Appears in *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pp. 843-848, Portland, OR, August 1996.

Multi-Strategy Learning of Search Control for Partial-Order Planning*

Tara A. Estlin and Raymond J. Mooney
Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712
{estlin,mooney}@cs.utexas.edu

Abstract

Most research in planning and learning has involved linear, state-based planners. This paper presents SCOPE, a system for learning search-control rules that improve the performance of a *partial-order* planner. SCOPE integrates explanation-based and inductive learning techniques to acquire control rules for a partial-order planner. Learned rules are in the form of selection heuristics that help the planner choose between competing plan refinements. Specifically, SCOPE learns domain-specific control rules for a version of the UCPOP planning algorithm. The resulting system is shown to produce significant speedup in two different planning domains.

Introduction

Efficient planning often requires domain-specific search heuristics; however, constructing appropriate heuristics for a new domain is a difficult task. Research in learning and planning attempts to address this problem by developing methods that automatically acquire search-control knowledge from experience. Most work has been in the context of linear, state-based planners (Minton 1989; Leckie & Zuckerman 1993; Bhatnagar & Mostow 1994). Recent experiments, however, support that partial-order planners are more efficient than total-order planners in most domains (Barrett & Weld 1994; Minton *et al.* 1992). However, there has been little work on learning control for partial-order planning systems (Kambhampati et al., 1996).

In this paper, we describe SCOPE, a system that uses a unique combination of machine learning techniques to acquire effective control rules for a partial-order planner. Past systems have often employed *explanation-based learning* (EBL) to learn control knowledge. Unfortunately, standard EBL can frequently produce complex, overly-specific control rules that decrease rather than improve overall planning performance (Minton 1989). By incorporating induction to learn simpler, approximate control rules, we can greatly improve the utility of acquired knowledge (Cohen 1990). SCOPE (Search Control Optimization of Planning through Experience) integrates *explanation-based generalization* (EBG) (Mitchell et al., 1986; DeJong & Mooney, 1986) with techniques from *inductive logic programming* (ILP) (Quinlan 1990; Muggleton 1992) to learn high-utility rules that can generalize well to new planning situations.

SCOPE learns control rules for a partial-order planner in the form of selection heuristics. These heuristics greatly reduce backtracking by directing a planner to immediately select appropriate plan refinements. SCOPE is implemented in Prolog, which provides a good framework for learning control knowledge. A version of the UCPOP planning algorithm (Penberthy & Weld 1992) was implemented as a Prolog program to provide a testbed for SCOPE. Experimental results are presented on two domains that show SCOPE can significantly increase partial-order planning efficiency.

The UCPOP Planner

The base planner we chose for experimentation is UCPOP, a partial-order planner described in (Penberthy & Weld 1992). In UCPOP, a partial plan is described as a four-tuple: $\langle \mathcal{S},\mathcal{B},\mathcal{O},\mathcal{L} \rangle$ where $\mathcal{S}$ is a set of actions, $\mathcal{O}$ is a set of ordering constraints, $\mathcal{L}$ is a set of causal links, and $\mathcal{B}$ is a set of codesignation constraints over variables appearing in $\mathcal{S}$. Actions are described by a STRIPS schema containing precondition, add and delete lists. The set of ordering constraints, $\mathcal{O}$, specifies a partial ordering of the actions contained in $\mathcal{S}$. Causal links record dependencies between the effects of one action and the preconditions of another. These links are used to detect *threats*, which occur when a new action interferes with a past decision.

UCPOP begins with a null plan and an agenda containing the top-level goals. The initial and goal states are represented by adding two extra actions to $\mathcal{S}$, A_0

*This research was supported by the NASA Graduate Student Researchers Program, grant number NGT-51332.

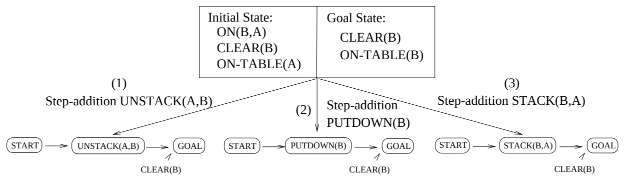

Figure 1: Three competing refinement candidates for achieving the goal *Clear(B)*.

and A_∞. The effects of A_0 correspond to the initial state, and the preconditions of A_∞ correspond to the goal state. In each planning cycle, a goal is removed from the agenda and an existing or new action is chosen to assert the goal. After an action is selected, the necessary ordering, casual link and codesignation constraints are added to $\mathcal{O}$, $\mathcal{L}$, and $\mathcal{B}$. If a *new* action was selected, the action's preconditions are added to the agenda. UCPOP then checks for threats and resolves any found by adding an additional ordering constraint. UCPOP is called recursively until the agenda is empty. On termination, UCPOP uses the constraints found in $\mathcal{O}$ to determine a total ordering of the actions in $\mathcal{S}$, and returns this as the final solution.

