Learning Goal-Decomposition Rules using Exercises

Chandra Reddy & Prasad Tadepalli
Department of Computer Science
Oregon State University, Corvallis, OR-97331. USA
{reddyc,tadepalli}@cs.orst.edu

Abstract

Exercises are problems ordered in increasing order of difficulty. Teaching problem-solving through exercises is a widely used pedagogic technique. A computational reason for this is that the knowledge gained by solving simple problems is useful in efficiently solving more difficult problems. We adopt this approach of learning from exercises to acquire search-control knowledge in the form of goal-decomposition rules (d-rules). D-rules are first order, and are learned using a new "generalize-and-test" algorithm which is based on inductive logic programming techniques. We demonstrate the feasibility of the approach by applying it in two planning domains.

1 Introduction

Planning is computationally an intractable task. Usually, control knowledge is used to make planning manageable; it helps a planner in constraining its search for a solution. The research work in learning control knowledge falls between two extremes: supervised speedup learning and unsupervised speedup learning. In supervised speedup learning, the system is provided with examples or solved instances by a teacher (DeJong & Mooney, 1986; Mitchell, Keller, & Kedar-Cabelli, 1986; Shavlik, 1990). In unsupervised speedup learning, the system is given only problems without their solutions (Laird, Rosenbloom, & Newell, 1986; Minton, 1988). In this paper, we explore a middle course of learning from exercises— useful subproblems of natural problems that are ordered in increasing order of difficulty.

Teaching problem-solving through exercises is a widely used pedagogic technique. A human teacher selects certain problems and orders them according to their level of difficulty to form a sequence of exercises. A human student starts by solving simple problems first; then attempts harder problems by applying the knowledge gained from solving the earlier problems; and then attempts still harder problems, and so on. This process continues until the student learns the concepts satisfactorily. Machine learning of problem solving using exercises, apart from following this pedagogic tradition, offers a compromise between supervised speedup learning and unsupervised speedup learning. Supervised speedup learning, although more efficient than the latter, places the burden of providing the solutions to the training problems on the teacher—usually a human. Unsupervised speedup learning, in contrast, expects the learner to solve the training problems, while unburdening the teacher. However, this is computationally hard for the learner for it lacks control knowledge and, hence, its only recourse is brute-force search. In the exercises approach, the teacher has the task of providing an exercise set—a sequence of problems ordered by difficulty. The learner has to solve the exercise problems using the bootstrapping method akin to the above-described method followed by a human student.

The exercises approach for speedup learning has been used for learning control rules (Natarajan, 1989). In this work, we use exercises for learning goal-decomposition rules (d-rules). The learning of control rules in (Natarajan, 1989) is essentially propositional, whereas learning d-rules is first-order or relational. Moreover, d-rules can be recursive, which complicates solving of exercise problems: to solve a particular goal, the learner should already know something about solving that very goal. Previously, learnability of d-rules in supervised setting—using solved examples—is demonstrated in (Reddy, Tadepalli, & Roncagliolo, 1996). In this work, we instead use exercises to learn d-rules.

Our approach, implemented as a system called X-Learn, follows two main steps: For each exercise, (1) an exercise-solver solves the exercise, and outputs the solution (the plan) and the subgoals used; and (2) a first-order inductive learner forms an example d-rule using the initial state as the d-rule condition and the subgoals as the d-rule subgoals, and then uses a "generalize-and-test" algorithm to include this example d-rule in the previous hypothesis d-rules. In the generalization phase, the generalize-and-test algorithm finds the least general generalization (lgg) of the example d-rule with previously formed similar hypothesis d-rules. In the test phase, to ensure appropriate generalization, the algorithm generates examples of the hypothesized d-rule and tests whether the examples are correct.

The rest of the paper is organized as follows. Section 2 provides a short introduction to d-rules. Section 3 discusses the learning approach and the related algorithms. Section 4 discusses the results of experiments. Section 5 concludes with a discussion of related issues and future work.

2 D-rule Representation

Macro operators and control rules are two prominent methods of representing control knowledge (Minton, 1988). D-rules is another method for this purpose.

