efficiently reduce exercises of any difficulty level to those of less *total* difficulty and generalize them, then our algorithmic schema can be used to efficiently learn problem solving from exercises. The routine that reduces, solves and generalizes the exercises of each stage is dependent on the representation of the control knowledge learned. We develop such routines for three different representations of control knowledge, namely, macro-operators, control rules, and decision lists. We provide empirical evidence for the effectiveness of our algorithms through experiments in three different domains: Eight Puzzle, symbolic integration, and robot planning. In all these cases, we show that learning from exercises performs competitively with learning from teacher-provided solutions of natural problems.

The rest of the paper is organized as follows. Section 2 introduces the formal preliminaries of problem solving using a problem reduction framework. This is followed by Section 3 with an introduction to our formal framework for speedup learning from exercises and a main theorem that characterizes the conditions under which such learning is possible. The next three sections instantiate our framework with algorithms and implementations that work with three different kinds of control knowledge, namely, macro-operators, control rules, and decision lists, as applied to Eight Puzzle, symbolic integration and robot planning domains respectively. Section 7 discusses the related work and limitations of the current work, followed by a discussion of future directions in Section 8.

## 2 Preliminaries

While state-space search has been the dominant paradigm for problem-solving, the problem reduction framework is more general and natural for our purpose.

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

- $S$ =A set of *problem instances*, also called *problems.*
- $G$ = A goal procedure that recognizes problem instances that are trivially solved and returns their solutions.

1. $\int kf(x)dx \rightarrow k\int f(x)dx$
2. $\int f(x) - g(x)dx \rightarrow \int f(x)dx - \int g(x)dx$
3. $\int f(x) + g(x)dx \rightarrow \int f(x)dx + \int g(x)dx$
4. $\int f(x) * g(x)dx \rightarrow g(x)\int f(x)dx + f(x)\int g(x)dx$
5. $\int x^n dx \rightarrow x^{(n+1)}/(n+1)$
6. $\int \sin x dx \rightarrow (-\cos x)$
7. $\int \cos x dx \rightarrow \sin x$

Figure 1: A table of integration operators, reproduced from (Tadepalli and Natarajan, 1996).

- $O$ = A set of *operators* $\{o_1, \ldots, o_k\}$, where each $o_i$ is a procedure that computes the set of immediate subproblems of any given problem $x \in S$.

Our operators reduce a given problem to a set of subproblems in $S$. If $o_i$ reduces a problem $s$ to subproblems, $s_1, \ldots, s_j$, we write $o_i(s) = \{s_1, \ldots, s_j\}$, and we say that $s$ directly reduces to the set $\{s_1, \ldots, s_j\}$. For this decomposition to be successful, i.e., to finally yield a solution to the original problem, all the subproblems $s_1, \ldots, s_j$ must be solved.

A *meta-domain* is a set of domains defined over the same set of states. A domain specification is a compact representation of procedures that implement the operators in $O$ and $G$, all of which are assumed to run in unit time.[1] Further, we assume that the solution to the original problem can be easily composed from the solutions of the subproblems using a fixed polynomial-time procedure, known to the learner. More formally, we assume that the learner has access to a polynomial-time procedure, COMPOSE, such that the following holds for every problem $s \in S$ and every operator $o_i \in O$.

If $o_i(s) = \{s_1, \ldots, s_j\}$, and if $Solution(s_1), \ldots, Solution(s_j)$ exist and represent the solutions of $s_1, \ldots, s_j$ respectively, then $Solution(s) = \text{COMPOSE}(o_i, Solution(s_1), \ldots, Solution(s_j))$.

The *problem size* is a syntactic measure of the complexity of a problem such as its length. The size of a problem $s$ is denoted by $|s|$.

A *problem solver* $f$ for a domain $D$ is a deterministic program that takes as input an instance, $s$, and computes its solution, if one exists. A *hypothesis space* $\mathcal{F}$ is a set of problem solvers whose run time is polynomial in the size of the problem. The restriction of problem solvers to the subset of $\mathcal{F}$ which apply to problem instances of size $\leq n$ is called a *subspace* of $\mathcal{F}$ and is denoted by $\mathcal{F}_n$.

For example, consider the integration operator 3 in Figure 1, which decomposes $\int f(x) + g(x)dx$ into two problems $\int f(x)dx$ and $\int g(x)dx$, whose solutions are composed by addition. Given a problem, $\int \sin x + \cos x dx$, this operator decomposes it to two problems $\int \sin x dx$ and $\int \cos x dx$. These two problems are further decomposed by operators 6 and 7 into $(-\cos x)$ and $\sin x$ respectively. The goal procedure recognizes these instances as trivially solved since they are in their simplest forms, and returns them. COMPOSE puts these two solutions together, yielding $(-\cos x) + \sin x$.

Since the learner has access to the COMPOSE procedure, the main task remaining is to learn to recursively decompose a problem into its subproblems until they all reduce to primitive problems. This is nontrivial because some of the subproblems of a problem may not have a solution, in which case the problem solver has to backtrack and try a different reduction of the original problem. Searching exhaustively in the space of all possible reductions is intractable, and hence there is need for learning control knowledge that guides the search and makes problem solving more efficient.

Note that while this problem reduction approach works perfectly when subproblems are completely independent, it can also be used when the subproblems have minor interactions that can be "fixed" by the COMPOSE procedure. For example, one of the fixes might be to order the solutions for the subproblems correctly. Another may be to fill in small gaps in the plan, such as releasing and reallocating the resources.

[1]This can be generalized by making their run time $t$ a parameter of the learning problem, as in (Tadepalli & Natarajan, 1996).

While a complete search procedure requires backtracking over the subproblem solutions in general, here we only consider domains where this can be done without backtracking using appropriate control knowledge.

## 3 A Model of Learning from Exercises

In this section, we introduce our refinement of Natarajan's model of "Learning from Exercises" (Natarajan, 1989). This model distinguishes between the so called "natural problems" that the learner will be required to solve during testing, and the unsolved "exercises" that are specifically designed for training. Although, these two distributions are different, solving exercises would eventually help the learner solve the problems from the natural distribution. Moreover, learning is rendered computationally tractable when it is based on the exercises. In the following, we make these intuitions more precise.

We define a *stage* to be a set of problem instances. A stage structure $\mathcal{G}$ over a set of problem instances $S$ consists of (a) a partitioning of the solvable problem instances in $S$ into a set of mutually exclusive stages, $G_1, \ldots, G_L$, and (b) a partial ordering $\geq$ which means "is at least as difficult as," over these stages. By convention, $G_1$ is the set of all problems solved by the goal procedure $G$ and the stages are numbered such that if the stage $G_i \geq G_j$ then $i \geq j$.[2] Further, $i$ represents the *difficulty* of the problems $x$ in stage $G_i$, and is denoted by difficulty($x$). $i$ is also called the *difficulty level*, or more simply, the *level* of $x$. By definition, all problems in the same stage are of the same difficulty level. Hence $i$ is also called the level of $G_i$.

We only consider problem solvers $\mathcal{F}$ that are "monotonic" with respect to the stage structure in the following sense. A problem reduction step of a *monotonic problem solver* $f \in \mathcal{F}$ consists of reducing a problem $x$ to a set of problems $R_f(x)$, whose sum of all difficulty levels is *strictly less than* that of the original problem. $R_f(x)$ is called the reduction set of $x$ with respect to $f$. The problem solvers in $\mathcal{F}$ are called monotonic because they recursively apply problem reduction steps to each of the subproblems of the original problem until they are all trivially solved. Note that each reduction step may in fact be composed of one or more problem reduction operators applied in a sequence or in the form of a tree. After the problem is fully decomposed, the solution to the original problem instance is computed by composing the solutions in a bottom-up manner.

By the property of monotonicity, we have the following for any problem $x$ solved by a monotonic problem solver.

$$\text{difficulty}(x) \geq 1 + \sum_{y \in R_f(x)} \text{difficulty}(y) \tag{1}$$

Let $f$ be a problem solver and $x$ be a problem. An *exercise tree* of $x$ with respect to $f$ is the tree which is rooted at $x$, where the children of each internal node $y$ (including $x$) represent its reduction set with respect to $f$, and the leaves are the trivially solved problems. The bag of all problems which are represented by the nodes of the exercise tree of $x$ with respect to $f$ is called its *exercise bag* and is denoted by $E_f(x)$.

The key idea of our framework is to provide the learning algorithm with a source of exercises of varying difficulties. This will permit the learning algorithm to consider increasingly difficult problems, using what it learned from the simpler problems to efficiently solve more difficult problems, improving its capabilities as it progresses.

**Proposition 1** *The difficulty of a solvable problem $x$ by a monotonic problem solver $f$ is an upper bound on the number of nodes in the exercise tree of $x$ with respect to $f$.*

Proof: We need to show difficulty($x$) $\geq |E_f(x)|$ for all problems $x$.

We prove the result by structural induction on the exercise tree. As the basis, a problem that is trivially solved, i.e., in stage $G_1$, has only a single node in its exercise tree, and difficulty($x$) $= |E_f(x)| = 1$.

By inductive hypothesis, we have, difficulty($y$) $\geq |E_f(y)|$ for all children $y$ of $x$ in the exercise tree. Hence, substituting $|E_f(y)|$ for difficulty($y$) in Equation 1, we have

[2]Note that the converse may not hold unless the stages are also totally ordered.

$$\text{difficulty}(x) \geq 1 + \sum_{y \in R_f(x)} |E_f(y)| \tag{2}$$

The right hand side in the above equation represents $|E_f(x)|$, proving the claim. •

Let $P$ be a probability distribution on the set of all problem instances $S$, according to which the learner is to be tested. We call this the *natural* distribution, and the set of instances selected using this distribution, the *natural* problems. Let $f$ be a target problem solver, and $f(x)$ be the solution of $x$ given by the target problem solver.

The exercises are problems in the exercise trees of natural problems with respect to the target problem solver. When called, EXERCISE outputs a problem instance $y \in E_f(x)$ and its difficulty level $l$, where $x$ is a natural problem and $f$ is the target problem solver.

We define the exercise distribution $P_e$ as the uniform distribution from the exercise bag of natural problems chosen using $P$. In other words, EXERCISE chooses a random problem $x$ with probability $P(x)$, and uniformly chooses from the exercise bag of $x$ with respect to the target problem solver $f$. Hence, if $|E_f(x)|$ is the size of the exercise bag and $\#(y \in E_f(x))$ denotes the number of different instances of a particular exercise $y$ in the bag, we have:

$$P_e(y) = \sum_{x \in S} \frac{P(x)\#(y \in E_f(x))}{|E_f(x)|} \tag{3}$$

Note that $P_e$ is a well-defined distribution over the instances, since the sum of $P_e$ over all instances is 1.

$$\sum_{y \in S} P_e(y) = \sum_{y \in S} \sum_{x \in S} \frac{P(x)\#(y \in E_f(x))}{|E_f(x)|} = \sum_{x \in S} \frac{P(x)}{|E_f(x)|} \sum_{y \in S} \#(y \in E_f(x)) = \sum_{x \in S} \frac{P(x)}{|E_f(x)|} |E_f(x)| = \sum_{x \in S} P(x) = 1$$

Exercises are related to the stage structure because the difficulty of a problem is defined with respect to a stage structure, and the total difficulty of the problems in the reduction set of a problem is supposed to strictly reduce in each step of any monotonic problem solver.

This brings up the question of how to set up the stage structure. A trivial stage structure might just have two stages, one containing the trivially solved problems and the other containing all solvable problems. But a "good" stage structure is one where it is possible to reduce any instance of a problem to a set of problems with less total difficulty with a "reasonable" computational effort (search) by the default problem solver, which is available during learning. Exercises are problem instances chosen from exactly such stage structures. Note that this notion of exercises is more general than that of Natarajan (1989) in a number of ways. First, our problem reduction framework allows reducing a problem to a set of subproblems rather than a single subproblem unlike in the state space search formulation of (Natarajan, 1989). Second, the definition of problem solving difficulty is based on a set of target problem solvers which do not necessarily produce optimal solutions. This is to allow the learning of non-optimal problem solvers, where optimal problem solvers are not efficient or not compactly representable. Third, we do not assume that the problem instances in successive stages are one operator away. We only assume that the exercises can be reduced to successive stages in polynomial time by some default problem solver.

This last generalization is motivated by domains like Eight Puzzle, where problem solving effort is reduced by finding problem solving stages which are not very far off from each other (Korf, 1985). In Eight Puzzle, these stages correspond to successively getting a set of tiles, first the blank, next the blank and the tile 1, then the blank and tiles 1 and 2, and so on to their goal positions until all the tiles are in their goal positions. Although there is no monotonic improvement in the problem state within a single operator application, there is such an improvement between successive stages of problem solving. The key insight of Korf, which also forms the basis of our framework, is that when the stages are such that it is possible to move from one stage to the next with a reasonable amount of search, giving the exercises is almost as useful as providing solutions, since the gaps between exercises can be filled in easily by the learner and can be learned from. This is more generally captured by the following constraint on the stage structure.