Learning Control For Planning

SCOPE learns search-control rules for planning decisions that might lead to a failing search path (i.e. might be backtracked upon). Figure 1 illustrates an example from the blocksworld domain where control knowledge could be useful. Here, there are three possible refinement candidates for adding a new action to achieve the goal *Clear(B)*. For each set of refinement candidates, SCOPE learns control rules in the form of selection rules that define when each refinement should be applied. A single selection rule consists of a conjunction of conditions that must all evaluate to true for the refinement candidate to be used. For example, shown next is a selection rule for the first candidate (from Figure 1) which contains several control conditions.

Select operator `Unstack(?X,?Y)` **to establish** `goal(`$Clear(?Y)$,s_1`)`
  If `exists-operator(`s_2`)` $\wedge$ `establishes(`s_2,$On(?X,?Y)$`)` $\wedge$
    `possibly-before(`s_2,s_1`)`.

This rule states that `Unstack(?X,?Y)` should be selected to add *Clear(?Y)* only when there is an existing action s_2 that adds *On(?X,?Y)* and s_2 can be ordered before the action s_1, which requires *Clear(?Y)*. Learned control information is incorporated into the planner so that attempts to select an inappropriate refinement will immediately fail.

The Prolog programming language provides an excellent framework for learning control rules. Search algorithms can be implemented in Prolog in such a way that allows control information to be easily incorporated in the form of clause-selection rules (Cohen 1990). These rules help avoid inappropriate clause applications, thereby reducing backtracking. A version of the UCPOP partial-order planning algorithm has been implemented as a Prolog program.[1] Planning decision points are represented in this program as clause-selection problems (i.e. each refinement candidate is formulated as a separate clause). SCOPE is then used to learn refinement-selection rules which are incorporated into the original planning program in the form of clause-selection heuristics.

The SCOPE Learning System

SCOPE is based on the DOLPHIN learning system (Zelle & Mooney 1993), which optimizes logic programs by learning clause-selection rules. DOLPHIN has been shown successful at improving program performance in several domains, including planning domains which employed a simple state-based planner. DOLPHIN, however, has little success improving the performance of a partial-order planner due to the higher complexity of the planning search space. In particular, DOLPHIN's simple control rule format lacked the expressibility necessary to describe complicated planning situations. SCOPE has greatly expanded upon the DOLPHIN algorithm to be effective on more complex planners.

The input to SCOPE is a planning program and a set of training examples. SCOPE uses the examples to induce a set of control heuristics which are then incorporated into the original planner. SCOPE's algorithm has three main phases, which are presented in the next few sections. A more detailed description can be found in (Estlin 1996).

Example Analysis

In the example analysis phase, two outputs are produced: a set of selection-decision examples and a set of *generalized* proof trees. Selection-decision examples record successful and unsuccessful applications of plan refinements. Generalized proofs provide a background context that explains the success of all correct planning decisions. These two pieces of information are used in the next phase to build control rules.

Selection-decision examples are produced using the following procedure. First, training examples are solved using the existing planner. A trace of the planning decision process used to solve each example is stored in a proof tree, where the root represents the top-level planning goal and nodes correspond to different planning procedure calls. These proofs are

[1]Our Prolog planner performs comparably to the standard LISP implementation of UCPOP on the sample problem sets used to test the learning algorithm.

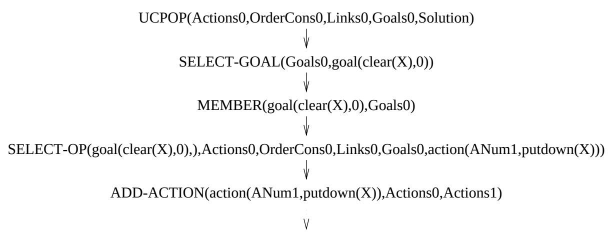
UCPOP(Actions0,OrderCons0,Links0,Goals0,Solution)
SELECT-GOAL(Goals0,goal(clear(X),0))
MEMBER(goal(clear(X),0),Goals0)
SELECT-OP(goal(clear(X),0),),Actions0,OrderCons0,Links0,Goals0,action(ANum1,putdown(X)))
ADD-ACTION(action(ANum1,putdown(X)),Actions0,Actions1)

