Learning Pruning Rules for Heuristic Search Planning

Michal Krajňanský[1] and **Jörg Hoffmann**[1] and **Olivier Buffet**[2] and **Alan Fern**[3]

Abstract. When it comes to learning control knowledge for planning, most works focus on "how to do it" knowledge which is then used to make decisions regarding which actions should be applied in which state. We pursue the opposite approach of learning "how to *not* do it" knowledge, used to make decisions regarding which actions should *not* be applied in which state. Our intuition is that "bad actions" are often easier to characterize than "good" ones. An obvious application, which has not been considered by the few prior works on learning bad actions, is to use such learned knowledge as action pruning rules in heuristic search planning. Fixing a canonical rule language and an off-the-shelf learning tool, we explore a novel method for generating training data, and implement rule evaluators in state-of-the-art planners. The experiments show that the learned rules can yield dramatic savings, even when the native pruning rules of these planners, i.e., preferred operators, are already switched on.

1 Introduction

Learning can be applied to planning in manifold ways. To name a few, existing approaches include learning to predict planner performance (e.g., [16]), learning macro actions (e.g., [2, 3]), learning to improve a heuristic (e.g., [20]), learning which heuristic to use when [6], and learning portfolio configurations (e.g., [17]).

The approach we pursue here is the venerable (i.e., old) idea of learning *control knowledge*, in the sense of "domain-dependent information about the structure of plans". That approach has a long tradition, focusing almost entirely on "how to do it" knowledge, mostly learning representations of closed-loop action-selection policies or open-loop macro actions. Learned policies are often used for search-free plan generation (e.g., [12, 7, 8, 19, 4]). Recent work has also used learned policies for macro generation during search (e.g., [20, 4]).

In this work, we pursue an alternative approach of learning "how to *not* do it" knowledge. Consider, e.g., Sokoban. Finding the "good" actions in many critical states is very hard to do, as it effectively entails search or already knowing what the solution is. In contrast, with a bit of practice it is often easy to avoid clearly "bad" actions (like, blocking an exit) based on simple features of the state. A plausible hypothesis therefore is that it may be easier to learn a representation that is able to reliably identify *some* of the bad actions in a state, compared to learning to reliably select a good action.[4]

Indeed, in the literature on search, *pruning rules* – conditions under which the search discards an applicable action – play a prominent role. Temporal logic pruning rules are highly successful in hand-tailored planning with TLPlan [1] and TALPlanner [13]. Pruning rules derived as a side effect of computing a heuristic function, commonly referred to as *helpful actions* or *preferred operators*, are of paramount importance to the performance of domain-independent heuristic search planners like FF [10], Fast Downward [9], and LAMA [15]. In fact, it has been found that such pruning typically is more important to performance than the differences between many of the heuristic functions that have been developed [14].

Despite the prominence of pruning from a search perspective, hardly any research has been done on learning to characterize bad actions (presumably due to the traditional focus on learning stand-alone knowledge as opposed to helping a search algorithm). To the best of our knowledge, there are exactly two such prior works. Considering SAT-based planning, Huang et al. [11] learn simple datalog-style conjunctive pruning rules, conveniently expressed in the form of additional clauses. They find this method to be very effective empirically, with speed-ups of up to two orders of magnitude on a collection of mostly transport-type domains (although, from today's perspective, it should be mentioned that the original planner, but not the one using the pruning rules, is time-step optimal). More recently, de la Rosa and McIlraith [5] tackled the long-standing question of how to automatically derive the control knowledge for TLPlan and TALPlanner. Accordingly, their pruning rules are formulated in linear temporal logic (LTL); they introduce techniques to automatically generate derived predicates to expand the feature space for these rules. Experiments in three domains show that these rules provide for performance competitive with that of hand-written ones.