Representation. A d-rule is a 3-tuple $\langle g, c, sg \rangle$ that decomposes a goal g into a sequence of subgoals sg, provided the condition c holds. The goal component g is a single literal. The condition component c and the subgoals component sg are conjunctions of literals. To represent subgoal order, sg, in addition, has an implicit left-to-right ordering. All literals in a d-rule are first-order function-free positive literals. A requirement on the subgoals component, sg, is that each variable in it occurs also either in g or in c. A d-rule, in STRIPS-world domain, looks like this:

```
  g:  (in-room robot ?rg)
  c:  ((in-room robot ?rst) (left-of ?rfr ?rg)
      (left-of ?rst ?rg) (door ?d ?rfr ?rg) )
 sg:  ⟨(in-room robot ?rfr) (in-room robot ?rg)⟩
```

This d-rule can be read as follows: to reach the room `?rg`, starting from `?rst`, go to the room `?rfr` that is immediately to the left of `?rg`, and then enter the `?rg` room. This rule is recursive in the sense that, to reach `?rfr`, the planner may have to make use of the same rule. A d-rule may also decompose a goal into primitive operations which can be readily executed. For example, going from the `?rfr` room to the neighboring `?rg` room, the robot needs to open the connecting door and enter, which are primitive operations.

D-rules are a specialization of hierarchical transition networks of (Erol, Hendler, & Nau, 1994), the only difference being that the subgoals are totally ordered in d-rules. D-rule representation compares favorably against macro operators and control rules, in terms of computational complexity of planning when there is a loosely coupled set of goals and subgoals.

D-rule planner. The d-rule-based planner for a domain takes as input a goal, a state from which the goal needs to be achieved, a set of d-rules, and the domain theory (comprising primitive-operator definitions, and background theory). The planner finds the d-rules that correspond to the goal, and picks a d-rule whose condition is satisfied in the current state. The chosen d-rule suggests a subgoal sequence. The suggested subgoals in the sequence are achieved one by one. Each subgoal in turn becomes a goal, and has its own d-rules. So, the planner achieves goals recursively, until the recursion bottoms out when a d-rule suggests only primitive operators as subgoals. Primitive operators are operators of the domain which are readily executable without recourse to a d-rule. The planner outputs the plan (the operator sequence) and the final state, in which the goal should be satisfied.

Since d-rules require their subgoal components to have only the variables in the condition and goal components, when the planner is trying to achieve a particular goal and the condition of a d-rule has been satisfied, the subgoals will all be ground. Moreover, the subgoals are ordered. This makes planning efficient. However, in general, there may be multiple d-rules for a goal, and their conditions may not be disjoint. This presents a choice for the planner. There are four design options here: (1) require that d-rules be disjoint; (2) if not, require that any d-rule choice for a goal achieve the goal; (3) allow backtracking; or (4) have control rules that know how to choose among different d-rules when there is a choice. Allowing backtracking makes the planner inefficient. We assume that the set of d-rules the planner gets is "good"—i.e., any applicable d-rule achieves its goal. With good d-rules, planning is linear in the number of d-rule applications, and, hence, in the solution or plan length. The d-rule planner can be incomplete if the d-rules are not perfect; but it is always efficient.

3 Learning from Exercises

Learning from exercises requires that the teacher give a sequence of problems ordered according to their level of difficulty: from least difficult problems which come first, to most difficult problems which come last. This is similar to the way exercises are presented at the end of a lesson in a textbook on mathematics. In the examples framework, on the other hand, the examples the learner gets are randomly chosen from a natural distribution. However, in the exercises framework, the burden of solving the input problems is on the learner, unlike in the examples framework where that burden is on the teacher.

We have not yet defined what a problem's level of difficulty means. In (Natarajan, 1989), the solution length of a problem is chosen as the level of difficulty—i.e., the problems with higher solution length are more difficult. This, however, does not work for d-rules. A d-rule, depending on the problem it is applied to, may produce varying solution lengths. Therefore, exercises are ordered, instead, according to a bottom-up examination of the goal-subgoal hierarchy of the domains. That is, an exercise related to a goal in the hierarchy is less difficult than an exercise related to goals that are higher in the hierarchy. This, however, does not work when there is recursion among goals, for there is no strict hierarchy then. In the case of recursion, exercises for base cases are considered less difficult than the exercises for more general recursive cases.

```
X-Learn
  dRuleSet ← {}
  ⟨state, goal⟩ ← Next-Exercise() /*next problem*/
  while ⟨state, goal⟩ is not FAIL do {
    depthLimit ← MIN-DEPTH
    done ← FALSE
    while depthLimit <= MAX-DEPTH
          and not done do {
      ⟨plan, subgs⟩ ←
          X-Solver(prob, dRuleSet, depthLimit, nil, nil)
      if ⟨plan, subgs⟩ is not FAIL
          then done ← TRUE
      else depthLimit ← depthLimit + 1
    } /* IDS */
    exampleDR ← Preprocess(⟨goal, state, subgs⟩)
    dRuleSet ← Generalize-Test(dRuleSet, exampleDR)
    ⟨state, goal⟩ ← Next-Exercise()
  } /* all exercises are processed */
  return dRuleSet
```