Figure 2: Top Portion of a Generalized Proof Tree

then used to extract examples of correct and incorrect refinement-selection decisions. A "selection decision" is a planning subgoal that was solved correctly by applying a particular plan refinement, such as adding a new action. As an example, consider the planning problem introduced in Figure 1. The planning subgoal represented by this figure is shown below:[2]

For $\mathcal{S}$ = (0:Start,G:Goal),
$\mathcal{O}$ = (0 $\prec$ G),
$\mathcal{L}$ = $\emptyset$,
agenda = (*Clear(B)*,G),(*On-Table(B)*,G),
Select operator ?OP **to establish** `goal`(*Clear(B)*,G)

Selection decisions are collected for all competing plan refinements. Refinements are considered "competing" if they can be applied in identical planning decisions, such as the three refinement candidates shown in Figure 1. A correct decision for a particular refinement is an application of that refinement found on a solution path. An incorrect decision is a refinement application that was tried and subsequently backtracked over. The subgoal shown above would be identified as a positive selection decision for candidate 2 (adding `Putdown(A)`), and would also be classified as a negative selection decision for candidates 1 and 3 (adding `Unstack(B,A)` or `Stack(A,B)`). Any given training problem may produce numerous positive and negative examples of refinement selection decisions.

The second output of the example analysis phase is a set of generalized proof trees. Standard EBG techniques are used to generalize each training example proof tree. The goal of this generalization is to remove proof elements that are dependent on the specific example facts while maintaining the overall proof structure. Generalized proof information is used later to explain new planning situations. The top portion of a generalized proof tree is shown in Figure 2. This proof was extracted from the solution trace of the problem introduced in Figure 1. The generalized proof of this example provides a context which "explains" the success of correct decisions.

[2]Binding constraints in our system are maintained through Prolog, therefore, the set of binding constraints, $\mathcal{B}$, is not explicitly represented in planning subgoals.

Control Rule Induction

The goal of the induction phase is to produce an operational definition of when it is useful to apply a refinement candidate. Given a candidate, C, we desire a definition of the concept "subgoals for which C is useful". In the blocksworld domain, such a definition is learned for each of the candidates shown in Figure 1. In this context, control rule learning can be viewed as relational concept learning. SCOPE employs a version of the FOIL algorithm (Quinlan 1990) to learn control rules through induction. FOIL has proven efficient in a number of domains, and has a "most general" bias which tends to produce simple definitions. Such a bias is important for learning rules with a low match cost, which helps avoid the utility problem. FOIL is also relatively easy to bias with prior knowledge (Pazzani & Kibler 1992). In our case, we can utilize the information contained in the generalized proof trees of planning solution traces.

FOIL attempts to learn a concept definition in terms of a given set of background predicates. This definition is composed of a set of Horn clauses that cover all of the positive examples of a concept, and none of the negatives. The selection-decision examples collected in the example analysis phase provide the sets of positive and negative examples for each refinement candidate.

```
Initialization
    Definition := null
    Remaining := all positive examples
While Remaining is not empty
        Find a clause, C, that covers some examples in
            Remaining, but no negative examples.
        Remove examples covered by C from Remaining.
        Add C to Definition.
```

FOIL's basic algorithm is shown above. The "find a clause" step is implemented by a general-to-specific hill-climbing search. FOIL adds antecedents to the developing clause one at a time. At each step FOIL evaluates all literals that might be added and selects the one which maximizes an information-based gain heuristic. SCOPE uses an intensional version of FOIL where background predicates can be defined in Prolog instead of requiring an extensional representation.

One drawback to FOIL is that the hill-climbing search for a good antecedent can easily explode, especially when there are numerous background predicates with large numbers of arguments. When selecting each new clause antecedent, FOIL tries *all* possible variable combinations for *all* predicates before making its choice. This search grows *exponentially* as the number of predicate arguments increases. SCOPE circumvents this search problem by utilizing the generalized proofs of training examples. By examining the proof trees, SCOPE identifies a small set of potential literals that could be added as antecedents to the current clause definition. Specifically, all "operational" predicates contained in a proof tree are considered. These are usually low-level predicates that have been classified as "easy to evaluate" within the problem domain. Literals are added to rules in a way that utilizes variable connections already established in the proof tree. This approach nicely focuses the FOIL search by only considering literals (and variable combinations) that were found useful in solving the training examples.