Against this background, our work is easy to describe: Like de la Rosa and McIlraith, we hook onto the search literature in attempting to learn a prominent form of pruning; while de la Rosa and McIlraith considered TLPlan, we consider action pruning (à la preferred operators) in heuristic search planning. The idea is to let that powerful search framework do the job of finding the "good" actions, reducing our job to helping out with quickly discarding the bad ones. Like Huang et al., we concentrate on simple datalog-style conjunctive pruning rules, the motivation being to determine first how far such a simple framework carries. (More complex frameworks, and in particular the application of de la Rosa and McIlraith's rules in heuristic search planning, are left open as future topics.) We also diverge from prior work in the generation of training data, which we derive comprehensively from all optimal states as opposed to just the states visited by one (or a subset of) solutions.

As it turns out, our simple approach is quite promising. Experimenting with the IPC'11 learning track benchmarks, we obtain dra-

[1] Saarland University, Saarbrücken, Germany, {krajnansky,hoffmann}@cs.uni-saarland.de
[2] INRIA / Université de Lorraine,Nancy, France, olivier.buffet@loria.fr
[3] Oregon State University, Corvallis, USA, afern@eecs.oregonstate.edu

[4] Note the "some" here: learning to reliably identify *all* bad actions is equivalent to learning to identify all good actions. Our focus is on learning a *subset* of the bad actions. From a machine learning perspective, this corresponds to the precision-recall tradeoff. We are willing to sacrifice recall (the percentage of bad actions that are pruned), in favor of precision (the percentage of pruned actions that are bad). This makes sense as it avoids removing solutions from the search space.

matic speed-ups over standard search configurations in Fast Downward, on several domains. The speed-ups are counter-balanced by equally dramatic losses on other domains, but a straightforward portfolio approach suffices to combine the complementary strengths of the different configurations involved.

We next introduce our notations. We then detail our features for learning, the generation of training data, our formulation of pruning rules and how they are being learned, as well as their usage during the search. We present our experiments and conclude.

2 Preliminaries

Our approach requires that states be represented as sets of instantiated first-order atoms (so we can learn first-order conjunctive pruning conditions), that actions are instantiated action schemas (so the pruning conditions can be interpreted as rules disallowing particular schema instantiations in a given state), and that the first-order predicates and the action schemas are shared across the entire planning domain (so the rules can be transferred across instances of the domain). Apart from this, we don't need to make any assumptions, in particular as to how exactly action schemas are represented and how their semantics is defined.

Our assumptions are obviously satisfied by sequential planning in all variants of deterministic non-metric non-temporal PDDL. Our pruning rules are designed for use during a forward search. In our concrete implementation, we build on FF [10] and Fast Downward (FD) [9]. In what follows, we introduce minimal notation as will be needed to describe our techniques and their use in forward search.

We presume a fixed *planning domain* D, associated with a set P of first-order *predicates*, each $p \in P$ with *arity* $arity_p$; we identify p with a string (its "name"). D is furthermore associated with a set A of *action schemas*, each of which has the form $a[X]$ where a is the schema's name and X is a tuple of *variables*; we will sometimes identify X with the set of variables it contains.

A *first-order atom* has the form $p[X]$ where $p \in P$ and X is an $arity_p$-tuple of variables; like for action schemas, we will sometimes identify X with the set of variables it contains. A *first-order literal* $l[X]$ is either a first-order atom $p[X]$ (a *positive literal*), or a negated first-order atom $\neg p[X]$ (a *negative literal*).

An *instance* Π of the domain D comes with a set O of *objects*. A *ground atom* then has the form $p[o_1, \ldots, o_k]$ where $p \in P$, $o_i \in O$, and $k = arity_p$. *Ground literals* are defined in the obvious manner. A *ground action* has the form $a[o_1, \ldots, o_k]$ where $a[X] \in A$, $o_i \in O$, and $k = |X|$; we will often denote ground actions simply with "a". A *state* s is a set of ground atoms.

Each domain instance Π is furthermore associated with a state I called the *initial state*, and with a set G of ground atoms called the *goal*. A state s is a *goal state* if $G \subseteq s$.