Figure 1: Algorithm for learning from exercises

First we present an overview of the learner. (See Figure 1 for the pseudo-code.) The learner (X-Learn) takes in exercises, which are in an order of increasing difficulty, as input. The goal-subgoal hierarchy of the domain can also be an optional input. X-Learn starts with an empty set of d-rules. It makes use of a routine **Next-Exercise**() to get the next exercise. An exercise is a problem—i.e., a state-goal pair. X-Learn employs an exercise solver X-Solver that uses depth-first search (DFS). X-Solver utilizes X-Learn's own previously learned d-rules along with primitive domain actions to solve the exercises. With DFS there is a danger of the solver going too deep in a wrong direction. Therefore, X-Learn uses iterative-deepening depth-first search (IDS)—i.e., starting with a depth limit and doing DFS until that depth limit is reached, and if no solution is found, increasing the depth limit and re-doing DFS with the new depth limit, and so on (Korf, 1987). IDS also guarantees that the solution, if any, is reached with the fewest number of operator applications. The solver returns the solution plan that solves the exercise problem along with the subgoal sequence used to solve the exercise.

Once an exercise is solved, an optional preprocessing step (`Preprocess`) prunes irrelevant literals and introduces abstract literals using a domain theory. This step prepares an example for the next, generalization step.

Next, X-Learn calls a first-order generalizer (Generalize-Test) to include the new case in the learned set of d-rules. Generalize-Test tries to include the new case in one of the existing d-rules by generalizing the d-rule. If that is not possible, the new case forms the basis for a new d-rule.

In the following subsections, we describe the exercise solver, X-Solver, the preprocessing step, Preprocess, and the d-rule generalizer, Generalize-Test, in detail.

3.1 Solving Exercises

To solve exercises in a domain, our exercise solver employs previously learned control knowledge—d-rules learned from the exercises seen so far—and primitive actions of a domain. Of course, to solve the initial exercises, only the primitive operators are used, for the learner has had little opportunity to learn yet.

The exercise solver, X-Solver, employs fixed-depth depth-first search to solve the exercises. It uses, along with primitive actions of a domain, previously learned d-rules as operators for the search. Primitive actions of a domain are specified as STRIPS-style operators, and their execution is straightforward (Fikes, Hart, & Nilsson, 1972).

```
X-Solver(prob, dRs, depth, prevPlan, prevSgs)
  ⟨state, goal⟩ ← prob
  if goal is reached then return ⟨prevPlan, prevSgs⟩
  else if depth > 0 then {
    operators ← dRs ∪ domainActions
    for each op ← operators do{
      if op's condition is satisfied then {
         if op is a d-rule then
           ⟨newState, plan⟩ ← D-rule-plan(prob, op, dRs)
         else /*op is a primitive action*/
           ⟨newState, plan⟩ ← execute(state, op)
         sgs ← prevSgs + op or op's goal /*append*/
         plan ← prevPlan + plan /*append*/

         newProb ← ⟨newState, goal⟩
         ⟨finalPlan, finalSgs⟩ ←
               X-Solver(newProb, dRs, depth − 1, plan, sgs)
               /* look deeper */
         if ⟨finalPlan, finalSgs⟩ is not FAIL
               return ⟨finalPlan, finalSgs⟩
      } /* if op's cond is...  */
    } /* for each */
    return FAIL /*no op is successful*/
  } /* else if depth */
  else return FAIL /* gone too deep!  */
```

Figure 2: DFS-based exercise solver

First, the solver checks whether the conditions of the action are satisfied. If yes, the current state is modified by adding and deleting, respectively, the add- and delete literals of the action. But, to execute d-rules, X-solver employs the d-rule-based planner (D-rule-plan). The d-rule-based planner applies the d-rules suggested by the search. If the conditions of the d-rule to be executed are satisfied in the current state, using a set of bindings suggested by this match, the solver instantiates the subgoal sequence of the d-rule. Then, X-solver asks D-rule-plan to achieve the subgoal sequence. D-rule-plan has the primitive actions of the domain and the current d-rule base available to it. D-rule-plan, if successful, returns a plan applied in achieving the subgoal sequence, and the resultant state; else, it reports failure. In case of failure and when the conditions of the action or the d-rule are not satisfied in the current state, the solver considers alternative operators. If there are no more untried alternative operators for the current state, X-solver backtracks and considers alternate operators for the previous state. In case of successful application of the operator (action or d-rule), the resultant state is made the current state and the process continues till the goal is reached or the fixed depth limit is reached. See Figure 2 for the algorithm.

Let us consider an example from air-traffic control (ATC) domain (see Section 4 for details.). To land a plane which is at holding level 2, it has to be first moved to holding level 1 (node `n2` in Figure 3), and then be landed on a free runway (node `n4` in Figure 3). To execute the d-rule corresponding to `move-plane`, the d-rule planner achieves the subgoals—such as `move-cursor, select-plane, deposit-plane`—suggested by the d-rule, and returns the resultant state and the sequence of actions. Similarly, for `land-plane`. Presumably, the sequence of exercises is such that the d-rules corresponding to the goals `move-plane` and `land-plane` have already been learned. In this example, the successful solution path contains only d-rule operators, and no primitive-action operators. X-solver finally outputs the instantiated goals of the d-rules tried—`(move-plane P L1) (land-plane P)`—as the subgoals for a d-rule hypothesis for the main goal—`land P`—and the concatenation of the actions produced by each operator as the final plan.

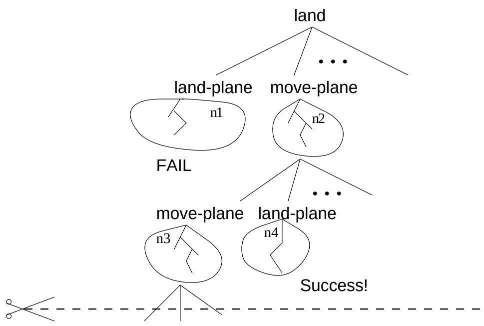

Figure 3: Exercise solver achieving the goal `land` using IDS with the current depth limit 2

If the learner is provided with the goal-subgoal hierarchy, X-Solve can exploit this to focus its search. In particular, it can limit the operator set used in the search to only those d-rules and primitive actions that address the subgoals of the current goal. In the above example, if the learner knows that `land` has only `land-plane` and `move-plane` as subgoals, then it can try only the instantiations of the d-rules of `land-plane` and `move-plane` as the operators, and ignore others.

3.2 Example Preprocessing Using Domain Theory

If X-Solver succeeds on an exercise, the resultant solved exercise forms an example of a d-rule: the goal and the initial state of the exercise problem correspond to the example d-rule's goal and condition, respectively; the subgoal sequence output by the exercise solver for the exercise problem corresponds to the subgoal part of the example d-rule. This example d-rule (or simply, example) is subjected to explanation-based pruning and abstract-literal introduction steps before the inductive generalization process. These two steps comprise Preprocess.

Explanation-based pruning. Examples may contain literals describing relations among objects that may be irrelevant to a target d-rule. For instance, in the above ATC example, the relations describing the planes other than `P` are irrelevant.

The extraneous literals, unless eliminated, increase the generalization cost. Our solution is to first explain the condition part of an example using the domain theory. Then collect the literals that are leaves of the proof tree that explains the example.[1] This solution limits the literals in the example to only the relevant ones.

Abstraction by forward chaining. If d-rules are specified using literals expressed at a higher level of abstraction than the literals in the operator add- and delete-lists, they are more readable and concise. Furthermore, abstract literals may be used to cover different disjunctive cases. Because the conditions in d-rules are conjunctive, without abstract literals, we need a separate d-rule for each different case. Consider, for example, adjacency of two rooms. A room, Room1, is adjacent to another room, Room2, if Room1 is to the north, south, east, or west of Room2. Without the abstract literal for adjacency, we may need four separate d-rules—one each for each of the directions. Thus, abstract literals reduce the number of d-rules required for a domain.

Axioms in the domain theory help the planner in understanding these high-level literals by translating them into the planner's language. For example, in the ATC domain, the high-level literal `(wind-axis NS)` is true when `(wind-direction north)` or `(wind-direction south)` is true.

To introduce these abstract literals, we employ forward chaining on the condition part of an example using the domain axioms. These axioms are shallow—mostly one-level—rules; therefore, there is no danger of forward chaining introducing too many literals.

For the example of Figure 3, the example d-rule, after the two steps, is

```
g:  (land P)
c:  ((at P 10) (type P prop) (runway-cond wet)
    (free 1) (hold-level 1 10) (cursor-loc 4)
    (fuel P 5) (wind-speed high) (wind-axis NS))
sg: ⟨(move-plane P L1) (land-plane P)⟩
```

[1]This is basically explanation-based generalization (EBG) without the variablization step. We do not use the variablization step of EBG, for the inductive learner in Section 3.3 does the necessary variablization anyway.

3.3 Generalization of d-rules

Given an example d-rule with all three components (goal, condition and subgoals), and a set of d-rules learned so far, the task is to form a new set of d-rules that cover the example. The sketch of the algorithm (Generalize-Test) to accomplish this is given in Figure 4.

First, Generalize-Test tries to include the example in one of the existing d-rules. If that is not possible, it creates a new d-rule with the current example as the basis.

The inclusion of an example d-rule in an existing d-rule is done employing Plotkin's least general generalization (lgg) (Plotkin, 1970; Muggleton & Feng, 1990). The lgg of two conjunctions C_1 and C_2 is $\bigwedge_{p_1 \in C_1, p_2 \in C_2} lgg(p_1, p_2)$. The lgg of two predicates $p(a_1, a_2 \ldots a_n)$ and $p(b_1, b_2 \ldots b_n)$ is a predicate $p(lgg(a_1, b_1), lgg(a_2, b_2) \ldots lgg(a_n, b_n))$; if the predicate symbols are not equal, their sign differs or their arity differs, then their lgg is True. The lgg of two terms a and b is itself, if they are the same; else, it is a variable v, where v stands for the ordered pair (a, b) throughout the computation of the lgg of the pair of conjunctions. For example, let $C_1 = l_1(a,b) \wedge l_2(a,c) \wedge l_3(b)$ and $C_2 = l_1(c,d) \wedge l_1(b,a) \wedge l_2(c,c)$. Then $lgg(C_1, C_2) = l_1(x,y) \wedge l_1(z,u) \wedge l_2(x,c)$, where the variables x, y, z and u stand for the pairs (a,c), (b,d), (a,b) and (b,a), respectively.

The use of lgg guarantees that a d-rule of the learner and an example are specializations of a target d-rule if and only if their lgg is also a specialization of the same target d-rule (Reddy & Tadepalli, 1997). Hence, we find the lgg of each hypothesis d-rule and the new example, and test it to see if it, indeed, is a specialization of one of the target d-rules. The learner tests the hypothesis d-rule (the lgg), first by generating a problem compatible with the condition of the hypothesis, and decomposing it to the subgoals suggested by the hypothesis. Next, this sequence of subgoals is given to the solver to solve. If the solver is successful, then it generates another test problem and attempts to solve it, in a similar way. If the hypothesis d-rule is successful on a number of test problems, then the hypothesis d-rule is accepted as correct. If not, by concluding that the example d-rule and the learner's candidate rule for inclusion are not compatible, the learner returns FAIL. This process is accomplished by Test in the generalization algorithm of Figure 4.

There is a complication here. The size of lgg of two conjunctions C_1 and C_2 is, in the worst case, the prod-

```
Generalize-Test(dRuleSet, egDrule)
   genDR ← FAIL
   while genDR is FAIL for each dRule in dRuleSet do
      genDR ← Lgg-generalize(dRule, egDrule, dRuleSet)
   if genDR is not FAIL then
      replace dRule by genDR
   else dRuleSet ← egDrule + dRuleSet
                        /*example becomes a new drule*/
   return dRuleSet

Lgg-generalize(dRule, eg, dRuleSet)
   if goals and subgoals of dRule and eg
                             are compatible then{
      lgg ← Least-general-generalization(dRule, eg)
      if Test(lgg, dRuleSet) is successful then
         return(Reduce(lgg, dRuleSet))
      else return FAIL
   }
   else return FAIL

Reduce(lgg, dRuleSet)
  for each lit in lgg do
    if Test(lgg − {lit}, dRuleSet) is successful then
      remove lit from lgg
  return lgg
```

Figure 4: Generalize-and-test algorithm for generalizing d-rules, given an example d-rule

uct of the sizes of C_1 and C_2. That means the size of the generalized d-rule may grow exponentially in the number of examples. To overcome this problem, we use testing to remove irrelevant literals in the d-rule: remove a literal in the condition of the hypothesis d-rule if Test on the resulting d-rule is successful. During this testing, we would like the examples generated by Test to be of "near-miss" kind—i.e., in removing a literal from the condition of a hypothesis d-rule, we would like test examples in which that literal is not true. Apart from keeping hypothesis d-rules small, the removal of irrelevant literals has an advantage of further generalizing them without the risk of overgeneralization. This is accomplished by Reduce in Figure 4.

4 Experimental Results

We have implemented the algorithms discussed so far as a system called X-Learn, in Common Lisp. X-Learn is tested on two domains: (1) A variant of STRIPS world (SW), with recursive d-rules; and (2) Kanfer-Ackerman air-traffic control (ATC) task. The purpose is to see how feasible and effective the approach of learning d-rules from exercises is. Since our d-rule-based planner is guaranteed to be efficient, we evaluate the performance of the system only by its coverage on test problems. In particular, we have measured the coverage of learned d-rules on randomly selected test cases against the number of exercises supplied. The test problems are natural problems that occur in a domain and are among the hardest, whereas the exercises are selected problems such that they span all levels of difficulty. The exercise set at each data point, except the first, includes the exercise set at the previous data point. The test set is, however, fixed for all the data points. Each exercise set has an almost equal number of exercises for all the goals at all levels of difficulty. Moreover, the exercises in a set are ordered in increasing order of difficulty. To expedite the experiments, we have supplied the solvers with the goal-subgoal hierarchies in both domains, so that X-Solve can focus its search in solving the exercises.