**Definition 1** *Let a* default problem solver *(DPS) be a problem solver which takes as a problem $e$ and an approximate problem solver $h_{l-1}$ for problems of level $l-1$, and returns a solution obtained by reducing $e$ to its subproblems solvable by $h_{l-1}$. Let $f$ be a target problem solver in $\mathcal{F}$. A stage structure $\mathcal{G}$ of $f$ is* dense *for DPS with respect to $\mathcal{F}$ if there is a constant $c$ independent of problem size $n$ and level $l$ such that given an exercise $e$ in $G_l$ and an approximation $h_{l-1}$ of $f$ that can solve the problems in $R_f(e)$, DPS can correctly solve $e$ in time $O(n^c)$.*

The target problem solver is any problem solver in the learner's hypothesis space. The target problem solver is used to define a useful goal structure, which in turn determines a valid exercise distribution. The default problem solver is typically an exhaustive search program the learner has access to. Intuitively, a dense goal structure requires the different subgoals of the problem to be close to each other so that it is tractable to fill-in the gaps between the different subgoals by the default problem-solver. The density of goal structure is similar to the density of rewards in reinforcement learning. Highly sparse reward or sparse subgoals makes a hard reinforcement learning problem because the learner is typically condemned to do exhaustive search between successive rewards. A good strategy for the teacher to provide reasonable exercise problems is to make sure that each new exercise only requires a small amount of search, given the learner's current knowledge.

We now define the notion of a learning algorithm for a meta-domain in this setting. This definition is similar to that of (Tadepalli & Natarajan, 1996), except for the use of exercises instead of solved problems, and the use of problem reduction framework as opposed to the state space search framework.

**Definition 2** *An algorithm $\mathcal{A}$ is a learning from exercises algorithm for a meta-domain $\mathcal{M}$ in a hypothesis space $\mathcal{F}$ of deterministic polynomial-time problem-solvers if for any domain $D \in \mathcal{M}$, any choice of a problem distribution $P$, and any target problem solver $f \in \mathcal{F}$, and stage structure $\mathcal{G}$,*

1. *$\mathcal{A}$ takes as input the specification of a domain $D \in \mathcal{M}$, maximum problem size $n$, maximum difficulty level $L$, an error parameter $\epsilon$, and a confidence parameter $\delta$,*

2. *$\mathcal{A}$ may call EXERCISE and DPS, the default problem solver. The number of oracle calls made by $\mathcal{A}$ and its running time must be a polynomial function $Q$ of the problem size $n$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, and $L$.*

3. *For all $D \in \mathcal{M}$ and all probability distributions $P$ over $D$, with probability at least $(1-\delta)$, $\mathcal{A}$ outputs a program $h$ that approximates $f$ in the sense that $\sum_{x \in \Delta} P(x) \leq \epsilon$, where $\Delta = \{x | h \text{ fails on } x \text{ while } f \text{ succeeds}\}$.*

4. *There is a polynomial $R$ such that, for a maximum problem size $n$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, maximum difficulty level $L$, if $\mathcal{A}$ outputs $h$, $h$ runs in time $R(n, L, \frac{1}{\epsilon}, \frac{1}{\delta})$.*

There are a few things that should be noted about this framework. First, the learning algorithm is a function of the hypothesis space to which the target problem solver belongs. The stage structure may be defined with respect to a single target problem solver or it can be more generally applicable to any problem solver in the hypothesis space. Just as in the standard PAC learning literature, we require learning to be successful independent of the teacher's choice of test distribution $P$ and the target problem solver $f$. The problem solver $h$ output by the learner is said to approximate the target problem solver, if it solves a problem successfully whenever the target problem solver $f$ succeeds with probability no less than $1-\epsilon$, when tested on the problems chosen from the distribution $P$. Since the system relies on random exercises to learn from, it may not be able to learn such a problem solver when its training exercises are not representative. For learning to be considered successful, an approximate problem solver should be learned at least with a probability $1-\delta$.

Finally, there is the requirement that the learned problem solver must be efficient, which we take to mean polynomial in various parameters. For simplicity of exposition, this framework assumes that the maximum problem size $n$ and the number of stages $L$ are given. For a given problem distribution, these can also be easily estimated from examples (Natarajan, 1989).

```
01 procedure ExerciseLearner
02   input ε, δ, D = (G, O), the problem size, n, the number of stages L in the stage structure;
03   initialize H_0 to the goal procedure G;
04   initialize Exs(l) to { } for all l;
05   repeat (L^2/ε)(ln B_n + ln (L/δ)) times
06           Let ⟨l, ex⟩ := EXERCISE().
07           Add ex to Exs(l)
08   end repeat;
09   for l := 1 to L do
10       H_l := Solve-and-Generalize(H_{l-1}, Exs(l));
11   output H_l;
12 end ExerciseLearner
```

Figure 2: Algorithm schema for learning from exercises

We now investigate the sufficient conditions for a meta-domain to have an algorithm for learning from exercises. In particular, we examine the conditions under which the algorithm schema ExerciseLearner in Figure 2 can successfully learn under this framework.

ExerciseLearner takes the domain specification, problem size, the maximum difficulty level of any solvable problem, and the error parameters as its inputs (line 02 of Figure 2). It initializes the hypothesized problem solver $H_0$ to the goal procedure $G$, which recognizes and solves the trivial problems (line 03). It then collects a sufficiently large set of exercises using the EXERCISE oracle, orders them according to their difficulty level $l$ and calls a routine Solve-and-Generalize on these exercises in the order of increasing difficulty (lines 05-10). After processing the exercises for levels 1 through $L$, it terminates, and outputs $H_L$ (line 11). Solve-and-Generalize is dependent on the representation used for the control knowledge of the problem-solver. This routine in turn enlists the service of the default problem solver (DPS) and $H_{l-1}$ to solve them, and generalizes the result to get a new problem solver $H_l$ that is good for levels 1 thru $l$.

One of the problems in learning from exercises is the existence of multiple solutions to an exercise. An arbitrary solution of an exercise may not fit well with another arbitrary solution of a different exercise in the sense that there may not exist a single efficient problem solver in the learner's search space that can produce both these solutions (Tadepalli & Natarajan, 1996). This is one reason to require that Solve-and-Generalize outputs a single problem solver that solves *a set of* exercises. The efficiency of this search is a function of the domain structure, the hypothesized space of target problem solvers, as well as the number of reduction operator applications needed to reduce exercises to those of less total difficulty.

Normally, in PAC learning, we use the number of hypotheses in the hypothesis space to bound the number of examples needed to learn. However, in speedup learning, there may be multiple hypotheses (problem solvers) which are all equally good in that they all give correct solutions. Thus, rather than counting different hypotheses, we will get a better bound by counting the number of distinct "failure regions" in the input space, where hypotheses might fail with a probability $> \epsilon$. Given the target and the current set of hypotheses at stages 1 thru $l$, the failure region is determined (see below for an exact definition). Hence, given the target and the hypothesis $H_{l-1}$, each candidate $H_l$ has exactly one such region. Since multiple candidate hypotheses might have the same region, the number of failure regions is upperbounded by the number of hypotheses, which leads to a tighter bound on the sample complexity.

We define the *failure region* of a hypothesis $H_l$ as the set of exercises $x$ with a non-zero probability on which the hypothesis $H_l$ fails, while all problems in its reduction set with respect to the target $f$, i.e., $R_f(x)$, are solvable by $H_{l-1}$.

Ideally, we wish to learn a hypothesis that does not have any failure region, or one with a low probability failure region. To guarantee this, we need to sample the exercise distribution a number of times determined by the number of such failure regions (Kearns & Vazirani, 1994). This is formalized by the second condition

of the following main theorem.

**Theorem 1** *ExerciseLearner is an algorithm for learning from exercises for a meta-domain $\mathcal{M}$ in the hypothesis space $\mathcal{F}$ of deterministic polynomial-time problem solvers with a stage structure $\mathcal{G}$, if*

- *there is a routine* Solve-and-Generalize *available to the learner, which, for any target problem solver $f \in \mathcal{F}_n$ and difficulty level $l > 0$, given the current problem solver $H_{l-1}$, and some $m_l$ exercises of difficulty $l$ as input, runs in time polynomial in parameters $n$, $l$, and $m_l$, and outputs a problem solver $H_l \in \mathcal{F}_n$ that (a) correctly solves all exercises of difficulty $< l$ that are solvable by $H_{l-1}$, and (b) correctly solves all exercises of difficulty $l$ that are solvable by $f$ if their reduction sets with respect to $f$ are solvable by $H_{l-1}$; and*
- *there is a function $B_n$ of $n$, where $\ln B_n$ is polynomial in $n$, which upper bounds the number of failure regions of all the hypotheses in the learner's hypothesis space for any target function.*

**Proof:** The learning algorithm works by incrementally generalizing the problem solver using increasingly difficult exercises and their solutions.

Let the stage structure used be $G_0, \ldots, G_L$. Let $C_l = \{x | H_l \text{ correctly solves } x \in G_l\}$. $C_l$ represents the set of problems in $G_l$ that $H_l$ can correctly solve. The difference between the sets $G_l$ and $C_l$ represents the set of problems of difficulty $l$, which cannot be solved by the problem solver $H_l$. We want to show that, after learning, with a high probability, the chance of a random exercise falling in this set is bounded from above.

More specifically, we first bound (see the claim below) the probability of a random exercise of level $l$ not being correctly solved by $H_l$ after learning is complete. We use induction on $l$, since the exercises of level $l$ are solved by reducing them to exercises of lower levels, and hence the accuracy of level $l$ depends on the accuracies of lower levels. After this bound is proved, we use it to bound the probability that any test problem chosen using the natural distribution is not solved by the learned problem solver. This is the test error of the problem solver. Both the inductive claim and the final bound on the test error depend on the number of exercises used in training, and only hold with certain probability, since there is some chance that the training exercises are unrepresentative. We show that if the number of exercises is chosen as in the algorithm of Figure 2, then the test error of the problem solver is bounded from above by $\epsilon$ with a probability at least $1 - \delta$.

**Claim:** Under the conditions of the theorem, with probability $\geq 1 - \frac{\delta l}{L}$, for all difficulty levels $l \geq 0$, the following equation holds simultaneously.

$$\sum_{x \in G_l - C_l} P_e(x) \leq \frac{\epsilon l}{L^2} \tag{4}$$

**Basis:** For $l = 0$, the algorithm outputs the goal procedure, which always correctly solves the primitive problems by definition.

Hence the lhs of Equation (4) is 0 and the rhs is $\geq 0$.

**Induction step:** Assume that the theorem holds for all difficulty levels $< l$. We prove it for level $l$.

We call any problem solver $H_l$ which does not satisfy Equation 4 a "bad" problem solver for iteration $l$. All other problem solvers are good. We estimate the probability of learning a bad problem solver $H_l$, starting with a good problem solvers $H_{l-1}$.

By inductive hypothesis, for all $j < l$, if the problem solver $H_j$ is good,

$$\sum_{x \in G_j - C_j} P_e(x) \leq \frac{\epsilon j}{L^2}. \tag{5}$$

Let $x$ be an exercise in $G_l$, and let $f$, the target problem solver, reduce it to $R_f(x) = \{x_1, \ldots, x_k\}$, in one reduction. Let $l_1, \ldots, l_k$ be the difficulties of problems $x_1, \ldots, x_k$ respectively. By inductive hypothesis,

$$\sum_{x_i \in G_{l_i} - C_{l_i}} P_e(x_i) \leq \frac{\epsilon l_i}{L^2} \tag{6}$$

Note that, since $f$ is deterministic, it always decomposes $x$ in the same way. Hence every time $x$ occurs in a decomposition tree, $x_i$ will be its $i^{th}$ child. Since the exercise distribution selects each reduction of $x$ with uniform probability, $x_i$ occurs in the exercise distribution with probability at least as much as that of $x$. In addition, $x_i$ might also be in the reduction sets of other problems, and hence might occur with more probability than $x$. Hence, we can say that $P_e(x) \leq P_e(x_i)$, and so,

$$\forall i \geq 0, \sum_{x_i \in G_{l_i} - C_{l_i}} P_e(x) \leq \frac{\epsilon l_i}{L^2} \tag{7}$$

Since $x_1, \ldots, x_k$ are children of $x$ of difficulties $l_1, \ldots, l_k$ respectively,

$$\sum_{i=1,k} \sum_{x_i \in G_{l_i} - C_{l_i}} P_e(x) \leq \sum_{i=1,k} \frac{\epsilon l_i}{L^2} \tag{8}$$

Since $l \geq l_1 + \ldots l_k + 1$ from Equation 1, if $H_{l-1}$ is a good problem solver, then

$$\sum_{\exists i s.t. x_i \in G_{l_i} - C_{l_i}} P_e(x) \leq \sum_{i=1,k} \sum_{x_i \in G_{l_i} - C_{l_i}} P_e(x) \leq \frac{\epsilon(l-1)}{L^2}. \tag{9}$$

The left hand side of the above equation denotes the probability that a randomly chosen exercise of difficulty $l$ has a member in its reduction set that cannot be solved by the current problem solver, $H_{l-1}$.

Since the problem solver $H_l$ is assumed to be bad,

$$\sum_{x \in G_l - C_l} P_e(x) > \frac{\epsilon l}{L^2} \tag{10}$$

We want to bound the probability of a randomly chosen exercise $x$ of difficulty $l$ being not solvable by $H_l$, even though all the members of its reduction set $R_f(x)$ are solvable by $H_{1-1}$ and hence by $H_l$ by the property (a) of Solve-and-Generalize stated in Theorem 1. This means that the learner has not seen the relevant exercise for solving problem $x$. The difference in the two probability bounds of Equations 10 and 9 bounds the probability of this occurrence.

$$\sum_{x \in G_l - C_l, \text{and} \forall i, x_i \notin G_i - C_i} P_e(x) > \frac{\epsilon l}{L^2} - \frac{\epsilon(l-1)}{L^2} = \frac{\epsilon}{L^2} \tag{11}$$

By the first condition of the theorem, Solve-and-Generalize ensures that $H_l$ solves all exercises of level $l$ which are solvable by $f$ if their reduction sets are all in $H_{l-1}$. In other words, none of the $m$ exercises satisfies the conditions stipulated in Equation 11, whose probability mass is at least $\frac{\epsilon}{L^2}$.