If s is a state and a is a ground action, then we assume that there is some criterion stating whether a is *applicable* to s, and what the *resulting state* of applying a to s is. A *solution* (or *plan*) for a domain instance is a sequence of ground actions that is iteratively applicable to I, and whose iterated application results in a goal state. The solution is *optimal* if its length is minimal among all solutions. (For simplicity, we do not consider more general action costs, although our approach is applicable to these in principle.)

3 Features

A basic decision is which features to use as input for the learning algorithm. Many previous works on learning control knowledge for states (e.g., [20, 19, 4, 5]) used features different from the state itself, or in addition to the state itself. We did not do that for now, as the simpler approach already led to good results. However, of course, *whether an action is "good" or "bad" often depends on the goal*. As the goal is not reflected in the states during a forward search, we need to *augment* the states with that information.

Given a domain instance Π and a predicate p, denote by $Goal(p)$ some new predicate unique to p (in our implementation, $Goal(p)$ prefixes p's name with the string "$Goal$-"), and with the same arity as p. The **augmented predicates** are obtained as $P \cup \{Goal(p) \mid p \in P\}$. Given a state s in Π, the **augmented state** is obtained as $s \cup \{Goal(p)[o_1, \ldots, o_k] \mid p[o_1, \ldots, o_k] \in G\}$ where G is the instance's goal. In words, we make goal-indicator copies of the predicates, and introduce the respective ground atoms into the states. We assume from now on that this operation has been performed, without explicitly using the word "augmented". The input to the learning algorithm are (augmented) states, the learned rules employ (augmented) predicates, and the rule usage is based on evaluating these (augmented) predicates against (augmented) states during the search.

For example, in a transportation domain with predicate $at[x, y]$, we introduce the augmented predicate $Goal\text{-}at[x, y]$. If $at[o_1, c_2] \in G$ is a goal atom, we augment all states with $Goal\text{-}at[o_1, c_2]$. In our experiments, the majority of the learned rules ($\geq 70\%$ in 5 of 9 domains) contain at least one augmented predicate in the rule condition.

4 Generating the Training Data

The pruning rules we wish to learn are supposed to represent, given a state s, what are the "bad action choices", i.e., which applicable ground actions should not be expanded by the search. *But when is an action "bad" in a state? How should we design the training data?*

Almost all prior approaches to learning control knowledge (e.g., [12, 7, 20, 19]) answer that question by choosing a set of *training problem instances*, generating a single plan for each, and extracting the *training data* from that plan. In case of learning which actions should be applied in which kinds of states, in particular, it is basically assumed that the choices made by the plan – the action a applied in any state the plan s visits – are "good", and every other action a' applicable to these states s is "bad". Intuitively, the "good" part is justified as the training plan works for its instance, but the "bad" part ignores the fact that other plans might have worked just as well, resulting in noisy training data. Some prior approaches partly counteract this by removing unnecessary ordering constraints from the plan, thus effectively considering a subset of equally good plans. However, those approaches are incomplete and can still mislabel "good" actions as "bad". Herein, we employ a more radical approach based on generating *all* optimal plans.

We assume any planning tool that parses domain D and an instance Π, that provides the machinery to run forward state space search, and that provides an admissible heuristic function h. To generate the training data, we use A* with small modifications. Precisely, our base algorithm is the standard one for admissible (but potentially inconsistent) heuristics: best-first search on $g + h$ where g is path length; maintaining a pointer to the parent node in each search node; duplicate pruning against all generated states, updating the parent pointer (and re-opening the node if it was closed already) if the new path is cheaper. We modify two aspects of this algorithm, namely (a) the termination condition and (b) the maintenance of parent pointers.