SW. This domain is a slight variation of STRIPS world (Fikes et al., 1972). The room configuration we use is a grid, unlike in (Fikes et al., 1972) where it is a random configuration. The rooms are laid out as a grid with doors between neighboring rooms. Each room can have zero or more boxes. There is a robot that can open or close doors of the room that it is in. It can go from room to room with or without pushing a box. Typical tasks for the robot are to go from one room to another room, or move a box from some room to some other room. The particular configuration we used is a 4×4 grid, with 3 to 6 boxes distributed among the 16 rooms. There are about 7 levels of difficulty. Basic tasks such as opening and closing doors, and holding and releasing objects are the least difficult. Going from a room to another room is recursive. For example, to go to a goal room that is to the left of the starting room, go to the right-neighbor room of the goal room, where going to the right-neighbor room may follow the same rule. Moving or pushing a box to a neighboring room, from a room, is at the next level. The most difficult tasks are starting from a room, and pushing a box from another room to yet another room.

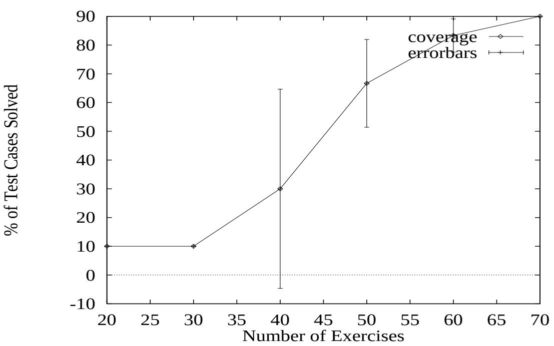

Figure 5: Performance of X-Learn in SW

Figure 5 gives the learning curve: coverage vs. number of exercises. At the end of training, X-Learn came up with 24 d-rules for SW. These d-rules are such that they are applicable to any-sized grid. These results demonstrate that learning from exercises is a feasible approach to learn recursive d-rules.

ATC. This domain is a simplified version of the Kanfer-Ackerman air-traffic control task (Ackerman & Kanfer, 1993). The task here is to land a plane that is in a holding position at some holding level onto an appropriate runway. New planes come in from a queue of incoming planes. There are 3 holding levels, each with 4 holding positions. Planes can come in at any position of any level. But they can land only from the first level. So, before landing, planes should be brought to the first level. There are other restrictions, such as wind condition, runway condition, type of the plane and types of runways, that affect landing of a plane. The primitive actions in this domain are selecting a plane, moving the cursor between holding positions, depositing a plane on a holding position or on a runway. The exercises that are the least difficult correspond to these basic tasks. There are 4 other gradations of difficulty: moving a plane from a holding level to the next holding level is the next gradation; landing a plane from the first holding level is the next difficult task; the most difficult exercises concern landing a plane without a runway being specified and from a holding level other than the first.

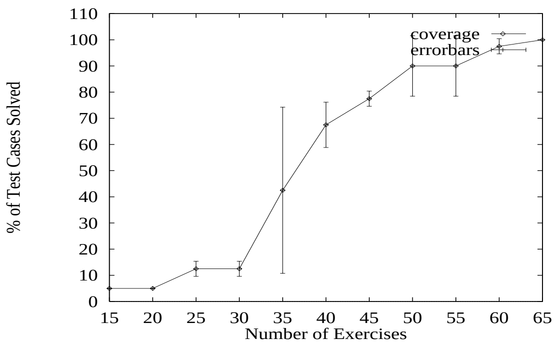

Figure 6: Performance of X-Learn in ATC

Figure 6 gives the learning curve: coverage vs. number of exercises. X-Learn came up with 14 d-rules for this domain at the end of training which perform correctly on all 20 test cases.

The error bars in Figure 5 and Figure 6 indicate one standard deviation on either side of the mean. In both the domains, if an exercise cannot be solved within a maximum depth limit, it is abandoned, and no learning results from it. Maximum depth limits are set such that if an exercise cannot be solved, it is only due to the inadequacy of X-Learn's learned d-rules. The inadequacy of X-Learn's d-rules is due to the inadequacy of training. (In the plots, unsolved exercises are also counted when counting the number of exercises used by X-Learn.) Some data points in both the plots have large error bars. The reason for large error bars early on in both the plots is that in some runs the exercises supplied resulted in X-Learn learning good lower level d-rules, and thus enabling solving of and learning from higher-level exercises. On the other hand, in Figure 6, the large error bars later on are due to some runs getting unlucky: the randomly picked exercises had coincidences, in the sense that they were about the same plane or the same runway, which prevented the occurrence of inductive leap in the d-rules learned from these exercises.

5 Discussion and Future Work

We have shown that learning d-rules from exercises is a feasible and effective approach. The approach used in X-Learn is directly related to speedup learning from examples and inductive logic programming (ILP). In the following, we position our work in the context of the related work in those two areas.

The speedup learning from examples system that is closely related to our work is ExEL (Reddy et al., 1996). This system learns goal-decomposition rules using examples—problems and their solutions. Unlike X-Learn's generalization algorithm, ExEL's does not use *lgg*, but uses an ad-hoc method for generalizing a hypothesis d-rule with an example d-rule. Therefore, ExEL's generalization method lacks theoretical justification. In comparison, the generalize-and-test algorithm of X-Learn, apart from being theoretically justified, is simpler. ExEL uses membership queries for checking the validity of a generalization. X-Learn instead uses self-testing, and is thus more autonomous than ExEL. It is not easy to replace membership queries with testing in ExEL. The subgoals part of the example d-rule in ExEL's approach has several irrelevant literals, unlike in the exercises approach where only relevant subgoals are identified. This causes difficulties in solving test problems because several of the proposed subgoals may be spurious. Moreover, in ExEL's approach, the learning is not in any particular order. Therefore, the d-rules required to solve a test subgoal may not be already present in the learner's repertoire of learned d-rules.

Considering that planning without search-control knowledge is intractable, any unsupervised speedup learner may, in practice, need to behave like the learning-from-exercises systems to be successful. If an unsupervised learner gets a hard problem in the beginning, it may get bogged down in trying to solve the problem, unless there is a limit on the amount of time it can spend on a problem. As a result, learning may not be feasible because there may not be enough of simple problems to learn from. If there are enough of simpler problems, then its behavior approaches that of learning from exercises. In any case, in the process, it may squander computing time and available training problems.

A set of d-rules could be looked at as first-order function-free Horn programs. In this respect, learning d-rules is an ILP problem. From (Cohen, 1995c), one can infer that this concept class is hard to learn without external help. Our system uses testing to learn this class of concepts. GOLEM, an ILP system, instead restricts the concept class to determinate clauses, which means in a clause there is a unique binding to all the free variables of a literal, given the bindings of all previous literals (Muggleton & Feng, 1990). In fact, this concept class with the further restriction of keeping the depth of the dependency chain of the variables in a clause constant, is shown to be PAC-learnable from examples (Cohen, 1995b; Džeroski, Muggleton, & Russell, 1992). Unfortunately, these restrictions are too limiting on the expressiveness of d-rules to enable efficient planning. Therefore, we circumvent this problem by employing exercises and self-testing. It is shown in (Reddy & Tadepalli, 1997) that d-rules for a single goal are PAC-learnable using the algorithm in Figure 4 with a modification—a membership query replacing Test. Learning d-rules for a single goal is equivalent to learning Horn clauses for a single head literal—i.e., Horn definitions (Reddy & Tadepalli, 1997). Horn programs—each of which is a set of Horn definitions—are shown to be not PAC-learnable from examples alone (Cohen, 1995a). The ordering-the-learning idea of the exercises approach, however, enables the Horn-definition learning algorithms to learn Horn programs also. X-Learn, in fact, demonstrates it in learning d-rules for multiple goals, which are equivalent to Horn programs.

FOIL is another ILP system (Quinlan, 1990) that is employed for learning control knowledge. DOLPHIN (Zelle & Mooney, 1993), Grasshopper (Leckie & Zukerman, 1993) and SCOPE (Estlin & Mooney, 1996) are three such systems that embed FOIL for learning control rules. FOIL, as well as GOLEM, are batch-learning algorithms. Therefore, the incremental nature of the exercises approach precludes using such systems for inductive learning in the exercises setting. One other difference between our system and the above systems is in the way negative examples are generated.

DOLPHIN, Grasshopper and SCOPE consider the choices that did not lead to a goal as the negative examples of a "good" control rule. In our setting, the choices the exercise solver makes are over what subgoals to use. Since we get the subgoal literals of a prospective d-rule directly from the exercise solver, the subgoal-learning part does not need any powerful inductive learner. So, the choice sequences of subgoals that did not lead to the goal are of no significant use for subgoal learning. However, they could possibly be used for learning the conditions part: each one is a negative example for a d-rule that has the matching goal and sequence of subgoals. This is because the sequence of subgoals in the example could not lead to the goal for the example's conditions are wrong. However, instead of using these negative examples, X-Learn generates self-critical examples for testing in an active manner, for the following connected reasons. First, not every hypothesis d-rule of the learner may have a negative example in the set collected during the search. Second, even if there are negative examples that address a hypothesis d-rule, they may not be near-miss examples. In X-Learn, since the learner has control over which examples are generated, it can generate suitable near-miss examples.

The way the testing examples are generated and used is somewhat similar to the way Gil in (Gil, 1993, 1994) generates and uses experiments in learning operator models. Gil uses them to specialize overgeneral conditions, whereas we use test examples to generalize overspecific conditions. In our case, to eliminate a literal for generalizing a condition, we choose a literal that is already there in the condition. Gil, in contrast, needs to consider all possible literals and their various variablizations, to add to a condition that is to be specialized. To combat the explosive nature of this process, Gil employs heuristics to choose a literal to add.

Similar to our approach in learning goal decompositions, the SteppingStones system also learns subgoal sequences (Ruby & Kibler, 1991). However, SteppingStones learns from single examples using an EBL-like method, unlike our system, X-Learn, which utilizes multiple examples to learn a goal decomposition. Different examples that have a similar goal decomposition may not share their entire solutions, and thus may not share many literals in their initial states. Combining such examples to form a d-rule enables X-Learn to eliminate unnecessary literals more easily, thus arriving at more concise and general rules.

In the future, we would like to establish PAC-learnability of d-rules using exercises, similar to the results for control rules in (Natarajan, 1989). Also, we would like to extend this work to enable learning in the context of real-time planning.

Acknowledgments. We gratefully acknowledge the support of ONR under grant# N00014-95-1-0557. We thank the Machine Learning Reading Group at OSU and the anonymous reviewers for their many helpful comments.

References

Ackerman, P., & Kanfer, R. (1993). *Kanfer-Ackerman Air Traffic Control Task© CD-ROM Database, Data-Collection Program, and Playback Program*. Dept. of Psychology, Univ. of Minn., Minneapolis, MN.

Cohen, W. (1995a). Pac-learning non-recursive prolog clauses. *Artificial Intelligence*, *79*(1), 1–38.

Cohen, W. (1995b). Pac-learning recursive logic programs: efficient algorithms. *Jl. of AI Research*, *2*, 500–539.

Cohen, W. (1995c). Pac-learning recursive logic programs: negative results. *Jl. of AI Research*, *2*, 541–573.

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

Džeroski, S., Muggleton, S., & Russell, S. (1992). Pac-learnability of determinate logic programs. In *Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory*, pp. 128–135.

Erol, K., Hendler, J., & Nau, D. (1994). HTN planning: complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*. AAAI Press.

Estlin, T., & Mooney, R. (1996). Multi-strategy learning of search control for partial-order planning. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 843–848. AAAI/MIT Press.

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

Gil, Y. (1993). Efficient domain-independent experimentation. In *Proceedings of the Tenth International Conference on Machine Learning*, pp. 128–134.

Gil, Y. (1994). Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 87–95.

Korf, R. (1987). Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, *27*(1), 97–109.

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

Leckie, C., & Zukerman, I. (1993). An inductive approach to learning search control rules for planning. In *Proceedings of the 13th IJCAI*, pp. 1100–1105.

Minton, S. (1988). *Learning Search Control Knowledge*. Kluwer Academic Publishers, Boston, MA.

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

Muggleton, S., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pp. 368–381. Ohmsha/Springer-Verlag.

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

Plotkin, G. (1970). A note on inductive generalization. In Meltzer, B., & Michie, D. (Eds.), *Machine Intelligence*, Vol. 5, pp. 153–163. Elsevier North-Holland, New York.

Quinlan, J. (1990). Learning logical definitions of from relations. *Machine Learning*, *5*, 239–266.

Reddy, C., & Tadepalli, P. (1997). Learning Horn definitions using equivalence and membership queries. Tech. rep., Dept. of Comp. Sci., Oregon State Univ. Submitted to ILP-97.

Reddy, C., Tadepalli, P., & Roncagliolo, S. (1996). Theory-guided empirical speedup learning of goal-decomposition rules. In *Proceedings of the 13th International Conference on Machine Learning*, pp. 409–417. Morgan Kaufmann.

Ruby, D., & Kibler, D. (1991). Learning subgoal sequences for planning. In *Proceedings of AAAI-91*. AAAI Press.

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

Zelle, J., & Mooney, R. (1993). Combining FOIL and EBG to speedup logic programs. In *Proceedings of the 13th IJCAI*, pp. 1106–1111.