SCOPE employs the same general covering algorithm as FOIL but modifies the clause construction step. Clauses are successively specialized by considering how their target refinements were used in solving training examples. Suppose we are learning a definition for when each of the refinements in Figure 1 should be applied. The program predicate representing this type of refinement is select-op, which inputs several arguments including the unachieved goal and outputs the selected operator. The full predicate head is shown below.

```
select-op(Goal,Steps,OrderCons,Links,Agenda,ReturnOp)
```

For each refinement candidate, SCOPE begins with the most general definition possible. For instance, the most general definition covering candidate 1's selection examples is the following; call this clause C.

```
select-op(Goal,Steps,OrderCons,Links,Agenda,unstack(A,B)) :-
    TRUE
```

This overly general definition covers *all* positive examples and *all* negative examples of when to apply candidate 1, since it will always evaluate to true. C can be specialized by adding antecedents to its body. This is done by unifying C's head with a (generalized) proof subgoal that was solved by applying candidate 1 and then adding an operational literal from the same proof tree which shares some variables with the subgoal. For example, one possible specialization of the above clause is shown below.

```
select-op((clear(B),S1),Steps0,OrderCons,Links,Agenda,unstack(A,B)) :-
    establishes(on(B,A),Steps1,S2).
```

Here, a proof tree literal has been added which checks if there is an existing plan step that establishes the goal *On(B,A)*. Variables in a potential antecedent can be connected with the existing rule head in several ways. First, by unifying a rule head with a generalized subgoal, variables in the rule head become unified with variables existing in a proof tree. All operational literals in that proof that share variables with the generalized subgoal are tested as possible antecedents. A second way variable connections are established is through the standard FOIL technique of unifying variables of the same type. For example, the rule shown above has an antecedent with an unbound input, `Steps1`,

```
select-op((clear(B),S1),Steps,OrderCons,Links,Agenda,unstack(A,B)) :-
    find-init-state(Steps,Init),
    member(on(A,B),Init),
    not(member((on(B,C),S1),Agenda),member(on-table(B),Init)).

select-op((clear(A),G),Steps,OrderCons,Links,Agenda,putdown(A)) :-
    not(member((on(A,B),G),Agenda)).

select-op((clear(A),S1),Steps,OrderCons,Links,Agenda,putdown(A)) :-
    member((on-table(A),S2),Agenda),
    not(establishes(on-table(A),S3)).
```

Figure 3: Learned control rules for two refinements

which does not match any other variables in the clause. SCOPE can modify the rule, as shown below, so that the `Steps1` is unified with a term of the same type from the rule head, `Steps0`.

```
select-op((clear(B),S1),Steps0,OrderCons,Links,Agenda,unstack(A,B)) :-
    establishes(on(B,A),Steps0,S2).
```

SCOPE considers all such specializations of a rule and selects the one which maximizes FOIL's information-gain heuristic.

SCOPE also considers several other types of control rule antecedents during induction. Besides pulling literals directly from generalized proof trees, SCOPE can use negated proof literals, determinate literals (Muggleton 1992), variable codesignation constraints, and relational clichés (Silverstein & Pazzani 1991). Incorporating different antecedent types helps SCOPE learn expressive control rules that can describe partial-order planning situations.

Program Specialization Phase

Once refinement selection rules have been learned, they are passed to the program specialization phase which adds this control information into the original planner. The basic approach is to guard each refinement with the selection information. This forces a refinement application to fail quickly on subgoals to which the refinement should not be applied. Figure 3 shows several learned rules for the first two refinement candidates (from Figure 1). The first rule allows `unstack(A,B)` to be applied only when `A` is found to be on `B` initially, and `stack(B,C)` should not be selected instead. The second and third rule allow `putdown(A)` to be applied only when `A` should be placed on the table and not stacked on another block.