For (a), instead of terminating when the first solution is found, we stop the search only when the best node in the open list has $g(s) + h(s) > g^*$ where g^* is the length of the optimal solution (which we

found beforehand). For (b), instead of maintaining just one pointer to the best parent found so far, we maintain a list of pointers to all such parents. Thanks to (a), as $g(s) + h(s)$ is a lower bound on the cost of any solution through s, and as all other open nodes have at least value $g + h$, upon termination we must have generated all optimal solutions. Thanks to (b), at that time we can find the set S^* of all states on optimal plans very easily: Simply start at the goal states and backchain over all parent pointers, collecting all states along the way until reaching the initial state. The training data then is:

- **Good examples E^+:** Every pair (s, a) of state $s \in S^*$ and ground action a applicable to s where the outcome state s' of applying a to s is a member of S^*.
- **Bad examples E^-:** Every pair (s, a) of state $s \in S^*$ and ground action a applicable to s where the outcome state s' of applying a to s is *not* a member of S^*.

Given several training instances, E^+, respectively E^-, are obtained simply as the union of E^+, respectively E^-, over all those instances.

To our knowledge, the only prior work taking a similar direction is that of de la Rosa et al. [4]. They generate all optimal plans using a depth-first branch and bound search with no duplicate pruning. A subset of these plans is then selected according to a ranking criterion, and the training data is generated from that subset. The latter step, i.e. the training data read off the solutions, is similar to ours, corresponding basically to a subset of S^* (we did not investigate yet whether such subset selection could be beneficial for our approach as well). The search step employed by de la Rosa et al. is unnecessarily ineffective as the same training data could be generated using our A*-based method, which does include duplicate pruning (a crucial advantage for search performance in many planning domains).

We will refer to the above as the

- **conservative** training data (i.e.based on all optimal plans), contrasted with what we call
- **greedy** training data.

The latter is oriented closely at the bulk of previous approaches: For the greedy data we take S^* to be the states along a single optimal plan only, otherwise applying the same definition of E^+ and E^-. In other words, in the greedy training data, (s, a) is "good" if the optimal plan used applies a to s, and is "bad" if the optimal plan passed through s but applied an action $a' \neq a$.

Note that above all actions in each state of S^* are included in either E^+ or E^-. We refer to this as the

- **all-operators** training data, contrasted with what we call
- **preferred-operators** training data.

In the latter, E^+ and E^- are defined as above, but are restricted to the subset of state/action pairs (s, a) where $s \in S^*$, and ground action a is applicable to s *and* is a helpful action for s (according to the relaxed plan heuristic h^{FF} [10]). Knowledge learned using this modified training data will be used only within searches that already employ this kind of action pruning: The idea is to focus the rule learning on those aspects missed by this native pruning rule.

Similarly to de la Rosa et al. [4], in our implementation the training data generation is approximate in the sense that we use the relaxed plan heuristic h^{FF} as our heuristic h. h^{FF} is not in general admissible, but in practice it typically does not over-estimate. Hence this configuration is viable in terms of runtime and scalability, and in terms of the typical quality of the training data generated.

There is an apparent mismatch between the distribution of states used to create the training data (only states on optimal plans) and the distribution of states that will be encountered during search (both optimal and sub-optimal states). Why then should we expect the rules to generalize properly when used in the context of search?

In general, there is no reason for that expectation, beyond the intuition that bad actions on optimal states will typically be bad also on sub-optimal ones sharing the relevant state features. It would certainly be worthwhile to try training on intelligently selected suboptimal states. Note though that, as long as the pruning on the optimal states retains the optimal plans (which is what we are trying to achieve when learning from conservative data), even arbitrary pruning decisions at suboptimal states do not impact the availability of optimal plans in the search space.

5 Learning the Pruning Rules

Our objective is to learn some representation R, in a form that generalizes across instances of the same domain D, so that R covers a large fraction of bad examples in E^- without covering any of the good examples E^+. We want to use R for pruning during search, where on any search state s, an applicable ground action a will not be expanded in case (s, a) is covered by R. It remains to define *what kind of representation will underlie R, what it means to "cover" a state/action pair (s, a), and how R will be learned.* We consider these in turn.

As previously advertized, we choose to represent R in the form of a set of *pruning rules*. Each rule $r[Y] \in R$ takes the form $r[Y] =$

$$\neg a[X] \Leftarrow l_1[X_1] \wedge \cdots \wedge l_n[X_n]$$

where $a[X]$ is an action schema from the domain D, $l_i[X_i]$ are first-order literals, and $Y = X \cup \bigcup_i X_i$ is the set of all variables occuring in the rule. In other words, we associate each action schema with conjunctive conditions identifying circumstances under which the schema is to be considered "bad" and should be pruned. As usual, we will sometimes refer to $\neg a[X]$ as the rule's *head* and to the condition $l_1[X_1] \wedge \cdots \wedge l_n[X_n]$ as its *body*.

We choose this simple representation for precisely that virtue: simplicity. Our approach is (relatively) simple to implement and use, and as we shall see can yield excellent results.

Given a domain instance with object set O, and a pruning rule $r[Y] \in R$, a *grounding* of $r[Y]$ takes the form $r =$

$$\neg a[o^1, \ldots, o^k] \Leftarrow l_1[o^1_1, \ldots, o^{k_1}_1] \wedge \cdots \wedge l_n[o^1_n, \ldots, o^{k_n}_n]$$

where $o^j = o^{j'}_{i'}$ whenever X and $X_{i'}$ share the same variable at position j respectively j', and $o^j_i = o^{j'}_{i'}$ whenever X_i and $X_{i'}$ share the same variable at position j respectively j'. We refer to such r as a *ground pruning rule*. In other words, ground pruning rules are obtained by substituting the variables of pruning rules with the objects of the domain instance under consideration.

Assume now a state s and a ground action a applicable to s. A ground pruning rule $r = [\neg a' \Leftarrow l_1 \wedge \cdots \wedge l_n]$ *covers* (s, a) if $a' = a$ and $s \models l_1 \wedge \cdots \wedge l_n$. A pruning rule $r[Y]$ covers (s, a) if there exists a grounding of $r[Y]$ that covers (s, a). A set R of pruning rules covers (s, a) if one of its member rules does.

With these definitions in hand, our learning task – learn a set of pruning rules R which covers as many bad examples in E^- as possible without covering any of the good examples E^+ – is a typical *inductive logic programming* (ILP) problem: We need to learn a set of logic programming rules that explains the observations as given by our training data examples. It is thus viable to use off-the-shelf tool support. We chose the well-known Aleph toolbox [18]. (Exploring application-specific ILP algorithms for our setting is an open topic.)

In a nutshell, in our context, Aleph proceeds as follows:

1. If $E^- = \emptyset$, stop. Else, select an example $(s, a) \in E^-$.

2. Construct the "bottom clause", i.e., the most specific conjunction of literals that covers (s, a) and is within the language restrictions imposed. (See below for the restrictions we applied.)
3. Search for a subset of the bottom clause yielding a rule $r[Y]$ which covers (s, a), does not cover any example from E^+, and has maximal *score* (covers many examples from E^-).
4. Add $r[Y]$ to the rule set, and remove all examples from E^- covered by it. Goto 1.

Note that our form of ILP is simple in that there is no recursion. The rule heads (the action schemas) are from a fixed and known set separate from the predicates to be used in the rule bodies. Aleph offers support for this simply by separate lists of potential rule heads respectively potential body literals. These lists also allow experimentation with different language variants for the rule bodies:

- **Positive vs. mixed conditions:** We optionally restrict the rule conditions to contain only positive literals, referring to the respective variant as "positive" respectively "mixed". The intuition is that negative condition literals sometimes allow more concise representations of situations, but their presence also has the potential to unnecessarily blow up the search space for learning.
- **With vs. without inequality constraints:** As specified above, equal variables in a rule will always be instantiated with the same object. But, per default, different variables also may be instantiated with the same object. Aleph allows "$x \neq y$" body literals to prevent this from happening. