Learning Declarative Control Rules for Constraint-Based Planning

Yi-Cheng Huang YCHUANG@CS.CORNELL.EDU
Bart Selman SELMAN@CS.CORNELL.EDU
Department of Computer Science, Cornell University, Ithaca, NY 14850 USA

Henry Kautz KAUTZ@CS.WASHINGTON.EDU
Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195 USA

Abstract

Despite the long history of research in using machine learning to speed-up state-space planning, the techniques that have been developed are not yet in widespread use in practical planning systems. One limiting factor is that traditional domain-independent planning systems scale so poorly that extensive learned control knowledge is required to raise their performance to an acceptable level. Therefore work in this area has focused on learning large numbers control rules that are specific to the details of the underlying planning algorithms, which can be extremely costly. In recent years a new generation of planning systems with much improved speed and scalability has become available. These systems formulate planning as solving a large constraint satisfaction problem. This formulation opens up the possibility that domain-specific control knowledge can be added to the planner in a purely declarative manner via a set of additional constraints.

In this paper we present the first positive results on automatically acquiring such high-level, declarative constraints using machine learning techniques. In particular, we will show that a new heuristic method for generating training examples together with a rule induction algorithm can learn useful control rules in a variety of domains. Only a small number of rules are needed to reduce solution times by two orders of magnitude or more on larger problems, training times are short, and the learned rules can be exported to other planning systems.

1. Introduction

Deterministic state-space planning is a hard combinatorial problem that arises in tasks such as robot control, software verification, and logistics scheduling. Although in general planning is PSPACE-complete (Bylander, 1991), for particular domains efficient algorithms — *i.e.*, polynomial or at worst exponential with a very low exponent — may exist for finding exact or approximate solutions. Research in machine learning have long studied the problem of automatically creating efficient planners by learning domain-specific rules or cases to control a general search engine (Minton, 1988; Carbonell, Knoblock, & Minton, 1990; Veloso, 1992; Etzioni, 1993; Bhatnagar & Mostow, 1994; Kambhampati, Katukam, & Qu, 1996; Borrajo & Veloso, 1997; Aler, Borrajo, & Isasi, 1998; Leckie & Zukerman, 1998; *etc.*). However, the successful practical application of machine learning techniques has been limited by at least two factors: First, traditional domain-independent planning systems (*e.g.*, PRODIGY, SOAR, NONLIN, UCPOP) scale so poorly that extensive learned control knowledge is required to raise their performance to an acceptable level. Second, previous work has focused on learning control rules that are specific to the details of the underlying general planner. This approach leads to the need to learn and manage large numbers of rules, which can be extremely costly in terms of time and the number of training examples. Furthermore, the learned domain-specific rules cannot be reused by other planners, nor can the learning module itself be ported to other systems without extensive modifications.

In recent years a new generation of planning systems with much improved speed and scalability has become available (Weld, 1999). These systems formulate planning as solving a large constraint satisfaction problem: Graphplan (Blum & Furst, 1995) and its descendents encode a CSP in a data structure called a plan graph, while SATPLAN and its descendents (Kautz & Selman, 1992; 1996; 1999) explicitly convert planning problems into Boolean satisfiability. The constraint-based formulation opens up the possibility that domain-specific control knowledge can be added to the planner in a purely declarative manner via a set of additional constraints. These new constraints do not make explicit reference to the workings of the underlying constraint satisfaction algorithm; like the constraints that define the original problem instance, they only refer to the solution (plan) space. In earlier work (Kautz & Selman, 1998; Huang, Selman, & Kautz, 1999), we showed that the same set of *hand-coded* declarative constraints can provide dramatic reductions in solution times of radically different constraint satisfaction algorithms (*e.g.*, local search or systematic search), and that a large subset of the constraints can be employed by fundamentally different planning architectures (*e.g.*, the forward-chaining planner TLPlan (Bacchus & Kabanza, 2000)). The next natural question to ask is whether this kind of declarative control knowledge can be learned.

In this paper we present our initial positive results on automatically acquiring such constraints using machine learning techniques. The training set consists of a small number of planning problems (in the experiments here, 10 or fewer) together with their optimal solutions. We will describe how positive and negative examples of the target concepts used by the control rules are heuristically extracted from the input data. The rules are generated by an inductive logic programming approach based on the FOIL algorithm (Quinlan, 1990; 1996); unlike much work in inductive logic programming, however, explicit background knowledge in the form of defined predicates is not supplied to the system. Experimental evaluation on five different benchmarks domains from a recent planning competition is quite promising: training time is short (on the order of a minute), and the system learned small sets of high-quality rules. Adding these rules to our constraint-based planner reduced solution times by two orders of magnitude on large problems while maintaining or improving plan quality.

In addition to the earlier work mentioned above, there is other recent work on speed-up learning which, like ours, combines aspects of supervised learning and rule induction. The systems of Khardon (1999) and of Martin and Geffner (2000) try to learn very large sets of production-style rules that replace, rather than improve, a search engine, and require thousands of training examples and long training times. The work by Estlin and Mooney (1996) differs from ours in that it requires the user to explicitly supply background knowledge to the learner in the form of additional predicates and "relational clichés" (Silverstein & Pazzani, 1991). Recently Kambhampati (1999) has shown how explanation based learning (EBL) techniques can be applied to Graphplan, but does not attempt to learn high-level, declarative rules.

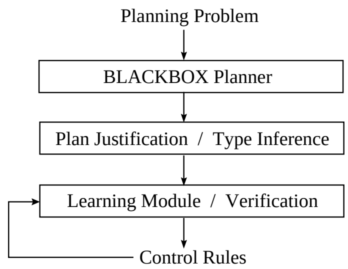

Figure 1. The basic learning framework.

We will illustrate the details of the system using examples from a logistics planning domain (Veloso, 1992) which has appeared as a benchmark in most recent work in planning. In brief, the task in this domain is to move a set of packages from various initial locations to various goal locations. Packages can be moved between locations within a city by truck. Airplanes can transport packages between airports in different cities. The basic actions are loading and unloading packages from vehicles, driving trucks, and flying airplanes. In the formulation used by the constraint-based planners considered here, all actions require one time unit for execution, and any number of non-conflicting actions may occur at the same time step. The number of time steps in a plan is its parallel length, and the number of actions is its sequential length. For example, a plan of parallel length 10 with 8 actions occurring at each time step would have sequential length 80. A plan can be deemed optimal in terms of its parallel or sequential length, or some combined measure; in our work, the optimality criteria is to minimize parallel length, and then to minimize sequential length.

2. Learning Framework

Our general learning framework is shown in Figure 1. The process begins by presenting the planning module with a problem instance of small to moderate size from the given planning domain. We use the Blackbox planner (Kautz & Selman, 1999), which combines both the Graphplan and SATPLAN systems mentioned above. Blackbox generates a plan of optimal parallel length. The plan is then passed through a justification algorithm which minimizes its sequential length by removing sets of unnecessary actions (Fink & Yang, 1992).

The justified plan also includes a description of the complete state at each time step, which is easily computed by simulating execution of the plan from the initial state. Meanwhile, a type inference algorithm computes type information for all operators and objects in the domain (Fox & Long, 1998). The justified plan and type information are passed to the learning module.

The learning module uses the plan in two ways. First, any previously learned rules are verified against the plan, and inconsistent rules are discarded. Second, a set of positive and negative examples of target concepts are heuristically extracted from the plan, as will be described in detail below. This step crucially depends upon the fact that the plan is optimal or near-optimal. Those examples not covered by previous rules are used as training data by an inductive rule learning algorithm, which generates one or more new control rules. The type information inferred earlier improves the speed and quality of rule induction. (Note, however, that some induced rules may be incorrect, and thus the need for the verification step.) The process is then repeated for several problem instances, and the final set of learned rules is output in the form of a set of logical axioms, which can be used by either the original planner or a variety of other recent planning engines.

We next consider aspects of the system in more detail.

2.1 Target Concepts for Actions

For each action in the planning domain, the system learns two complementary concepts, *select action* and *reject action*. Each concept is defined by a set of logic programming-type rules in a simple temporal logic. Select rules indicate conditions under which the action must be performed, and reject rules indicate conditions under which it must not be performed.

Both kinds of rules can be divided into two categories according to the information upon which they rely (Huang, Selman, & Kautz, 1999). A *static* rule is one whose body depends only upon the initial and goal states specified in the planning problem, but not upon the particular time step at which the action should be selected or rejected. Thus, a static rule either holds for all time steps in the problem instance or for none. A *dynamic* rule is one whose body also depends upon what is true at to the "current" time step. As described below, static and dynamic rules are learned separately.

Examples of different kinds of rules from the logistics domain are:

static reject: Do not unload a package from an airplane at an airport if that airport is not in the package's goal city.

dynamic reject: Do not move an airplane if it currently contains a package which needs to be unloaded at that city.

dynamic select: Unload a package from a truck at the package's goal location.

The logistics domain does not happen to contain any static select rules; these would arise in domains where some particular action must be repeatedly performed at every step of the plan in order to achieve the goal. (Note that only actions whose preconditions hold in every state can appear in static select rules.)

2.2 Heuristics for Identifying Training Examples

A traditional EBL approach would find examples of the select and reject concepts by examining a trace of the planner or by re-deriving the solution to a solved instance. In our approach, by contrast, the training examples are heuristically derived from the solved problem instance. The heuristic is based on the notion that there is a good chance that the particular actions that appear in an *optimal* solution *must* be selected, and those that do not appear *must* be rejected. (In other words, we are using learning to operationalize the optimality criteria of the planner and justification module.) This method of generating examples is obviously fairly noisy, and we will describe the techniques we use to minimize the effect of incorrectly labeled examples.

In particular, assume that the plan $\mathcal{P}$ is found by the planner for a given problem. We will say an instantiated action is (1) *real* at time i if it appears in $\mathcal{P}$ at time i; (2) *virtual* at time i if all of its preconditions hold at time i but it does not belong to plan $\mathcal{P}$ at time i; and (3) *mutex virtual* at time i if it is virtual and there exists at least one real action that is mutually exclusive with it at time i. (Two actions are mutually exclusive if they cannot occur at the same time, even if their preconditions both hold; for example, loading an airplane is mutually exclusive with flying that airplane.)

Each element of the set of all actions instantiated at all times (up to the length of the solution) is categorized according to this scheme. Then the positive and negative examples for learning static and dynamic select and reject rules for each action are chosen according the scheme specified in Table 1. Note that the training set for learning dynamic rules is a subset of the

Table 1. Types of actions used in training sets for the static and dynamic varieties of the select and reject rules.

	select rule		*reject* rule	
	positive example	negative example	positive example	negative example
static	real	virtual	virtual	real
dynamic	real	mutex virtual	mutex virtual	real

Table 2. Outline of the rule induction algorithm, based on Quilan's FOIL.

$Rules = \emptyset$
$Remaining$ = examples for the target concept R
while $Remaining \neq \emptyset$
  $rule = R \leftarrow$ **null**
  while *rule* covers negative examples
    Add a literal L to the body of
     rule that maximizes gain
  Remove from *Remaining* examples that
   are covered by *rule*
  Add *rule* to *Rules*

training set used for learning static rules, because it contains fewer examples based on actions that do not appear in the plan.

Why is the training set for dynamic rules restricted in this manner? In short, to reduce the amount of noise it contains. In general, learning good dynamic rules is more difficult than learning good static rules, because there are more dynamic rules that are consistent with the data. Furthermore, examples based on actions that do not occur are clearly less reliable than ones based on actions that do occur. One would like to concentrate on examples of non-occurring actions that are relevant to the problem instance. Mutex virtual actions tend to be more relevant than others because their preconditions and effects overlap with those of actions that *do* occur.

2.3 Rule Induction

Control rules are generated from the training examples by a greedy general-to-specific search in the space of restricted temporal logic programs, as shown in Table 2, using an algorithm based on the FOIL procedure (Quinlan, 1990; 1996).

The simple temporal logic programs we consider are constructed as follows. There are three kinds of predicates: *static predicates*, which refer to facts whose truth cannot be changed by any action (*e.g.*, the predicate "in-city" used to assert that a particular location is part of a particular city); *fluent predicates*, used for facts whose truth varies over time (*e.g.*, the predicate "at" which relates a movable object and a location); and *action predicates*, used for parameterized actions (*e.g.*, "FLY-AIRPLANE"). There is a single modal operator "goal", used to assert that its argument is one of the specified goals of the planning problem. A *literal* is an expression of the following form or its negation, where P is a predicate, F is a fluent predicate, and each X_i is a variable:

$$X_i = X_j, \quad P(X_1, ..., X_n), \quad goal(F(X_1, ..., X_n))$$

A *rule* contains a distinguished literal, its head, and a set of literals that make up its body. An *instantiation* of a rule is created by substituting constants for all of its variables. A rule is *consistent* with a justified plan iff for all of its instantiations, the head is true at each time step at which all literals in the body are true. Note that goal literals and static predicate literals have the same truth value at all time steps in a justified plan; in particular, "goal" here refers only to *final* goals, not intermediate goals. This language can be enriched in various ways, for example by including other modal operators such as "next" or "eventually", or by allowing a rule to contain explicit constants; we will explore these extensions in future research.

The head of a select control rule must be a positive action predicate literal, and the head of a reject rule must be a negative action predicate literal. As we have mentioned, static and dynamic rules are learned separately. The two kinds of rules are distinguished by the kinds of literals that may appear in their bodies:

> **Static rules** may contain positive or negative equality literals, static predicate literals, and goal literals. Note that the truth of each of these kinds of literals does not vary across time steps.
>
> **Dynamic rules** may contain all of the above, but must also contain at least one negative or positive (non-goal) fluent literal. Note that the when checking the consistency of a dynamic rule, the (non-goal) fluent literals are evaluated at the same time step used to evaluate the rule's head.

As in FOIL, the literals added to the rule at each step are chosen according the following criteria:

- the literal with greatest gain if this gain is close to the maximum possible; otherwise

- all determinate literals found;[1] otherwise
- the literal with highest positive gain; otherwise
- the first literal considered that introduces a new variable.

We experimented with a number of different definitions of the *gain* function. We obtained the best performance using the "Laplace estimate" used in a number of recent learning systems (CN2 (Clark & Boswell, 1991), *etc.*):

$$\text{Gain}(r) = \frac{p+1}{p+n+2} ,$$

where r is the candidate rule, p is the number of positive examples covered by r, and n is the number of negative examples covered by r.

Because the Laplace estimate penalizes rules with low coverage, it proved to be more robust against noise in the training set. Noise is also handled, as noted earlier, by pruning learned rules that turn out to be inconsistent with future examples of solved planning problems given to the learner.

We mentioned above that during plan justification type inference on all objects and predicates is performed using the algorithm from TIM (Fox & Long, 1998). This inferred type information serves two purposes: First, types are used to reduce the number of rules considered in the learning procedure. For example, when considering candidate equality literals, the arguments of the equality literal must be of the same type. Second, when adding a literal that introduces new variables, inferred types are used to correctly find bindings for every new variable. For instance, when adding a literal (at o l) to a rule, where the types of o and l associated to the literal are "package"and "location" respectively and o is a new variable, only objects with type "package" will be considered as bindings for variable o. In other words, objects with type "truck" or "airplane" that can also be bound to the first argument of predicate (at o l) will not be considered. This use of type information is important to correctly acquire control rules.

It is common in inductive logic programming for the user to provide explicit background knowledge to the system in the form of additional relations or axioms (Quinlan, 1990). For example, the user might write a definition for a predicate that holds of "packages that need to be moved". One might argue, however, that manually defining good background knowledge is as difficult as defining good control rules. In our system, by contrast, no additional background knowledge is input by the user. The categorization of the different kinds of predicates, the state information that appears in the justified plan, and the inferred type information can all be considered to be kinds of background knowledge that is automatically acquired by the system.

[1] A *determinate* literal is one that introduces new variables so that there is exactly one binding for each positive example and at most one binding for each negative example in the partially-constructed rule (Muggleton & Feng, 1992; Quinlan, 1996).

Table 3. A simple logistics problem and its solution. There are three cities (A, B, and C), each containing two locations, an airport and a postoffice (*e.g.*, apt-A and po-A). In each city there is a truck (trk-A, trk-B, trk-C), and there is one airplane (pln). There are two packages (o1, o2) to be delivered.

Initial:	(at o1 apt-A), (at o2 apt-B),
	(at pln apt-A), (at trk-C apt-C),
	(in-city apt-A A), (in-city po-A A), ...
Goal:	(at o1 po-C), (at o2 po-C)
Plan:	1 LOAD-AIRPLANE (o1 pln apt-A)
	2 FLY-AIRPLANE (pln apt-A apt-B)
	3 LOAD-AIRPLANE (o2 pln apt-B)
	4 FLY-AIRPLANE (pln apt-B apt-C)
	5 UNLOAD-AIRPLANE (o1 pln apt-C)
	5 UNLOAD-AIRPLANE (o2 pln apt-C)
	6 LOAD-TRUCK (trk-C o1 apt-C)
	6 LOAD-TRUCK (trk-C o2 apt-C)
	7 DRIVE-TRUCK (trk-C apt-C po-C)
	8 UNLOAD-TRUCK (trk-C o1 po-C)
	8 UNLOAD-TRUCK (trk-C o2 po-C)

2.4 Using Learned Rules

As reported in Huang, Selman, and Kautz (1999), we extended the PDDL planning input language (McDermott, 1998) of the Blackbox planner to allow control knowledge to be specified using temporal logic formulas. Thus the rules created by the learning module can be directly fed back to the planner. Blackbox uses static reject rules to prune the Boolean SAT encodings it creates of planning problems. All other kinds of rules are converted in propositional clauses that are added to the encoding. The general effect of the added clauses is to make the encoded problem easier to solve by increasing the power of the constraint propagation routines used by the system's SAT engines.

3. An Example

We will now step through an example of learning a control rule for the logistics domain. Consider a problem instance and the (justified) plan found by the Blackbox planner as shown in Table 3. Suppose we are learning static reject rules for the action "UNLOAD-AIRPLANE

Table 4. Learning static reject rules for the action "UNLOAD-AIRPLANE (o p a)". The types of the variables are: o - package; p - airplane; a - airport; c - city; and l - location.

	time	o	p	a	c	l
+	2	o1	pln	apt-A	A	po-C
+	3	o1	pln	apt-B	B	po-C
+	4	o1	pln	apt-B	B	po-C
+	4	o2	pln	apt-B	B	po-C
-	5	o1	pln	apt-C	C	po-C
-	5	o2	pln	apt-C	C	po-C

(o p a)". (See the caption of Table 4 for the types associated with each variable in this example.) According to the heuristic for generating training examples for static reject rules, UNLOAD-AIRPLANE (o1 pln apt-A) at time 2 is a positive example because all of its preconditions hold at time 2 (such as (in o1 pln) and (at pln apt-A)) but it does not appear in the plan. On the other hand, UNLOAD-AIRPLANE (o1 pln apt-C) at time 5 is a negative example because it appears in the plan at time 5. The complete set of positive and negative examples are shown in the first five columns in Table 4.

Following the learning procedure outlined above, two determinate literals (in-city a c) and (goal (at o l)) are first added to the rule, and they introduce two new variables c and l (see last two columns in Table 4).

In the next iteration, a literal ¬(in-city l c) is found to give the highest possible gain and is added to the rule. Now the rule covers only positive examples and none of negative examples. Therefore the procedure terminates with the rule:

¬ UNLOAD-AIRPLANE (o p a) ←
(in-city a c) ∧ (goal (at o l)) ∧ ¬(in-city l c)

It is worth noting that the above rule captures the concept of "an object that is not in its goal city". In other recent work on learning control knowledge for planning (Estlin & Mooney, 1996), this concept can only be learned by introducing hand-coded background knowledge.

4. Experimental Results

We have performed a preliminary empirical evaluation of our approach on a set of planning domains (logistics, grid, gripper, and mystery) from the 1998 AI Planning Systems Competition (AIPS98), as well as the tireworld domain from the PDDL (McDermott, 1998) and TLPlan (Bacchus & Kabanza, 2000) distributions. All experiments were run on a 300Mhz Sparc Ultra. As noted earlier, the logistics domain has become a particularly popular benchmark for recent work in planning.

Table 5. Learning time (in seconds) and number of rules acquired. Learning time includes time to both generate and verify rules. Mystery training problems are from AIPS98 competition, and Tireworld problems are from PDDL distribution. All other training problems are randomly generated.

domain	# training problems	learning time	# rules learned
grid	6	66.79	10
gripper	2	0.13	3
logistics	10	22.54	11
mystery	6	120.6	4
tireworld	4	1.23	17

Table 6. Blackbox without and with (c) learned control knowledge. Results are averaged over 10 runs and times are given in cpu seconds. Grid and tireworld problems are randomly generated. Logistics-d and logistics-e are from the Blackbox distribution. All other problems are from the AIPS98 competition.

problem	time step (# obj/goal)	Blackbox	Blackbox(c)
grid-a	13 (28/3)	20.99	4.81
grid-b	18 (29/1)	74	16.62
grid-c	23 (29/2)	2132	37.03
gripper-p03	15 (12/8)	> 7200	7.18
gripper-p04	19 (14/10)	> 7200	259.8
logistics-d	14 (36/9)	15.85	5.76
logistics-e	15 (40/10)	3522	290.9
logistics-p05	12 (43/4)	> 7200	10.58
logistics-p07	9 (60/6)	1522	32.34
mystery-p10	8 (77/1)	> 7200	47.25
mystery-p13	8 (83/2)	161	12.24
tireworld-a	24 (15/19)	5.8	3.65
tireworld-b	30 (17/23)	60.6	28.39

Table 5 summarizes the learning time and number of control rules acquired for each domain. Most of the training examples are randomly generated small instances, while a few are taken directly from the available distributions. In general our learning times are very short relative to other speed-up learning systems, which typically take several minutes to several hours to generate a good set of rules.

Table 6 summarizes the results of running Blackbox with and without the learned control rules on a set of larger and harder problems drawn from each domain. For each domain Blackbox's various parameters are set to optimize performance for the case where no learned rules are added. To give a very rough idea of the size of the each instance, the solution parallel length (time steps), number of objects (constants), and number of goals are indicated. It can be easily seen that Blackbox with learned control rules is significantly faster, often by two orders of magnitude. Several problems that could not be solved by original Blackbox were solved by Blackbox with learning.

We also discovered that learned domain knowledge could help the planner to find better quality solutions. For example, consider the problem logistics-p07 from the AIPS98 competition. During the competition, only the HSP planner (Bonet & Geffner, 1997) could find a plan as short as 112 actions. However, Blackbox with learned control knowledge is able to find a plan that takes 9 time steps with only 46 actions!

It is interesting to compare rules learned by our system with hand-coded ones. We consider the logistics planning domain and the hand-coded rules from TLPlan. We found that all static reject rules used in TLPlan are correctly acquired by our learning system. In addition, our system learns several useful dynamic select rules that were not used in TLPlan. For example, our system learns the dynamic select rule: "Unload a package from a truck at the package's destination". Because TLPlan is a purely sequential planner (unlike Blackbox, Graphplan, *etc.*) it cannot make use of rules that would select for a set of actions of equal priority to be executed simultaneously. (Instead, one would have to write a much more complex select rule that enforced some arbitrary temporal ordering between the unloads.) Our system did fail, however, to learn some of the dynamic reject rules used by TLPlan.

Unlike EBL approaches, the learned rules are not necessarily logical consequences of the domain (Mitchell, Keller, & Kedar-Cabelli, 1986; Dejong & Mooney, 1986). In particular, if the training set is too small, one could in principle learn rules that exclude all solutions to a particular problem instance, although this did not occur in our experiments.

5. Conclusions

In recent years research in state-space planning has been re-energized by a new generation of highly efficient constraint-based planners. We believe that the field of speed-up learning is poised to undergo a similar resurgence. In this paper we have presented the first positive results on acquiring and using declarative domain-specific control knowledge for constraint-based planning. Our approach blends aspects of explanation-based learning, supervised learned, and inductive logic programming. Our learning architecture is simple and modular, and initial empirical evaluation on established benchmarks has shown that control knowledge can be learned that is on par with that created by hand. Our current and future work includes a more careful and detailed empirical evaluation of the approach; an investigation of learning and using more expressive control rule languages; and a study of ways to create training problems that will most aid learning. In particular, we are investigating an *active learning* approach, in which the current set of learned control rules is used to influence the creation of the next training problem.

Acknowledgements

Part of the work reported in this paper was supported by a summer internship at AT&T Labs.

References

Aler, R., Borrajo, D., & Isasi, P. (1998). Genetic programming and deductive-inductive learning: A multistrategy approach. *Proceedings of the Fifteenth International Conference on Machine Learning* (pp. 10–18). Madison, WI: Morgan Kaufmann.

Bacchus, F. & Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, *116*.

Bhatnagar, N. & Mostow, J. (1994). On-line learning from search failures. *Machine Learning*, *15*, 69–117.

Blum, A. & Furst, M. L. (1995). Fast planning through planning graph analysis. *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (pp. 1636–1642). Montreal, Canada.

Bonet, B. & Geffner, H. (1997). A fast and robust action selection mechanism for planning. *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (pp. 714–719). Provide, RI.

Borrajo, D. & Veloso, M. M. (1997) Lazy incremental learning of control knowledge for efficiently obtaining quality plans. In David Aha (Ed.), *Lazy Learning*, Kluwer Academic Publishers.

Bylander, T. (1991). Complexity results for planning. *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence* (pp. 274–279), Sidney, Australia: Morgan Kaufmann.

Carbonell, J., Knoblock, C., & Minton, S. (1990). PRODIGY: An integrated architecture for planning and learning. In K. VanLeh (Ed.), *Architectures for Intelligence*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Clark, P. & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3(4)*, 261–283.

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

Estlin, T. A. & Mooney, R. J. (1996) Multi-strategy learning of search control for partial-order planning. *Proceedings of the Thirteen National Conference on Artificial Intelligence* (pp. 843–848). Portand, OR.

Etzioni, O. (1993). Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, *62(2)*, 255–302.

Fikes, R. E. & Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, *5(2)*, 189–208.

Fink, E. & Yang, Q. (1992). Formalizing plan justifications. *Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 9–14).

Fox, M. & Long, D. (1998). The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, *9*, 367–421.

Martin, M. & Geffner, H. (2000). Learning generalized policies in planning using concept languages. *Proceedings of the Seventh International Conference on Principles of Knowledge Representation and Reasoning*. Breckenridge, CO.

Gerevini, A. & Schubert, L. (1998). Inferring state constraints for domain-independent planning. *Proceedings of the Fifteen National Conference on Artificial Intelligence* (pp. 905–912). Madison, WI.

Huang, Y.-C., Selman, B., & Kautz, H. (1999). Control knowledge in planning: benefits and tradeoffs *Proceedings of the Sixteen National Conference on Artificial Intelligence* (pp. 511–517). Orlando, FL.

Kambhampati, S., Katukam, S., & Qu Y. (1996). Failure driven dynamic search control for partial order planners: An Explanation Based Approach. *Artificial Intelligence*, *88(1-2)*, 253–315.

Kambhampati, S. (1999). Improving graphplan's search with EBL & DDB techniques. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*. Stockholm, Sweden: Morgan Kaufmann.

Kautz, H. & Selman, B. (1992). Planning as satisfiability. *Proceedings of the Tenth European Conference on Artificial Intelligence* (pp. 359–363). Vienna, Austria: John Wiley & Sons.

Kautz, H. & Selman, B. (1996). Pushing the envelope: planning, propositional logic, and stochastic search. *Proceedings of the Thirteen National Conference on Artificial Intelligence* (pp. 1194–1201). Portand, OR: AAAI Press.

Kautz, H. & Selman, B. (1998). The role of domain-specific axioms in the planning as satisfiability framework. *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, Pittsburgh, PA: AAAI Press.

Kautz, H. & Selman, B. (1999). Unifying SAT-based and Graph-based planning. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence* (pp. 318–325). Stockholm, Sweden: Morgan Kaufmann.

Khardon, R. (1999). Learning action strategies for planning domains. *Artificial Intelligence*, *113(1-2)*, 125–148.

Knoblock, C. (1994). Automatically generating abstractions for planning. *Artificial Intelligence*, *68(2)*, 243–302.

Leckie, C. & Zukerman, I. (1998). Learning search control rules for planning. *Artificial Intelligence*, *101*, 63–98.

McDermott, D., *et al.* (1998). PDDL — the planning domain definition language. Yale University, New Haven.

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

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564–569). St. Paul, MN: AAAI Press.

Muggleton, S., & Feng, C. (1992). Efficient induction of logic programs. In S. Muggleton(Ed.), *Inductive Logic Programming*. London: Academic Press Limited.

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

Quinlan, J. R. (1996). Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, *5*, 139–161.

Silverstein, G. & Pazzani, M. J. (1991). Relational clichés: Constraining constructive induction during relational learning. *Proceedings of the Eighth International Workshop on Machine Learning* (pp. 432–436). Evanston, IL: Morgan Kaufmann.

Veloso, M. M. (1992). *Learning by analogical reasoning in general problem solving*. Doctoral dissertation, Department of Computer Science, Carnegie Mellon University, Pittsburgh.

Weld, S. D. (1999). Recent advances in AI planning. *AI Magazine*, *20(2)*, 93–123.