Since the exercises are chosen independently using identical distribution, the probability of the learned hypothesis $H_l$ to possess *a particular* failure region is $\leq (1 - \frac{\epsilon}{L^2})^m$. Since $B_n$ is an upper bound on all non-zero probability failure regions, the probability of the learned hypothesis to have *any* failure region is at most $B_n(1 - \frac{\epsilon}{L^2})^m$.

We eliminate the probability of learning a bad problem solver with confidence $(1 - \frac{\delta}{L})$ by making the above quantity less than $\frac{\delta}{L}$. That is, by making

$$B_n \left(1 - \frac{\epsilon}{L^2}\right)^m < \frac{\delta}{L}$$

which can be satisfied by

$$m \geq \frac{L^2}{\epsilon} \left(\ln B_n + \ln \frac{L}{\delta}\right)$$

This is the sample size used by the ExerciseLearner. Hence,

$$Pr\{H_l \texttt{ is bad} | H_{l-1} \texttt{ is good}\} < \frac{\delta}{L}$$

By inductive hypothesis,

$$Pr\{H_{l-1} \texttt{ is bad}\} < \frac{\delta(l-1)}{L}$$

Hence,

$$Pr\{H_l \texttt{ is bad}\} < \frac{\delta}{L} + \frac{\delta(l-1)}{L} = \frac{\delta l}{L}$$

proving the induction for level $l$. For $l = L$ the above reduces to

$$Pr\{H_L \textit{ is bad}\} < \delta \tag{12}$$

This completes the induction. To show that the test error probability of the final problem solver is less than $\epsilon$, we use Equation 3. We can relate the two distributions $P$ and $P_e$ by the following equation.

$$P_e(y) = \sum_{x \in S} \frac{P(x)\#\{y \in E_f(x)\}}{|E_f(x)|} \geq \frac{P(x)\mathbf{1}_{y \in E_f(x)}}{|E_f(x)|} \geq \frac{P(x)\mathbf{1}_{y=x}}{|E_f(x)|} \geq \frac{P(y)}{L} \tag{13}$$

Here $\mathbf{1}_{C(x)}$ is an indicator function which is 1 if the condition $C(x)$ is true and 0 otherwise. The first two inequalities follow from the facts that $\#\{y \in E_f(x)\} \geq \mathbf{1}_{y \in E_f(x)}$ and that $x \in E_f(x)$ for all $x$. The final inequality follows from $|E_f(x)| \leq L$ for all $x$. From Equation 4, with probability $\geq 1 - \frac{\delta l}{L} \geq 1 - \delta$,

$$\sum_{x \in G_l - C_l} P_e(x) \leq \frac{\epsilon l}{L^2} \tag{14}$$

Since $\frac{P(x)}{L} \leq P_e(x)$, simplifying and canceling $L$ on both sides we get

$$\sum_{x \in G_l - C_l} P(x) \leq \frac{\epsilon l}{L} \leq \epsilon \tag{15}$$

Since all difficulty levels are mutually exclusive, this shows that the probability that a random test problem $x$ cannot be solved by the learned problem solver while it can be solved by the target problem solver is bounded by $\epsilon$. Since Equation 15 holds with a probability at least $\geq 1 - \delta$, condition 3 of Definition 1 is satisfied by ExerciseLearner. Since $\ln B_n$ is polynomial in $n$, the sample size of Figure 2 is polynomial in all the required parameters. If Solve-and-Generalize runs in time polynomial in all the parameters, then so does ExerciseLearner. Since all problem solvers in $\mathcal{F}$ run in time polynomial in the problem size, it follows that ExerciseLearner is a learning algorithm for $\mathcal{F}$. •

# 4 Learning macro-operators

One standard technique to make problem solvers efficient is to learn macro-operators (Korf, 1985; Iba, 1989; Fikes, Hart, & Nilsson, 1972). In this section, we consider efficient learning of macro-operators from exercises. First we describe a formalization of macro-operator learning in serially decomposable domains using exercises and prove a positive result for the learning by exercises model. Then we show some empirical results in the domain of Eight Puzzle.

## 4.1 Problem solving with macro-operators

A macro-operator (or a macro) is any sequence of operators that achieves a subgoal (Korf, 1985; Iba, 1989). Macro-operators make the grain size of the search space coarser than the space of primitive operators, thereby increasing the efficiency of search. In this section, we consider the learning of macro-operators and formalize it using the learning framework that we introduced in Section 3.

Macro-operators are especially effective for serially decomposable domains, which will be defined shortly. These domains have factored state spaces, where each state is described by a vector of features, $\langle f_1, \ldots, f_n \rangle$. Each feature takes discrete values from $\{v_1, \ldots, v_n\}$.

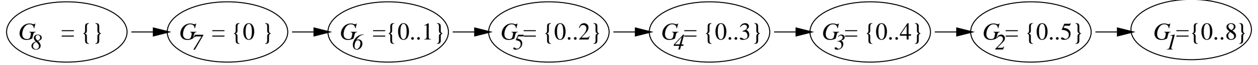


Figure 3: The stage structure of 8-puzzle. In any stage $G_l = \{0..i\}$, tile 0 (empty space) thru tile $i$ are in their goal positions and tile $i+1$ is not. $G_8$ is the set of all solvable problems where the empty space is not in its final position. In $G_7$, the space is in its final position, but tile 1 is not. In $G_6$, the space and tile 1 are in their final positions, and tile 2 is not, and so on. In $G_2$, the tiles 6,7, and 8 are out of place. In $G_1$ they all reach the goal positions simultaneously. The arrows represent "more difficult than" relation.

For example, in Rubik's Cube, the features are cubie names, and their values are cubie positions. In Eight Puzzle, the features are tiles, and their values are tile positions. Turning of each face of the Rubik's Cube is an operator; and so is moving each tile next to the space in Eight Puzzle. Thus, both in Rubik's Cube and Eight Puzzle, each operator reduces a problem to a single new problem, which represents its next state.

A domain is *serially decomposable* if there is a total ordering on the set of features $f_1$ thru $f_n$ such that 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). Both Eight Puzzle and Rubik's Cube are serially decomposable. In Rubik's Cube the effect of a turn on the position of a cubie is completely independent of the positions of other cubies. In Eight Puzzle, the effect of any operator like Up, Down, etc. on a tile depends not only on the position of that tile, but also on the position of the space. If we treat the space as a special tile, say tile 0, in Eight Puzzle, then it is serially decomposable for any ordering of features starting with the space as the first feature $f_1$.

We assume that the goal is satisfied by a single goal state $\langle g_1, \ldots, g_n \rangle$ known to the learner. A problem solving stage corresponds to the set of exercises where features 0 thru $i$ have their goal values. The ordering of the features is such that the domain is serially decomposable with respect to it. The stage $G_1$ corresponds to the singleton set $\{\langle g_1, \ldots, g_n \rangle\}$ that contains the goal state and is trivially solved. $G_2$ is the next more difficult level in which all but a small number of features have their goal values. All features simultaneously reach their goal values when an appropriate macro-operator is applied to the problems of this stage. For example, in Eight Puzzle, $G_2$ may be the set of problems in which all tiles except 6,7, and 8 are in their goal positions. The stages $G_3$, $G_4$, etc. are successively more difficult until $G_8$ which represents the set of all solvable problems where the space is out of place. The stages are mutually exclusive and collectively cover all solvable[3] problems (see Figure 3).

Without loss of generality assume that the domain is serially decomposable with respect to the feature ordering $f_1, \ldots, f_n$. Generalizing the above, we choose the problem solving stages so that all problems in stage $G_l$ have some features $F_l = \{f_1, \ldots, f_i\}$ at their goal values for some $i$ and feature $f_{i+1}$ has a non-goal value (if it exists). Since $G_1$ is the goal state, $F_1$ includes all features. Moreover for all $l$, $F_l \subset F_{l-1}$. Typically, $F_{l-1}$ has one more feature than $F_l$, which is stored in $\Omega(l)$. But sometimes as in $F_2$ and $F_1$ in Eight Puzzle, they might differ by more than one feature, all of which reach their goal values simultaneously when some operator sequence is applied. If that is the case, $\Omega(l)$ is some feature in $F_{l-1} - F_l$. By the property of serial decomposability, since all the features in $F_l$ have their goal values for all problems in stage $G_l$, the effect of any operator sequence on the feature $\Omega(l) = r$ in a problem $s$ only depends on that feature's value $v_j$ and nothing else. This makes it possible to store a single macro-operator $M_{j,l}$ for each value $v_j$, which takes all states that share this value for $\Omega(l)$ from stage $G_l$ to the next stage $G_{l-1}$. This repeats for each successive stage until the final stage $G_1$.

A domain satisfies *operator closure* if the set of solvable instances is closed under operators, i.e., every problem that can be reached from a solvable problem by an operator is solvable. Korf showed that if a domain is serially decomposable and satisfies operator closure, then it has a macro-table that can be used to solve all solvable problems (Korf, 1985). In our formulation, a macro-table consists of a set of macros for each stage $G_l$, one for each value of the feature $\Omega(l)$.

Figure 4) shows how to solve a problem using a macro-table. Recall that if the value of the feature $\Omega(l)$

[3]Only half the problems in this domain are solvable.

```
01 procedure Macro-problem-solver
02      input problem s;
03      let solution = " ";
04      for l := L thru 0 by −1 do
05              let Ω(l) = feature r;
06              let the value of feature r of problem s be v_j;
06              if v_j is different from the value of feature r in the goal state then
07                  solution := solution.M_{j,l};
08                  s := apply(M_{j,l}, s);
09      end for;
10      return (solution);
11 end Macro-problem-solver;
```

Figure 4: Macro Problem Solver

in state $s$ is $v_j$, then the $j^{th}$ macro for stage $G_l$, denoted by $M_{j,l}$, takes $s$ to a state where all features in $F_{l-1}$ are at their goal values. Note that the trajectories of the macro-operators for problems in $G_l$ might pass through higher level stages before eventually reaching $G_{l-1}$ or lower-level stages. The algorithm of Figure 4 first initializes the solution to empty sequence and the level $l$ to $L$. It retrieves the feature $r = \Omega(l)$ that corresponds to the current level $l$ (lines 03-05 of Figure 4) and its value $v_j$. If $v_j$ is different from that of the corresponding value for the goal state, then the macro $M_{j,l}$ is applied to $s$ and appended to the solution (lines 06-08). This is repeated until the goal is reached and the resulting solution is returned (line 10).

An example problem and solution in Eight Puzzle are shown in Figure 5. The letters `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 space and $i$ for tile $i$. The goal is arbitrary but fixed for each target problem solver. The space is ordered first in any serializable ordering, followed by tiles 1 through 8 in our example.

We now turn to learning of macro-tables. One of the difficulties here is that the ordering of the features in the macro-table is not known and must be discovered. All that the learner knows is that the domain is serially decomposable with respect to this ordering, and the exercises are processed by the generalizing routine in the increasing order of their difficulty. The MT-Generalize routine for the macro-table learning takes the difficulty level $l$, the set of features $F_{l-1}$ which corresponds to the known macro-table at this point, and a set $S$ of exercises as input (line 02 of Figure 6). It builds the $l^{th}$ column of the macro-table which has macro-operators that reduce problems of difficulty level $l$ to those of lower levels.

The algorithm first finds the set of features $Q_l$ for which all exercises in $S$ have the goal values (line 03). Now, the $l^{th}$ column of the macro-table must belong to some feature for which we do not yet have a column, i.e., a feature not in $F_{l-1}$. Since all exercises in $S$ have their goal values for features in $Q_l$, we can also exclude these as candidate features for the $l^{th}$ column of the macro table. Hence, the candidate features are selected from features $F$ not in $F_{l-1}$ or $Q_l$ (line 04). In general, there may be several features in $F$. Depending on the problem distribution and the domain, when one feature is taken to its goal value, several other features might also reach their goal values for all problems in the exercise set. In fact, as we noted earlier, this happens for the stage $G_2$ in Eight Puzzle. Hence all such features are potential candidates. Arbitrarily picking one from them may not work, because if we pick a wrong feature, it may not satisfy the serial decomposability property in that the effect of an operator on that feature might also depend on other candidate features. Hence, the algorithm searches for the correct feature from this candidate set, by verifying that the serial decomposability is obeyed for all examples in the candidate set if this feature is chosen.

In particular, for each candidate feature $r \in F$, the algorithm partitions the set $S$ into subsets $S_1, \ldots, S_k$ where the value of the feature $r$ is $v_1, \ldots, v_k$ respectively (line 08). For all problems in $S$, it searches for macro operators that take the values of the $r^{th}$ feature to their goal values while preserving all the feature

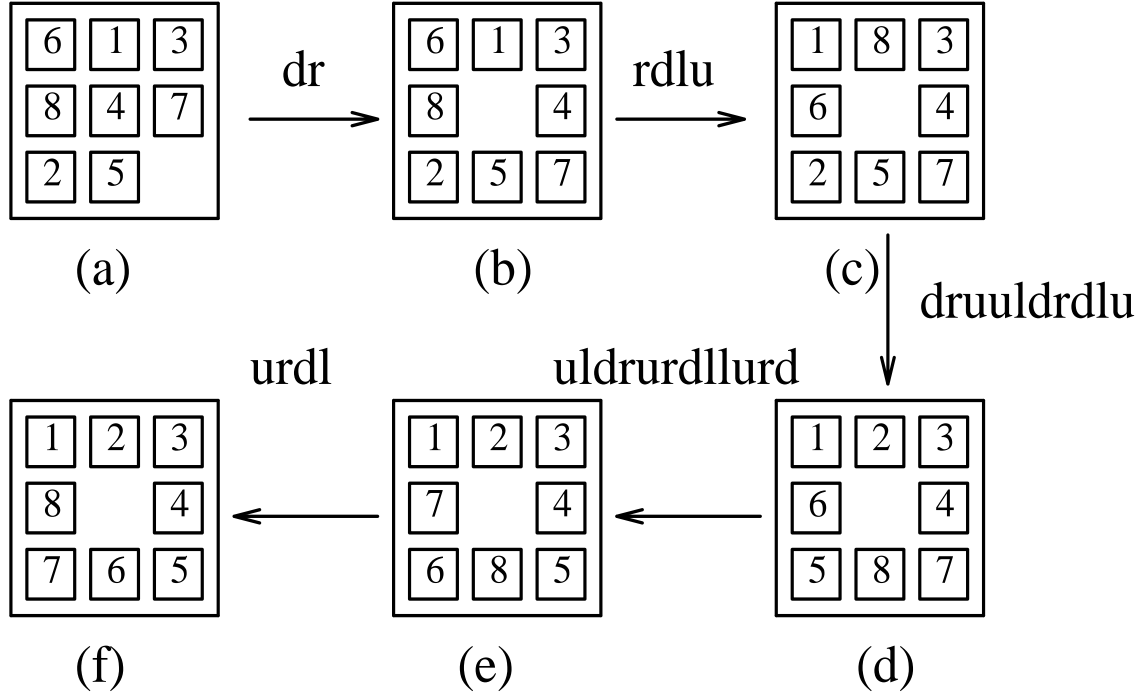


Figure 5: An Eight Puzzle problem and the intermediate subgoals. The macro-operators solve the successive subgoals. Problems (a), (b), (c), (d), (e), and (f) are respectively in $G_8$, $G_7$, $G_6$, $G_3$, $G_2$, and $G_1$.

values in $F_{l-1}$ (lines 09-10). We assume that there is a serially decomposable ordering of features for the domain, and the target problem solver is a macro-table which is ordered by one such ordering of features.

If all the features in $F_{l-1}$ have been correctly identified according to this ordering, then there is a serially decomposable ordering of features $F_{l-1} \bigcup \{r\}$ starting with some feature $r$ for the problems in $S$. Hence, there is a consistent macro-table for these problems where $r$ is a correct feature of column $l$. Since all exercises in $S_j$ have the same value for feature $r$, any operator sequence which reduces *any* exercise in $S_j$ to subproblems in the easier stages, can similarly reduce *all* exercises in $S_j$. If the above is not true for some feature $r$, it is rejected as a candidate feature for the $l^{th}$ column, and the other features are searched for until the correct feature is found (lines 11-15). The correct feature is stored in $\Omega(l)$ (line 16). All other features in $F$ that have also reached their goal values simultaneously along with $\Omega(l)$ are stored in $F_l$ (line 14). For each value $v_j$ of the feature $\Omega(l)$, the operator sequence that reduces the exercises in $S_j$ to simpler subproblems is stored in the macro table indexed by $(j, l)$ (line 10). Finally, the updated feature arrays $\Omega$ and $F$, and the macro-table are returned (line 18).

In order for the macro-table learning to be efficient, it must be possible to reduce an exercise of a given difficulty to a set of subproblems whose total difficulty is strictly less in polynomial time. In other words, the stage structure of the domain must be dense for the default problem solver, which in this case is called Macro-Search. Hence it is one of the conditions for the following main theorem of this section.

**Theorem 2** *Let $\mathcal{H}$ be a set of macro-tables for a collection of serially decomposable domains in a meta-domain $\mathcal{M}$. If the stage structure defined by the target macro-tables is* dense *for the Default Problem Solver (DPS) at the disposal of the learning algorithm, then there exists an algorithm for learning the problem solvers in $\mathcal{H}$ from exercises.*

**Proof:** We claim that ExerciseLearner where MT-Generalize is used for generalization in the place of its Solve-and-Generalize is a learning-from-exercises algorithm for $\mathcal{M}$ in $\mathcal{H}$.

First, from Figure 4, we observe that the Macro-problem-solver runs in time $O(ns)$ assuming that the

```
01 procedure MT-Generalize
02     input Level l, Features F_{l-1}, Exercise set S
03     Q_l := All features that have the goal values for all problems in S.
04     F := All features other than those in Q_l ⋃ F_{l-1}
05     for each feature r ∈ F until Success = true
06         F_l := F;
07         Success := true;
08         Partition S into S_1, ..., S_k based on the value of feature r
09         for j := v_1 thru v_k while Success = true
10             M_{j,l} := Macro-Search(x), for some x ∈ S_j
11             for each problem y in S_j
12                 if z := apply(M_{j,l}, y) is not in a solvable stage,
13                 then Success := false;
14                 else F_l := F_l ⋂ all features which have goal values in z;
15         end for;
16         If Success = true Ω(l) := r;
17     end for;
18     return Ω, F, M
19 end MT-Generalize;
```

Figure 6: The MT-Generalize routine that learns macro-operators. Macro-Search returns an operator sequence that reduces the problem $x$ into subproblems which can be solved by the current macro-table. It is stored as $M_{j,l}$ if it works for all exercises whose feature $\Omega(l)$ has a value $v_j$.

maximum length of a macro in the macro table is bounded by $s$. Hence $\mathcal{H}$ is a set of polynomial-time problem solvers.

To apply Theorem 1, we need to show that (a) MT-Generalize finds a set of macro-operators that correctly solve all exercises of stage $G_l$ whose reduction sets are solvable by $H_{l-1}$ in polynomial time, and (b) the log of the number of possible failure regions for the candidate macro-table columns for level $l$ is polynomial.

Let $E_l$ be the training exercises of stage $G_l$. Hence they should all have the same values for features that correspond to stages $G_{l+1}$ thru $G_L$. Let $Q_l$ be the set of features whose values are shared by all exercises in stage $G_l$ and the goal state. For each exercise, the algorithm first calls Macro-Search (the default problem solver) to find a macro-operator that takes it to the next stage. If $\Omega(l)$ is a key feature for stage $G_l$, then the same macro-operator $M_{j,l}$ should reduce all exercises that have the same value for this feature to exercises of stage $G_{l-1}$ or lower stages. Since there exists some correct feature for stage $G_l$, it can be found by considering all potential key features of stage $G_l$ and testing them.

We now bound the number of possible failure regions of the macros of stage $G_l$. Note that there are two different pieces of information learned in each stage: the feature $\Omega(l)$, and the corresponding macro-operators. The feature $\Omega(l)$ can be selected in at most $n$ ways. For each possible choice, the only way in which the macros of stage $G_l$ are going to be incorrect is if some macro-operators in that column have not been learned (because there was no corresponding exercise in the training set). Since the number of values of any feature is also bounded by $n$, there are at most $2^n$ subsets of macro-operators in the $l^{th}$ stage. Hence the number of the failure regions of bad problem solvers for this stage is upper bounded by $B_n = 2^n n$. Since $\ln B_n$ is bounded by $\ln n + n \ln 2$, the sample size $m$ is $\frac{L^2}{\epsilon}(\ln n + n \ln 2 + \ln \frac{L}{\delta})$. It is easy to see from Figure 6 that the run time of MT-Generalize is bounded by $O(n^2 m b^d)$, where $m$ is the number of exercises, $d$ is the length of longest macro-operator, and $b$ is the branching factor of search. Since the running time and sample size of MT-Generalize are both polynomials in all the parameters except $d$, and $d$ is independent of $n$ and $L$, all the conditions of Theorem 1 are satisfied, and it is a learning-from-exercises algorithm for $\mathcal{M}$ in $\mathcal{H}$. □

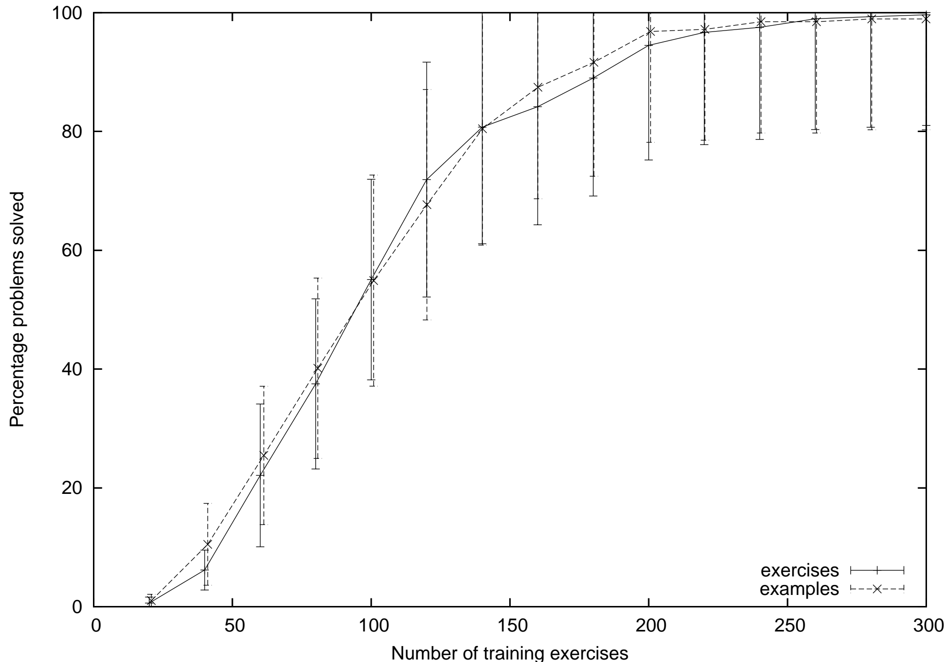


Figure 7: Learning curves for the macro-operator learning system from exercises and examples. The X-axis shows the number of exercises used in training. When learning from examples, the number of exercises involved in the solutions are counted. The Y-axis shows the average accuracy on the test distribution. Each point is an average of 30 different learning trials. The error bars are 1 standard deviation away on either side.

## 4.2 Experimental Results

In this section, we present empirical results on the performance of an implementation of the algorithms of previous section, in the domain of Eight Puzzle (Korf, 1985; Tadepalli & Natarajan, 1996).

We used the IDA* algorithm with the city-block distance heuristic as the default problem solver. Note that the domain satisfies our density requirement in that it is possible to move from any problem instance to an easier problem instance with bounded search by our default problem solver. This can be generalized to the $N \times N$ version of Eight Puzzle and the corresponding stage structure. As was illustrated by (Finkelstein & Markovitch, 1998), the city-block distance heuristic coupled with the IDA* algorithm limits the search by first bringing the tile in question to a nearest location to its goal position without disturbing previously arranged tiles, and then doing a bounded search to place that tile and any previously arranged tiles in their goal positions.

Substituting $L = 7, n = 9$, and $\epsilon = \delta = 0.1$, in the formula $\frac{L^2}{\epsilon}(\ln n + n \ln 2 + \ln \frac{L}{\delta})$ for the sample size of the learning algorithm, we get 6,215 exercises for Eight Puzzle! Obviously, this is too high in practice and indicates the worst case nature of the analysis. To get a more realistic bound, we turn to experiments.

Our program was trained on solvable exercises that belong to the different stages such as those shown in Figure 5. The exercises were selected uniformly randomly from each stage, making sure, using a previously learned macro-table, that they are solvable. For example, the exercises in stage $G_1$ have the tiles 0 thru 5 in their goal positions, and the rest of the tiles distributed according to one of the 2 solvable (and not solved) configurations.

The system was trained from 0 to 300 random exercises in increments of 20, and was tested on an independent test sample of 100 random test examples after each increment. The results in Figure 7 are averages over 30 different training sets. The error bars denote 1 standard deviation intervals on both sides. Learning reaches a 89% average accuracy with 180 exercises and 94.5% average accuracy with 200 exercises.

This is more than an order of magnitude faster than what is predicted from the worst case theoretical analysis. For comparison, we show the results of learning from examples where each example consists of a problem chosen using a uniform distribution and a solution derived from a teacher's macro table. In this case, we also assume that the subgoal order of the teacher's macro table is known. The learner simply parses the solution into macro-operators according to the subgoals in the teacher's macro table and stores them (Tadepalli & Natarajan, 1996). Thus each example is equivalent to multiple exercises, one for each subgoal in its solution. Hence to make the comparision to learning from exercises fair, on the X-axis, we show the number of subgoals in the solutions of a given number of training examples, averaged over 30 trials. The Y-axis shows the accuracy on the randomly chosen test examples. It is clear that there is no significant difference between the two plots showing that learning from exercises is as effective as learning from teacher-given solutions in this case.

The S-shape of the two learning curves is worth noting and is characteristic of problems which need multiple independent heuristics (e.g., macro-operators) to solve them. In the beginning, the learning is slow because solving a new test problem requires knowing appropriate macro-operators for each stage. With only a limited amount of training, it is likely that one or more of these macro-operators are missing, which means that the test problems cannot be solved. But as more exercises are solved at each stage, it becomes more likely to have learned the appropriate macros for each stage, which leads to a sharp increase in the performance.

# 5 Learning control rules

In this section, we examine building efficient problem solvers by learning control rules (Mitchell et al., 1983; Langley, 1983; Minton, 1990). Control rules reduce search by selecting, rejecting or ordering operators appropriately.

## 5.1 Selection rule learning

We consider a simple problem solving architecture based on selection rules. A selection rule is a pair $\langle U(o), o\rangle$, where $U(o)$ describes the set of problem states on which this rule selects the problem reduction operator $o$. $U(o)$ is called the *select-set* of $o$.

We assume that the select-sets of operators of the domain are described in some language $\mathcal{L}$. We consider a hypothesis space $\mathcal{F}_{\mathcal{L}}$ of problem solvers, where every problem solver consists of a set of select-sets in $\mathcal{L}$, one for each operator in the domain. Let $L_n$ be the select-sets restricted to problems of size $\leq n$. We say that the control rules are *non-overlapping*, if the select-sets of different operators are mutually exclusive, so that every state is in the select set of exactly one operator.

Figure 8 shows the problem solver that works by using control rules. Given a problem and a set of control rules, it picks the operator $o_i$ whose select-set contains the problem if any such exists and applies it, getting a new set of problems $S_x$ (lines 06-07 of Figure 8). The problem solver is recursively applied to each of the problems in $S_x$, and their solutions are composed with $o_i$ and returned (line 09).

There is a natural partial ordering over the sentences in $\mathcal{L}$ defined by the "more general than" relation. A sentence is *more general than* another if the set represented by the first sentence is a superset of that represented by the second sentence. The idea of our generalization algorithm is based on learning with one-sided error, which means that the various select-sets we learn may be over-specific compared to the target sets, but never over-general (Natarajan, 1991). For a given target problem solver $f \in \mathcal{F}_{\mathcal{L}}$, define $H_f$ to be the set of problem-solvers in $\mathcal{F}_{\mathcal{L}}$, whose select sets are more specific than those of $f$. These are all the problem solvers that our learning algorithm can potentially learn from the exercises of $f$. To eliminate the risk of over-generality, the algorithm computes "least general generalizations," defined as follows.

**Definition 3** *The* least general generalization *(lgg) of a set $S$ of instances is a select-set $G$ in $\mathcal{L}$ which is least general among all select-sets in $\mathcal{L}$ that represent the supersets of $S$.*

Since our algorithm learns from self-generated solutions rather than the user-given solutions, to guarantee no over-general rules, we have to make sure that the reduction set generated by the system is the same as

```
01  procedure Control-rule-problem-solver
02          input x;
03          if x is solved by the goal procedure G
04            return G(x)
05          else begin
06                      pick the operator o_i s.t. x ∈ U(o_i);
07                      if no such o_i exists, halt with a failure;
08                      S_x := o_i(x);
09                      return COMPOSE (o_i, ⋃_{y∈S_x} Control-rule-Problem-solver (y));
10          end;
11  end Control-rule-problem-solver
```

Figure 8: A problem solver that uses control rules

```
01 procedure Learn-Select-Rules
02       input Stage l, Exercise set S, Current select sets U(o_1), ..., U(o_k)
03       for each exercise e ∈ S
04             Let the default problem solver reduce e to an exercise set R(e), where
05                 all problems in R(e) can be solved using current select sets
06             for each ancestor node e_j of the nodes in R(e) in the solution tree of e
07                 if o_i is the operator applied to e_j
08                     U(o_i) = Generalize(U(o_i), e_j);
09       end for;
10       return U(o_1), ..., U(o_k)
11 end Learn-Select-Rules;
```

Figure 9: The Learn-Select-Rules routine that learns selection rules. The default problem solver reduces the exercises to easier ones. The Generalize routine returns the least general generalization ($lgg$) of the input sets.

that generated by the target problem solver. To ensure this, we assume that the exercises are *uniquely solvable* in that they can only be reduced in one way.

**Definition 4** *Let $e$ be an exercise of stage $l$ with respect to a target problem solver $f$. We say that $e$ is* uniquely solvable *by the default problem solver with respect to $f$, if $R_f(e)$ is the only set of problems that $e$ can be reduced to by DPS so that they can be solved by $f$ in fewer than a total of $l$ steps.*

The learning algorithm Learn-Select-Rules in Figure 9 works as follows. Using the default problem solver, it first tries to reduce each exercise $e$ to a set $R(e)$, where each problem in $R(e)$ can be solved using the current problem solver (lines 04-05). Note that, in general, several primitive reduction operators may be needed to reduce $e$ to its reduction set. Thus, the final solution tree of $e$ might consist of several ancestors of the nodes in $R(e)$, in addition to $e$ itself. For one such ancestor $e_j$, let $o_i$ be the corresponding operator that has been applied in the solution tree. The routine Generalize incrementally generalizes the current select set $U(o_i)$ of the operator $o_i$ to include $e_j$ (lines 06-08). To avoid the risk of over-generalization, we employ the least general generalization of the select sets. This is repeated for all ancestors of nodes in $R(e)$. After processing all exercises, the final select sets are returned (line 10). We are now ready to state our theorem.

**Theorem 3** *Let $\mathcal{F}_{\mathcal{L}}$ be the set of problem solvers for a meta-domain $M$, based on polynomially computable, non-overlapping select sets from the language $\mathcal{L}$. Then there exists an algorithm for learning $\mathcal{F}_{\mathcal{L}}$ from exercises, if*

1. *the stage structures defined by the target problem solvers are* dense *for the default problem solver (DPS) at the disposal of the learning algorithm,*
2. *the exercises are guaranteed to be* uniquely solvable,
3. *for any exercise $e$ and select-set $U(o_i)$,* Generalize *computes the least general generalization of $e$ and $U(o_i)$ in $\mathcal{L}_n$ in polynomial time; and*
4. $\ln|\mathcal{L}_n|$ *is a polynomial in $n$.*

**Proof:** We show that ExerciseLearner where Learn-Select-Rules is used in the place of Solve-and-Generalize is a learning-from-exercises algorithm for $\mathcal{F}_{\mathcal{L}}$.

Unlike in the macro-tables example, where there is a set of macros for each stage, here the learning algorithm builds a set of selection rules that work for all stages. The key idea in the proof is as follows: Given a set of exercises $S$, the learning algorithm Learn-Select-Rules in Figure 9 will use the default problem solver (DPS) to produce a set of reduced problems that can be solved by the problem solver learned so far. This process generates some new examples, which are used to generalize the select-sets of the current problem solver in a least general way. If the conditions of the theorem are satisfied, this can be carried out in polynomial time. With these select-sets in place, the algorithm Solve of Figure 8 behaves as an approximate problem solver for the domain $D$. The rest of the proof deals with the details.

The ExerciseLearner calls DPS on each exercise. By the assumption that the exercises are uniquely solvable, DPS's solution would have the same reduction set as the target problem solver at the top level. By the assumption of the density of the stage structure, DPS will be successful in reducing the exercises to their reduction sets with respect to the target problem solver if the problems in the reduction set are solvable by the current problem solver.

Since Generalize outputs a *least general generalization* of the current hypothesis and the new examples, its select sets are always subsets of those of the target problem solver provided the exercises generated by Learn-Select-Rules (see Figure 9) are also generated by the target problem solver. Since the target's select sets are non-overlapping, so are the hypothesized select sets. Thus, any solution produced by the hypothesis problem solver is also produced by the target problem solver, and hence the exercises generated by the system at any level are in fact those that will be generated by a target problem solver.[4]

We now show that the sample size in our algorithm is sufficient for learning. For problems of size $n$ or less, each set $U(o)$ can be chosen in at most $|\mathcal{L}_n|$ ways. Since there are $k$ operators, the number of distinct select-set tuples, and hence the number of distinct problem solvers is at most $|\mathcal{L}_n|^k$. Since the number of failure regions is upper bounded by the number of problem solvers in the hypothesis space $\ln B_n \leq \ln|\mathcal{L}_n|^k = k\ln|\mathcal{L}_n|$. Since $\ln|\mathcal{L}_n|$ in polynomial in $n$, a polynomial-size sample is sufficient for learning.

Since the membership in the sets in $\mathcal{L}_n$, and the least general generalizations of the sets of examples can both be computed in polynomial time by the assumptions of the theorem, Control-Rule-Learner runs in polynomial time as well. Hence, $\mathcal{F}_{\mathcal{L}}$ is learnable from exercises. □

## 5.2 Application to symbolic integration

In this section, we describe an application of our theory to learning control rules for the domain of symbolic integration, which also effectively exploits our problem reduction framework. This domain has been studied extensively in previous work (Mitchell et al., 1986; Tadepalli & Natarajan, 1996). The purpose of this simple illustration is to demonstrate the practicality of our theory as well as to put the earlier experimental work on a firmer theoretical footing.

Consider the class of symbolic integrals that can be solved by the standard integration operators, some of which are shown in Figure 1. These can be naturally described as problem-reduction operators. The

[4] A more rigorous proof can be given using mathematical induction on the difficulty level of the exercise.

$Prob \rightarrow \int Exp \ d \ Var$
$Exp \rightarrow Term | Term + Exp | Term - Exp$
$Term \rightarrow P\text{-}term \mid P\text{-}term * Term \mid P\text{-}term \ / Term$
$P\text{-}term \rightarrow Const | Var | (-Term) | Trig | Power | Prob | (Exp)$
$Power \rightarrow (Var \uparrow Term)$
$Trig \rightarrow (\sin Var) | (\cos Var)$
$Const \rightarrow Int | a | k$
$Var \rightarrow x$
$Int \rightarrow 0|1|2|3|4|5|6|7|8|9$

Figure 10: A partial grammar to generate the integration problems. Figure reproduced from (Tadepalli and Natarajan, 1996).

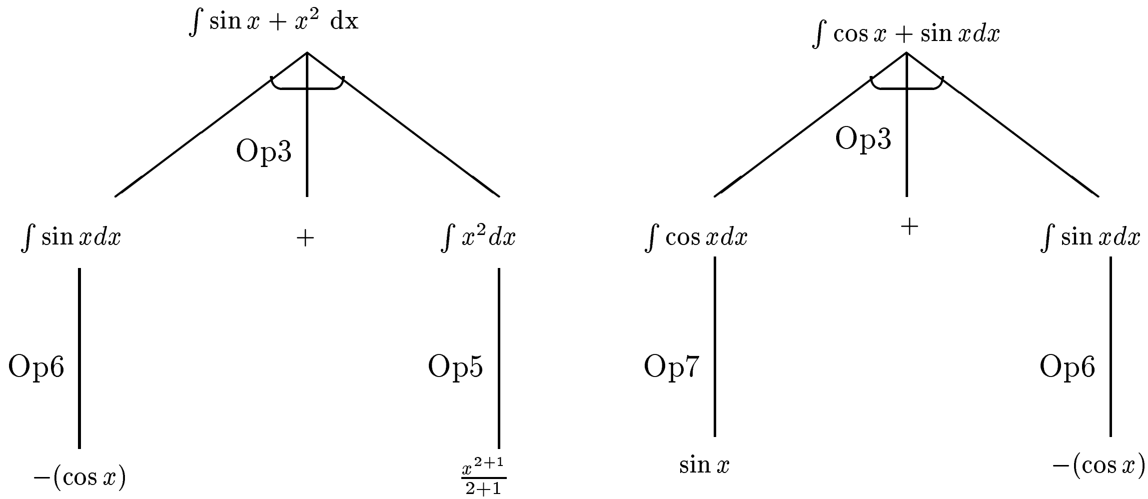


Figure 11: Tree representations of the solutions of the two examples. Figure reproduced from (Tadepalli and Natarajan, 1996).

problems of this domain can be described by an unambiguous context free grammar $\Gamma$ such as shown in Figure 10.

Let $\alpha$ be any sentential form, i.e., a string of terminals and variables, of the grammar $\Gamma$ derivable from the start symbol *Prob*. A sentential form $\alpha$ represents the set of problems derivable from $\alpha$ using the productions of $\Gamma$. Consider a hypothesis space $\mathcal{F}$ of problem solvers whose select-sets are represented by the sentential forms of the grammar $\Gamma$. We argue that the ExerciseLearner, with an appropriate Generalize routine that computes the least general generalization of a set of problems, is a learning algorithm for $\mathcal{M}$ in $\mathcal{F}$.

We define the difficulty of an exercise to be the number of nodes generated when solving that exercise using the target problem solver. Note that this guarantees that the difficulty of an exercise is strictly greater than the sum of the difficulties of the exercises in its reduction set. To reduce an exercise to its less difficult stages, it only needs to apply a single, correct problem reduction operator. Problems are solved by a series of reductions. Figure 11 shows the solutions of two such problems: $\int \sin x + x^2 dx$ and $\int \cos x + \sin x dx$.

**Theorem 4** *Let $\mathcal{L}$ be the set of sentential forms derivable from the start symbol of an unambiguous context free grammar $\Gamma$. Let $\mathcal{F}_{\mathcal{L}}$ be the hypothesis space of problem solvers defined using non-overlapping control rules using the select-sets from $\mathcal{L}$. Then there exists an algorithm that learns $\mathcal{F}_{\mathcal{L}}$ from exercises provided the exercises are uniquely solvable by the default problem solver.*

**Proof:** First, we note that there exist several polynomial-time algorithms to test the membership in the context-free precondition language, e.g., see (Earley, 1970). Since different stages are separated by only a single problem reduction operator, a simple default problem solver that tries every possible reduction operator exactly once on the exercise would be able to find an appropriate reduction set and satisfy the first condition of Theorem 3.

The second condition, namely unique solvability of the exercises, is assumed in the statement of the theorem.

We have to show that the last two conditions of Theorem 3 hold for $\mathcal{L}$. Since the proof is the same as that of (Tadepalli & Natarajan, 1996), we will be brief. A sentential form $\alpha$ is more general than $\beta$ if $\beta$ can be derived from $\alpha$. This is because any expression that is in the set represented by $\beta$, i.e., can be derived from $\beta$, can also be derived from $\alpha$ and hence is in the set represented by $\alpha$. Since all sentential forms are derived from the start symbol, given any two sentential forms, there always is a sentential form that is more general than both. To construct least general generalization of two sentential forms, we march down both derivation trees simultaneously starting from the two start symbols, and including all the children of the two corresponding nodes when they are the same, i.e., the same production is used at both parent nodes. If the sets of children of a pair of nodes differ, the procedure terminates with the parent and does not include the children. The yield of the resulting tree gives the least general generalization of the two inputs. An example is given in Figure 12. Generalize computes the $lgg$ of more than 2 examples incrementally by repeatedly finding the $lgg$ of the current result and each new example. Since the grammar $\gamma$ is unambiguous, the result at each stage is unique. It is easily seen to be computable in time polynomial in the sizes of all the trees.

Since the size of $|\mathcal{L}_n|$ is exponential in $n$, its log is polynomial, satisfying the last condition. Hence $\mathcal{F}_{\mathcal{L}}$ is learnable from exercises by ExerciseLearner. □

## 5.3 Experimental Results

The ExerciseLearner and the Solve routine are implemented and tested in the integration domain introduced in (Tadepalli & Natarajan, 1996). The expressions to be solved are generated using the grammar in Figure 10. The domain has 39 operators including those in Figure 1, and some differentiation and simplification operators. Solving a problem consists of removing the integral sign and simplifying the result as much as possible. The exercises are drawn from the intermediate nodes in the solutions generated by a program that used hand-coded select-sets. The context-free grammar and the control rules of the hand-coded problem solver are uniquely solvable. The difficulty of the exercise is the number of nodes in its solution.

The "natural" distribution of integration problems consisted of sums of products of powers of $x$ and some trigonometric functions of $x$. In particular, each "natural" problem is of the following form and is selected uniformly randomly: $\int\{0-9\}x^{\{3-9\}} + \{\sin x, \cos x, 0-9\} * x^2 + \{\sin x, \cos x, 0-9\} * x + \{\sin x, \cos x, 0-9\} dx$.

At each point, the system was tested on a set of 100 test problems. Figure 13 shows the percentage of the test problems correctly solved from the test set averaged over 30 training trials plotted against the number of training exercises. The test problems are selected from the same training distribution mentioned before. A test problem was counted as correctly solved by the learner if it is reduced to an expression without an integral sign.

The error bars denote one standard deviation intervals on both sides of the mean. The learning converges, reaching an average accuracy of 91.6% with 531 exercises and an average accuracy of 99.3% with 944 exercises. For comparision, we show the learning curve for learning from examples obtained by repeating the experiments in (Tadepalli & Natarajan, 1996). The X-axis shows the total number of stages involved in the solutions of a given number of training examples, averaged over 30 trials. The Y-axis shows the percentage of the test problems correctly solved at that point. Unlike in Eight Puzzle, in this domain, learning from examples is slightly slower than learning from examples, although the difference is not statistically significant. For example, it reaches an average accuracy of only 79.6% after learning from 534 exercises and an average accuracy of 97.9% after 944 exercises. The slightly lower performance with learning from examples is due to the fact that different subproblems generated from the same example are correlated and lead to under-generalization compared to learning from exercises generated independently. For example, all subproblems generated from the solution of the same problem typically involve the same multiplicative constants. Due

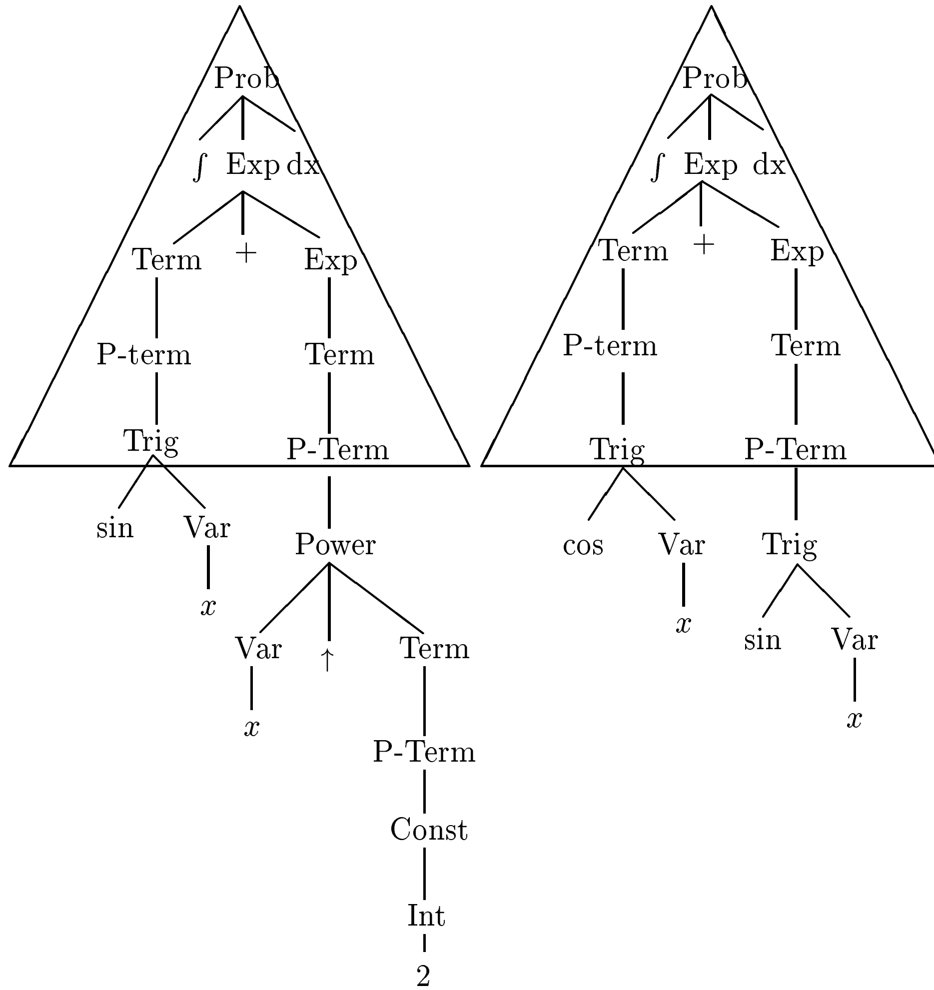


Figure 12: The parse trees of the two problems $\int \sin x + x^2 dx$ and $\int \cos x + \sin x dx$. The yield of the largest common super-tree, $\int Trig{+}\ P\text{-}term\ dx$, is their unique least general generalization. Figure reproduced from (Tadepalli and Natarajan, 1996).

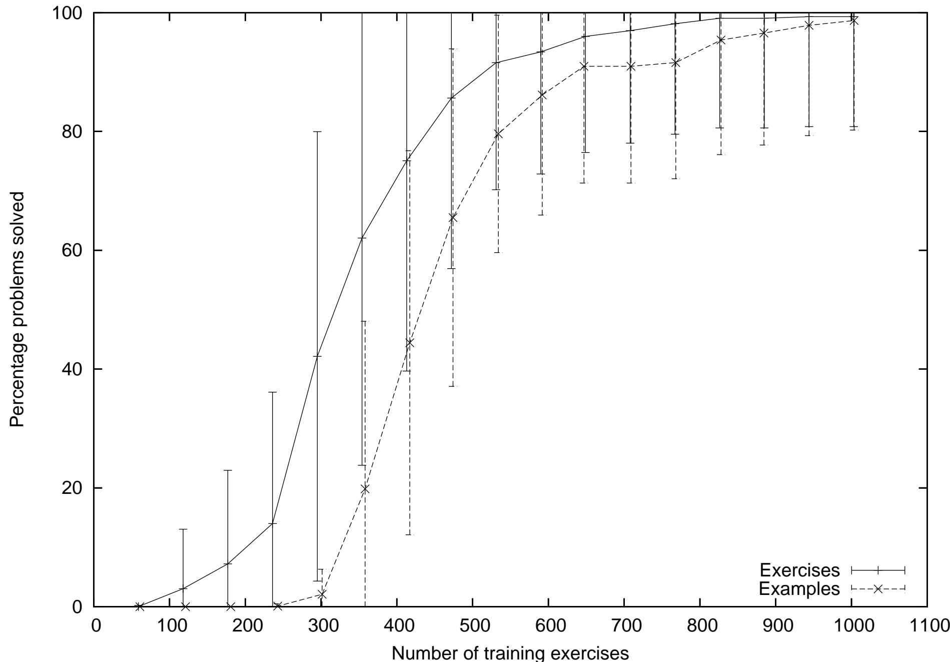


Figure 13: Learning curves for learning from exercises and learning from examples in the symbolic integration domain. In each case, the X-axis shows the number of training exercises in the training set. The Y-axis shows the average accuracy on the test distribution. Each point is an average of 30 different learning trials. The error bars are 1 standard deviation away on either side.

to the least general generalization technique employed by the learning algorithm, these constants do not get generalized. Since the exercises generated by the exercise learner are mutually independent, they typically have exercises with different multiplicative constants, which yield more general rules.

# 6 Learning Decision List Policies

We now illustrate our framework for a different form of control knowledge, namely a decision list. Here, we consider domains where each operator transforms the current problem to a single subproblem. Since this is equivalent to state-space formulation, we also refer to problems as states. We refer to the maps from states to operators as policies.

A decision list policy consists of a sequence of clauses of the form "if ⟨Test⟩ then apply ⟨Op⟩," where ⟨Test⟩ consists of a conjunction of $k$ literals. In any state, the first clause in the sequence whose ⟨Test⟩ evaluates to true "fires" and the corresponding operator ⟨Op⟩ is applied. A greedy decision list learning algorithm is devised by Rivest (Rivest, 1987), and is adapted by Khardon for learning action strategies (Khardon, 1996).

The algorithm DecisionList-learner of Figure 14 works as follows. First, all exercises of a given difficulty are reduced using the default problem solver to problems of lower levels of difficulty. In our domain, this reduction is achieved by searching for a sequence of operators that takes the original state to a single state of less difficulty (line 05). All states encountered during this sequence and the corresponding operators applied are collected as a set of one-step examples (lines 06-07). If a state is already in the set, that state is not added to the set to avoid conflicting operators on that state. The set of such one-step examples is generalized into a separate decision list policy for each stage using Rivest's algorithm (Rivest, 1987). The generalization algorithm uses a "separate and conquer" strategy, and works as shown in Figure 14. Given a set of one-step examples $S$ that consist of problem states $x$ and corresponding operators $y$, it searches for a clause $C =$ "if

```
01 procedure DecisionList-learner
02  input Level l, Exercise set E of level l;
03  Initialize decision list DL and set S to be empty;
04  for all problems x ∈ E
05          Use DPS to find a sequence of operators that reduces x to an easier level
06               Let x_1, a_1, ... x_{n-1}, a_{n-1}, x_n be the state-operator sequence where x_n is in a level < l.
07               Add all (x_i, a_i) to S, where 1 ≤ i < n and x_i is not already in S.
08  while S is non-empty do
09          Find a clause C = "if Test then apply Op," such that
10                ∀(x, a) ∈ S if Test(x) is true, then a is the same as Op
11          Append C to the decision list DL and remove all examples which satisfy Test from S
12  end;
13  return the decision list DL;
14 end DecisionList-learner
```

Figure 14: The decision-list policy learner

⟨Test⟩ then ⟨Op⟩," which is consistent with that pool of examples (lines 09-10). A clause is considered to be consistent with an example $(x, y)$, if whenever the ⟨Test⟩ is true for its problem state $x$, $y$ is the same as the corresponding operator. After a clause $C$ is chosen, it is appended to the decision list, and the examples which satisfy the ⟨Test⟩ are removed from the pool of examples (line 11). Then the loop repeats by choosing the next clause.

**Theorem 5** *Let $\mathcal{H}$ consist of a set of stage-specific decision list policies, each of which has at most $k$ tests in its clauses. Suppose that all policies that solve the domain are generated by some policy in $\mathcal{H}$. If the stage structure defined by the target decision list in $\mathcal{H}$ is* dense *for the Default Problem Solver (DPS), then there is an algorithm for learning $\mathcal{H}$ from exercises.*

**Proof:** We claim that Exercise-Learner with Solve-and-Generalize implemented by the DecisionList-Learner of Figure 14 learns the problem solvers in $\mathcal{H}$ from exercises.

By the density assumption, DPS will be able to solve each of the exercises of stage $l$ whose reduction sets are solvable by $\mathcal{H}_{l-1}$. Since we assumed that all policies are generated by some decision-list policy of the corresponding stage, the one-step examples generated by the DecisionList-Learner are consistent with some decision-list policy of stage $l$.

We can now use Rivest's result (Rivest, 1987) that showed that the above greedy generalization algorithm learns a decision list policy consistent with the training set, satisfying the first condition of Theorem 1.

The number of failure regions $B_n$ is bounded by the number of distinct decision list policies $|\mathcal{H}| \leq |O|^{n^k} n^k!$, where $n$ is the number of instantiations of each test, and $O$ is the set of operators. Hence $\ln B_n \leq n^k \ln |O| k n^k \ln n$, satisfying the second condition of Theorem 1 and proving that Exercise-Learner is going to be successful. □

We use a simulated robot domain for learning decision-list policies. The robot planning domain is depicted in Figure 15. The goal of the robot is to pick up the key in one of the rooms (R2) and use it to open the chest in room R4.

We divide the problem instances of this domain into 9 stages and learn a decision list policy for each stage. As usual, the stage $G_1$ contains all the states in which the robot has opened the chest, i.e., all goal states. The other stages are based on the location (room) of the robot and whether the robot has picked up the key, and are shown in Figure 16. In particular, in stages $G_2$ thru $G_5$, the robot has the key and in stages $G_6$ thru $G_9$, it does not have the key. The "more difficult than" relation between the stages follows the goal-subgoal structure of the problem to be solved. Since the robot must have the key before opening the chest, all the stages in which the robot does not have the key are more difficult than the stages in which

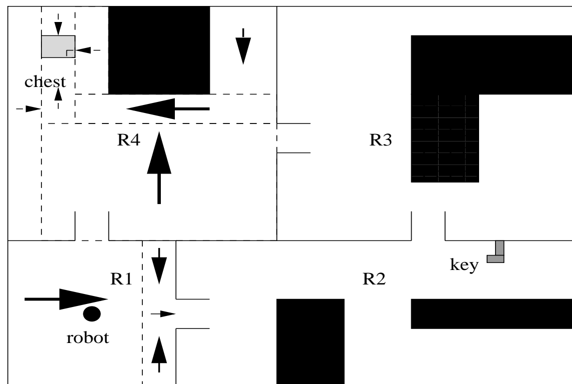


Figure 15: A simulated robot in a grid world of size $17 \times 13$ divided into 4 rooms. The blackened part of the rooms are inaccessible. The goal is to pick up the key in room R2 and open the chest in room R4. The robot is in R1. The dashed lines in R1 and R4 indicate the regions in which the robot has the same best action, as indicated by the arrows.

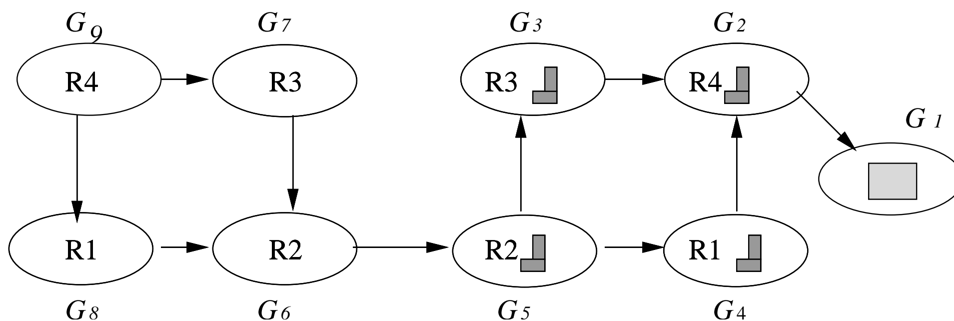


Figure 16: The problem space of the robot domain, divided into 9 regions. The final stage $G_1$ is on the right where the robot has opened the chest. $G_2$ thru $G_5$ correspond to problems where the robot has the key, and the rest correspond to problems where the robot does not have the key. The robot's location is indicated inside the ovals. The arrows correspond to "more difficult than" relationship.

the robot has the key. When the robot has the key, the goal is to be in room R4, which makes it the least difficult among all stages where the box is closed ($G_2$). The other stages are related to this stage by "more difficult than" based on their access relationships between the corresponding rooms. For example, since there is a door between R3 and R4 and not between R2 and R4, $G_3$ (which corresponds to R3) is directly related to $G_2$ (which corresponds to R4) by "more difficult than," and $G_5$ (which corresponds to R2) is not directly related to $G_2$. Similar relationships hold between goals $G_9$-$G_8$-$G_7$-$G_6$, $G_6$ being the least difficult since it corresponds to room R2 which has the key.

In particular, the tests of the clauses of the decision lists constrain the location of the robot to a rectangular region and whether or not it has the key. The then-part specifies a single operator. Notice that every possible policy is expressible by a decision list, since we can describe each possible state by the location of the robot, which is a single grid cell (and a trivial rectangle), and whether or not it has the key. Hence this domain satisfies the first assumption we made in Theorem 5. The density assumption is satisfied, if DPS is allowed to search as deep as necessary to move from one room to another.

In each iteration of the decision list learning algorithm, a consistent clause is found by merging all one-step examples that have the same operator into the most specific rectangular region that contains all their locations, and then specializing the clause until it does not cover any examples with different operators. The resulting clause is appended to the decision list and the examples that are covered by it are removed. The algorithm terminates when all examples are covered by some clause.

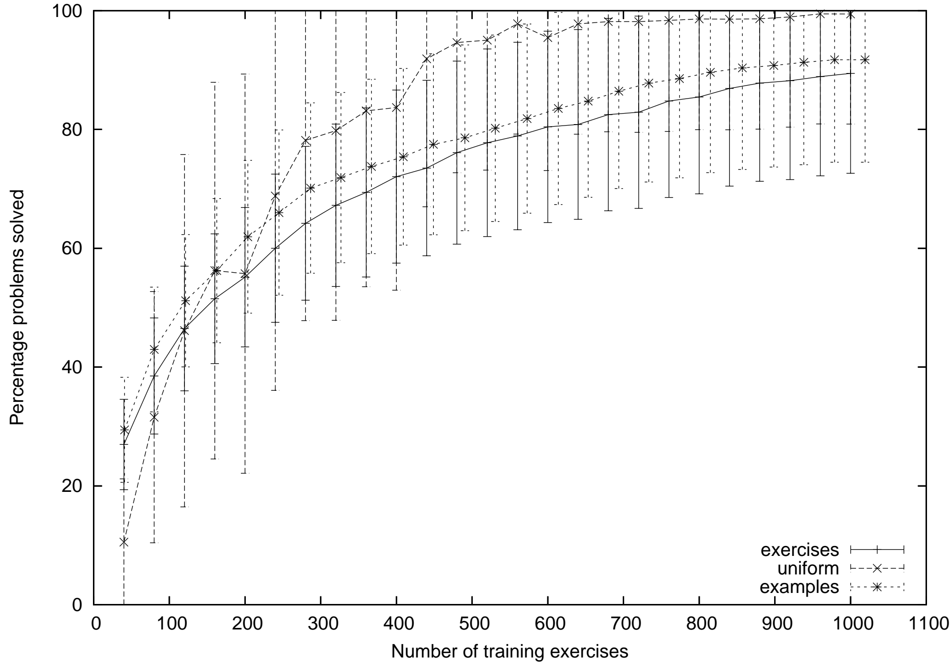


Figure 17: Learning curves for the robot planning domain where the test problems are chosen uniformly at random over all stages. The X-axis shows the number of exercises involved in training and the Y-axis is the average accuracy on the test set. The curve labeled "exercises" is generated by our learning algorithm. In the case of "examples," all exercises in the teacher-given solution are used and counted. In the case of "uniform," the exercises are chosen uniformly randomly over all stages. Each point is an average of 30 different learning trials. The error bars are 1 standard deviation away on either side.

The dotted lines in Figure 15 specify the regions that correspond to different clauses and the arrows indicate the operators suggested in those regions. Only the rules that correspond to stage $G_2$ (where the robot is in room R4 and has the key) and stage $G_8$ (where the robot is in room R1 and does not have the key) are depicted in the figure. For example, the big up-arrow in R4 says, "if you are between (1,5) and (7,8) and have the key, go up."

In this domain, when an example is reduced from one stage to the next, it has a small number of entry points that correspond to "doorways" between the rooms. Thus, when we generate exercises, the locations that correspond to doorways should have a higher probability. To generate the exercises properly, we first generate a problem instance $x$ using the "natural" distribution, which is uniform over all stages. We then solve the problem $x$ using a domain-specific problem solver, and collect the set of exercises that occur in its solution, where each exercise is the first problem in the sequence of subproblems of $x$ with a given difficulty. We then uniformly sample from this set of exercises generated from $x$.

We compare the performance to an example-based approach as before. That is, we choose problems using the natural distribution, and for each problem, choose the first exercise of *each* stage that occurs in its solution. We use all of them for training.

One difficulty with our approach is that generating exercises from the exact exercise distribution requires having an oracle that can solve natural problems. Unfortunately, this may not always be available. To show that our general algorithmic schema works even when the exercises are chosen from a different distribution, we did another experiment. We chose exercises using a uniform exercise distribution, i.e., used the natural distribution directly to choose exercises. In this case, since the natural distribution itself has problems of all levels of difficulty, we conjectured that it might perform just as well as the true exercise distribution.

In the learning curves in Figure 17 the X-axis shows the number of exercises used in training in all 3

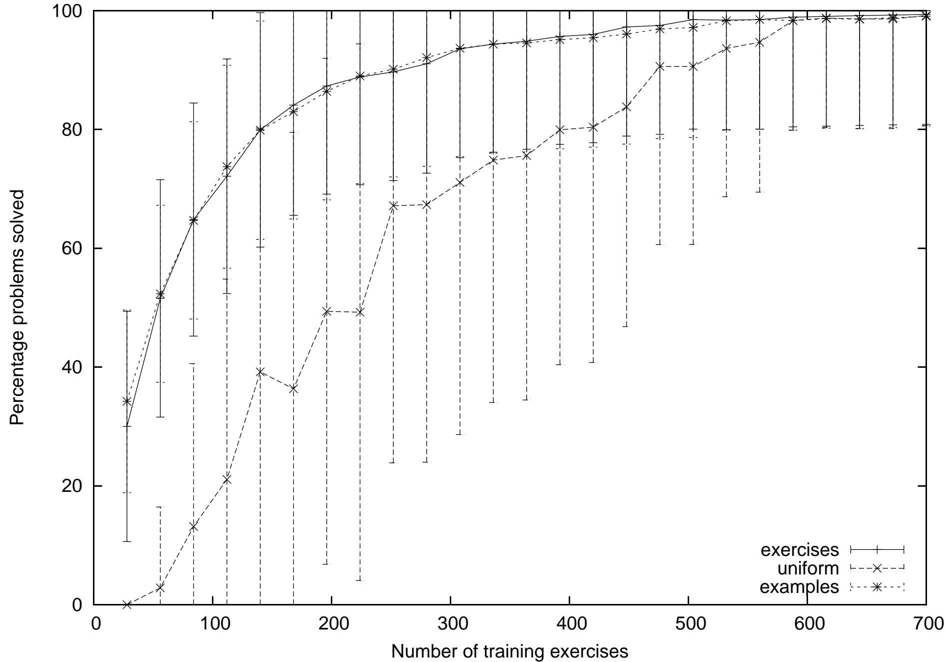


Figure 18: Learning curves for the robot planning domain where the test problems are chosen uniformly from the most difficult stage G9, where the robot is in room R4 and does not have the key. The X-axis shows the number of exercises involved in training and the Y-axis is the average accuracy on the test set. The curve labeled "exercises" is generated by our learning algorithm. In the case of "examples," all exercises in the teacher-given solution are used and counted. In the case of "uniform," the exercises are chosen uniformly randomly over all stages. Each point is an average of 30 different learning trials. The error bars are 1 standard deviation away on either side.

cases, and the Y-axis shows the performance on test problems chosen using the natural distribution. The results show that the performances of exercises and examples are similar, with a slight, but insignificant, edge for learning from examples. For example, the average accuracy for learning from exercises is 89.4% after 1,000 exercises. It is 91.7% for learning from examples after the same number of exercises. This compares quite favorably with the theoretical bound of 61,965, which is derived from the formula $\frac{L^2}{\epsilon}\left(B_n + \ln \frac{L}{\delta}\right)$ in Figure 2 by substituting $\epsilon = \delta = 0.1$ and $L = 9$. $B_n = 72$, which is the number of cells in the largest room and upper bounds the number of rectangular regions necessary to describe any policy in a single stage.

On the other hand, the performance for learning from the exact exercise distribution is much worse compared to learning based on uniformly random exercises. For example, an average accuracy of 91.9% was obtained with only 440 uniformly random exercises with average accuracy reaching 99.4% with 1,000 examples. This is because, the uniform exercise distribution has a wider range that is identical to the natural distribution. In contrast, in the exercise distribution derived from the solutions of the natural problems, most of the exercises belonged to the doorway locations and are highly redundant.

To test the effect of choosing exercises from an inexact exercise distribution further, we generated a new natural distribution in which all test problems belonged to the most difficult stage G9, where the robot is in room R4 and does not have the key. The examples and exercises are generated as before using a domain-specific problem solver. We also tested the case where the exercises are generated uniformly over all stages as before. The results are shown in Figure 18. Learning from the exercises and examples performed better than before, respectively reaching 91% and 92% accuracies with only 280 exercises. This is to be expected since the test problems are now chosen from a narrower range. However, as predicted, the performance of the uniform exercise distribution is significantly worse especially in the beginning. For example, with 280

exercises, it only achieved an accuracy of 67.3%. This is so because, since the test problems are generated only from the most difficult stage, most of the exercises generated uniformly from all the other stages are useless, leading to slow convergence.

# 7 Discussion and Related Work

This paper is in the tradition of developing and analyzing formal frameworks for speedup learning (Tadepalli & Natarajan, 1996; Natarajan & Tadepalli, 1988; Khardon, 1996). In this work, we view learning as improving the asymptotic complexity of problem solving. Learning is possible when the problem domain exhibits enough structure so that there is an efficient problem solver which can be uncovered from solving a reasonably small set of examples. Consistent with the PAC framework, we allow the learned problem solver to be only approximately correct with a high probability. A formal framework for supervised learning of problem solving strategies was introduced in (Tadepalli, 1991; Tadepalli & Natarajan, 1996). Khardon extended this framework to dynamic, stochastic, and partially observable domains (Khardon, 1999). These results showed that problem solving strategies can be effectively learned from observation if there is a sufficiently strong syntactic bias that constrains them.

The idea of exercises was first introduced by Natarajan in his seminal paper (Natarajan, 1989). Unlike in the supervised setting, where the teacher gives the learner a set of solved problems, in this framework a teacher only gives the learner a set of ordered exercises. The exercises occur as intermediate subproblems of natural problems the learner is expected to solve. This approach takes a compromise position between having the teacher provide solutions to problems as in supervised speedup learning, and letting the learner solve the problems all by itself as in unsupervised speedup learning and reinforcement learning. While giving solutions to random natural problems can be burdensome to the teacher, having the learner solve them without any heuristic guidance is computationally prohibitive. Natarajan showed that by providing the learner with a special distribution of exercises and ordering them by difficulty, it is possible to solve the exercises efficiently and learn from them. We generalized the notion of exercises in (Natarajan, 1989) in many ways. First, we adopted the problem reduction formulation of problem solving which is more general and natural than the state space formulation. Second, we allowed for applying a sequence of operators to reduce an exercise to simpler exercises. Third, we simplified the framework and the proofs. We applied them to three different representations of control knowledge, namely macro-operators, control rules, and decision lists, and three different domains.

Our application of the formal framework to macro-operators is based on Korf's work on macro-operators (Korf, 1985). Korf's implementation starts with an ordering of features with respect to which the domain is serially decomposable, and builds a macro-table by searching exhaustively backwards from the goal. Eight Puzzle has been implemented in Soar in a goal-independent manner by introducing variables. However, the implementation depends on being given a particular ordering of features (Laird et al., 1986). Our algorithm does not need the correct ordering of features in the macro-table to be given, but instead uses the exercises to infer this ordering. More recently Finkelstein and Markovitch implemented a macro-learning algorithm for an $N \times N$ generalization of Eight Puzzle (Finkelstein & Markovitch, 1998). They also exploit the idea of learning from easier problems first, thus validating the approach taken here. However, their motivation is slightly different in that they emphasize *selective* learning of macros. In particular, they learn macros that take any state to a better state as evaluated by the city-block distance heuristic. Their technique is effective in learning a small number of macros that can solve any $N \times N$ puzzle. The current paper can be viewed as a theoretical argument as to why this works. The nature of the $N \times N$ puzzle is that there is a dense stage structure implicitly specified by the city-block distance heuristic. This property coupled with the serial decomposability of the domain makes it very amenable to macro-operator learning.

A difficult issue in learning from exercises is the existence of multiple solutions to a problem. Some solutions may not permit successful generalization. Natarajan's results on control rule learning assumes that the exercise learner returns a unique solution for each problem, i.e., the first according to a lexicographic ordering among the shortest solutions. We made different sets of assumptions that depend on the representation. For example, in the case of control rule learning, we assume that the exercises have unique solutions and there is a set of non-overlapping select sets. The reason for these assumptions is to make sure that the

control rules of the learned problem solvers remain more specific than those of the target problem solver so that learning can be restricted to positive examples, and known results on learning with one-sided error can be applied (Natarajan, 1991). In the case of macro-operator learning, we assume that the domains in the meta-domain are serially decomposable which ensures that solutions that work for one problem work for all. We also changed the framework so that the exercises can be solved in batch making it easier to discover the feature ordering that makes the domain serially decomposable.

The existence of non-overlapping control rules may be too restrictive for certain domains. When the control rules are overlapping, one might use a tie-breaking mechanism for choosing which operator to apply or rely on backtracking to find the correct operator. Our results can be generalized to the case of tie-breaking with a fixed lexicographic ordering as in (Natarajan, 1989). The decision list policy has an implicit tie-breaking mechanism based on the position of the clause in the list and does not assume non-overlapping clauses. Alternatively, one could use macro-operators, which have non-overlapping applicability conditions. Incorporating backtracking search into this framework is more difficult, because of the complex interaction between learning and search. The separation of learning and generalization routines which are representation-dependent from the overall theoretical framework makes it possible to apply our framework to learn control knowledge in a number of domains by choosing an appropriate representation in each case.

Our approach is not applicable when it is not possible to come up with a reasonable set of subtasks which can be solved by the default problem solver. Consider a problem such as traveling sales-person. If the cities belong to a set of continents, where the distances between the continents are much larger than the distances within the continents, it seems reasonable to define problem solving stages which correspond to solving each continent. However, if there is no such easily discernable structure to the problem, there is no easy way to define the problem solving stages, and hence we cannot apply our framework. Secondly, although our definition of the exercise distribution is sufficient for theoretical purposes, in some problems, it may be possible to define a better exercise distribution, knowing the test distribution. This is illustrated in the robot navigation domain, where the uniform exercise distribution performed better than the theoretically motivated exercise distribution when the test distribution is also uniform. More work is needed to understand how to define the exercise distribution for the best expected performance on a test distribution without losing the worst-case theoretical guarantees.

Exercises play the role of shaping rewards in reinforcement learning (Ng, Harada, & Russell, 1999). Shaping rewards are rewards given to encourage the learner to achieve some intermediate subgoals, which may be useful in solving higher level goals. Without such intermediate subgoals, the search is usually too expensive to be useful. It has been noted that the *reward horizon*, or the expected horizon of receiving a non-trivial reward is an important factor that influences the efficiency of reinforcement learning in guiding its search for optimal policy (Laud & DeJong, 2003). Another way to achieve more efficient search in reinforcement learning is to design hierarchical problem spaces (Dietterich, 2000; Sutton, Precup, & Singh, 1999; Parr & Russell, 1998; Barto & Mahadevan, 2003; Seri & Tadepalli, 2002). Part of the reason for these hierarchies to work well is that the learning that occurs in the lower levels of the hierarchy is useful in guiding the search at the higher levels in more fruitful directions.

Bootstrapping the knowledge learned from smaller problems to bigger problems is explored in relational domains as well (Reddy & Tadepalli, 1997; Fern, Yoon, & Givan, 2006). Reddy and Tadepalli (1997) explicitly pursue the idea of learning from exercises in the setting where knowledge is represented as a set of relational goal decomposition rules. They show empirical results that suggest that this approach is applicable in non-trivial planning domains. Fern et al. (2006) explore a technique called “Approximate Policy Iteration,” which starts from a random or uninformed policy and improves it by successive iterations of sample-based policy improvements followed by policy generalization. They generate problems using random walks from the goal state, and use the length of the random walk to measure the degree of their difficulty. Consistent with the current work, they show that learning from easier problems and generalizing their solutions would help solve harder problems in many planning domains.

# 8 Conclusions and Future Work

We conclude that learning from exercises provides a practical compromise between the supervised and unsupervised forms of speedup learning. We showed the applicability of this framework to learning macro-operators, control rules and decision lists, and demonstrated its success in the domains of Eight Puzzle, symbolic integration and a simulated robot domain. Learning from exercises is based on the idea of exploiting what one learns from simpler problems to solve more difficult problems. As such, this phenomenon seems ubiquitous in human learning starting from complex physical skills such as playing basketball to more abstract skills such as programming and theorem proving.

There are several possibilities for future work. This work may be extended to other forms of control knowledge. Although successful experimental results have shown the utility of this approach in relational domains, formal results that characterize their generality are missing. The intractability of learning in such structural domains necessitates other kinds of inputs to the learner. For example, in (Reddy & Tadepalli, 1998), it is shown that it is possible to learn efficient decomposition rules for the blocks world from solved problems and "membership queries" that verify whether an example is covered by a target rule. Extending this result to learning from exercises would provide a theoretical justification of the empirical results of (Reddy & Tadepalli, 1997). Thirdly, we should study practical ways of generalizing this work to unsupervised speedup learning. These include seeking various kinds of "advice" or "hints" from the teacher on the order in which the subgoals and operators must be attempted to solve the training problems. Generalizing the theory and experimental results to stochastic, multi-agent, and partially observable settings are other directions to pursue.

## Acknowledgments

This work is supported by National Science Foundation under grant number IIS-0329278 and Defense Advanced Research Projects Agency under grant numbers FA8650-06-C-7605 and FA8750-05-2-0249. I am indebted to Balas K. Natarajan who initiated me on this work and has been a constant source of inspiration and encouragement. I thank Alan Fern, Roni Khardon, Neville Mehta, Chandra Reddy, and the anonymous reviewers for their comments on earlier drafts.

## References

Barto, A. G., & Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Systems Journal*, *13*, 41–77. Special Issue on Reinforcement Learning.

DeJong, G., & Mooney, R. (1986). Explanation based learning: An alternative view. *Machine Learning*, *1*, 145–176.

Dietterich, T. G. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, *13*, 227–303.

Earley, J. (1970). An efficient context-free parsing algorithm. *Communications of ACM*, *13*(2), 94–102.

Erol, K. (1995). *Hierarchical Task Network Planning: Formalization, Analysis, and Implementation*. Ph.D. thesis, University of Maryland, College Park.

Fern, A., Yoon, S., & Givan, R. (2006). Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *Journal of Artificial Intelligence*, *25*, 75–118.

Fikes, R., Hart, P., & Nilsson, N. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, *3*, 251–288.

Finkelstein, L., & Markovitch, S. (1998). A selective macro-learning algorithm and its application to the $N \times N$ sliding-tile puzzle. *Journal of AI Research*, *8*, 223–263.

Gratch, J., & DeJong, G. (1992). Composer: A probabilistic solution to the utility problem in speedup-learning. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pp. 235–240.

Iba, G. A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning*, *3*, 285–317.

Kearns, M. J., & Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory*. The M.I.T. Press, Cambridge, MA.

Khardon, R. (1996). Learning to take actions. In *Proceedings of the 13th National Conference on Artificial Intelligence*, pp. 787–792.

Khardon, R. (1999). Learning to take actions. *Machine Learning*, *35*, 57–90.

Korf, R. (1985). Macro-operators: a weak method for learning. *Artificial Intelligence*, *26*, 35–77.

Laird, J., Rosenbloom, P., & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, *1*(1), 11–46.

Langley, P. (1983). Learning effective search heuristics. In *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pp. 94–96.

Laud, D., & DeJong, G. (2003). The influence of reward on the speed of reinforcement learning: An analysis of shaping. In *Proceedings of the 20th International Conference on Machine Learning*, pp. 440–447.

Marthi, B., Russell, S., & Wolfe, J. (2007). Angelic semantics for high-level actions. In *Proceedings of the 17th International Conference on AI Planning and Scheduling*, pp. 232–239.

Minton, S. (1990). Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, *42*(2-3), 363–391.

Minton, S., Carbonell, J., Knoblock, C., Kuokka, D., Etzioni, O., & Gil, Y. (1989). Explanation-Based Learning: a problem solving perspective. *Artificial Intelligence*, *40*, 63–118.

Mitchell, T., Keller, R., & Kedar-Cabelli, S. (1986). Explanation based generalization: A unifying view. *Machine Learning*, *1*, 47–80.

Mitchell, T., Utgoff, P., & Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem solving heuristics. In Michalski, R., Carbonell, J., & Mitchell, T. (Eds.), *Machine Learning*, pp. 163–190. Tioga, Palo Alto, CA.

Natarajan, B. (1989). On learning from exercises. In *Proceedings of the 2nd Annual Workshop on Computational Learning Theory*, pp. 72–87.

Natarajan, B. (1991). *Machine Learning: A Theoretical Approach*. Morgan Kaufmann, San Fransisco, CA.

Natarajan, B., & Tadepalli, P. (1988). Two new frameworks for learning. In *Proceedings of the 5th International Machine Learning Conference*, pp. 402–415.

Ng, A. Y., Harada, D., & Russell, S. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the 16th International Conference on Machine Learning*, pp. 278–287.

Parr, R., & Russell, S. (1998). Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems*, *10*, 1043–1049.

Reddy, C., & Tadepalli, P. (1997). Learning goal-decomposition rules using exercises. In *Proceedings of the 14th International Conference on Machine Learning*, pp. 278–286.

Reddy, C., & Tadepalli, P. (1998). Learning Horn definitions: Theory and an application to planning. *New Generation Computing*, *17*, 77–98.

Rivest, R. (1987). Learning decision lists. *Machine Learning*, *2*(3), 229–246.

Seri, S., & Tadepalli, P. (2002). Model-based hierarchical average-reward reinforcement learning. In *Proceedings of the 19th International Machine Learning Conference*, pp. 562–569.

Shavlik, J. (1990). Acquiring recursive and iterative concepts with explanation-based learning. *Machine Learning*, *5*, 39–70.

Sutton, R., & Barto, A. (1998). *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, MA.

Sutton, R. S., Precup, D., & Singh, S. (1999). Between MDPs and Semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, *112*(1-2), 181–211.

Tadepalli, P. (1991). A formalization of explanation-based macro-operator learning. In *Proceedings of the 12th International Joint conference on Artificial Intelligence*, pp. 616–622.

Tadepalli, P. (1992). A theory of unsupervised speedup learning. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pp. 229–234.

Tadepalli, P., & Natarajan, B. (1996). A formal framework for speedup learning from problems and solutions. *Journal of Aritificial Intelligence Research*, *4*, 445–475.

Valiant, L. (1984). A theory of the learnable. *Communications of the ACM*, *27*(11), 1134–1142.