Experimental Evaluation

The blocksworld and logistics transportation domains were used to test the SCOPE learning system. In the logistics domain (Veloso 1992), packages must be delivered to different locations in several cities. A test set of 100 independently generated problems was used to evaluate performance in both domains. SCOPE was

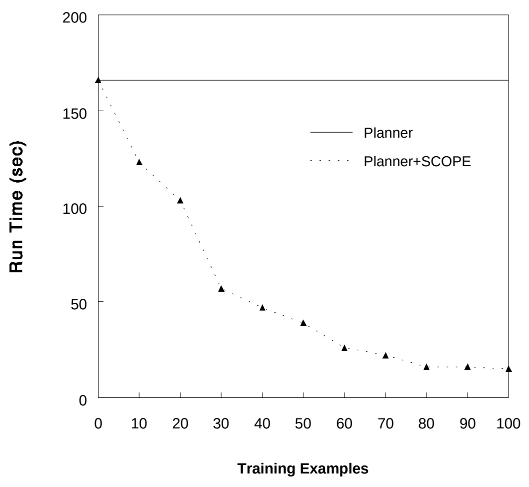

Figure 4: Performance in Blocksworld

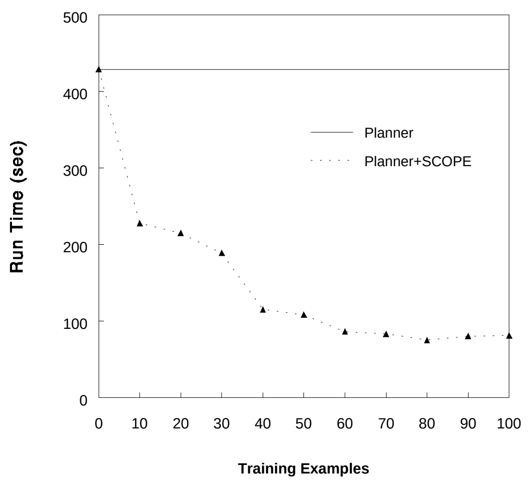

Figure 5: Performance in Logistics

trained on separate example sets of increasing size. Ten trials were run for each training set size, after which results were averaged. Training and test problems were produced for both domains by generating random initial and final states. In blocksworld, problems contained two to six blocks and one to four goals. Logistics problems contained up to two packages and three cities, and one or two goals. No time limit was imposed on planning, but a uniform depth bound on the plan length was used during testing.

For each trial, SCOPE learned control rules from the given training set and produced a modified planner. Since SCOPE only specializes decisions in the original planner, the new planning program is guaranteed to be sound with respect to the original one. Unfortunately, the new planner is not guaranteed to be complete. Some control rules could be too specialized and thus the new planner may not solve all problems solvable by the original planner. In order to guarantee completeness, a strategy used by Cohen (1990) is adopted. If the final planner fails to find a solution to a test problem, the initial planning program is used to solve the problem. When this situation occurs in testing, both the failure time for the new planner and the solution time for the original planner are included in the total solution time for that problem. In the results shown next, the new planner generated by SCOPE was typically able to solve 95% of the test examples.

Figures 4 and 5 present the experimental results. The times shown represent the number of seconds required to solve the problems in the test sets after SCOPE was trained on a given number of examples. In both domains, the SCOPE consistently produced a more efficient planner and significantly decreased solution times on the test sets. In the blocksworld, SCOPE produced modified planning programs that were an average of 11.3 times faster than the original planner. For the logistics domain, SCOPE produced programs that were an average of 5.3 times faster. These results indicate that SCOPE can significantly improve the performance of a partial-order planner.

Related Work

A closely related system to SCOPE is UCPOP+EBL (Kambhampati et al., 1996), which also learns control rules for UCPOP, but uses a purely explanation-based approach. UCPOP+EBL employs standard EBL to acquire control rules in response to past planning failures. This system has been shown to improve planning performance in several domains, including blocksworld. To compare the two systems, we replicated an experiment used by Kambhampati et al. (1996). Problems were randomly generated from a version of the blocksworld domain that contained between three to six blocks and three to four goals.[3] SCOPE was trained on a set of 100 problems. The test set also contained 100 problems and a CPU time limit of 120 seconds was imposed during testing. The results are shown below.

System	*Orig Time*	*Final Time*	*Speedup*	*Orig %Sol*	*Final %Sol*
UCPOP+EBL	7872	5350	1.47X	51%	69%
SCOPE	5312	1857	2.86X	59%	94%

Both systems were able to increase the number of test problems solved, however, SCOPE had a much higher success rate. Overall, SCOPE achieved a better speedup ratio, producing a more efficient planner.

[3]In order to replicate the experiments of Kambhampati et al.(1996), the blocksworld domain theory used for these tests slightly differed from the one used for the experiments presented previously. Both domains employed similar predicates however the previous domain definition consists of four operators while the domain used here has only two.

By combining EBL with induction, Scope was able to learn better planning control heuristics than EBL did alone. These results are particularly significant since UCPOP+EBL utilizes additional domain axioms which were not provided to Scope.

Most other related learning systems have been evaluated on different planning algorithms, thus system comparisons are difficult. The Hamlet system (Borrajo & Veloso 1994) learns control knowledge for the nonlinear planner underlying Prodigy4.0. Hamlet acquires rules by explaining past planning decisions and then incrementally refining them. Since, Prodigy4.0 is not a partial-order planner it is difficult to directly compare Hamlet and Scope. When making a rough comparison to the results reported in Borrajo & Veloso (1994b) , Scope achieves a greater speedup factor in blocksworld (11.3 vs 1.8) and in the logistics domain (5.3 vs 1.8).

Future Work

There are several issues we hope to address in future research. First, replacing Foil's information-gain metric for picking literals with a metric that more directly measures rule utility could further improve planning performance. Second, Scope should be tested on more complex domains which contain conditional effects, universal quantification, and other more-expressive planning constructs. Finally, we plan to examine ways of using Scope to improve plan quality as well as planner efficiency. Scope could be modified to collect positive control examples only from high-quality solutions so that control rules are focused on quality issues as well as speedup.

Conclusion

Scope provides a new mechanism for learning control information in planning systems. Simple, high-utility rules are learned by inducing concept definitions of when to apply plan refinements. Explanation-based generalization aids the inductive search by focusing it towards the best pieces of background information. Unlike most approaches which are limited to total-order planners, Scope can learn control rules for the newer, more effective partial-order planners. In both the blocksworld and logistics domains, Scope significantly improved planner performance; Scope also outperformed a competing method based only on EBL.

References

Barrett, A., and Weld, D. 1994. Partial order planning: Evaluating possible efficiency gains. *Artificial Intelligence* 67:71–112.

Bhatnagar, N., and Mostow, J. 1994. On-line learning from search failure. *Machine Learning* 15:69–117.

Borrajo, D., and Veloso, M. 1994. Incremental learning of control knowledge for nonlinear problem solving. In *Proc. of ECML-94*, 64–82.

Cohen, W. W. 1990. Learning approximate control rules of high utility. In *Proc. of ML-90*, 268–276.

DeJong, G. F., and Mooney, R. J. 1986. Explanation-based learning: An alternative view. *Machine Learning* 1(2):145–176.

Estlin, T. A. 1996. Integrating explanation-based and inductive learning techniques to acquire search-control for planning. Technical report, Dept. of Computer Sciences, University of Texas, Austin, TX. Forthcoming. URL: http://net.cs.utexas.edu/ml/

Kambhampati, S.; Katukam, S.; and Qu, Y. 1996. Failure driven search control for partial order planners: An explanation based approach. *Artificial Intelligence.* Forthcoming.

Langley, P., and Allen, J. 1991. The acquisition of human planning expertise. In *Proc. of ML-91*, 80–84.

Leckie, C., and Zuckerman, I. 1993. An inductive approach to learning search control rules for planning. In *Proc. of IJCAI-93*, 1100–1105.

Minton, S.; Drummond, M.; Bresina, J. L.; and Phillips, A. B. 1992. Total order vs. partial order planning: Factors influencing performance. In *Proc. of the 3rd Int. Conf. on Principles of Knowledge Rep. and Reasoning*, 83–92.

Minton, S. 1989. Explanation-based learning: A problem solving perspective. *Artificial Intelligence* 40:63–118.

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

Muggleton, S. H., ed. 1992. *Inductive Logic Programming.* New York, NY: Academic Press.

Pazzani, M., and Kibler, D. 1992. The utility of background knowledge in inductive learning. *Machine Learning* 9:57–94.

Penberthy, J., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. of the 3rd Int. Conf. on Principles of Knowledge Rep. and Reasoning*, 113–114.

Quinlan, J. 1990. Learning logical definitions from relations. *Machine Learning* 5(3):239–266.

Silverstein, G., and Pazzani, M. J. 1991. Relational clichés: Constraining constructive induction during relational learning. In *Proc. of ML-91*, 203–207.

Veloso, M. M. 1992. *Learning by Analogical Reasoning in General Problem Solving.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University.

Zelle, J. M., and Mooney, R. J. 1993. Combining FOIL and EBG to speed-up logic programs. In *Proc. of IJCAI-93*, 1106–1111.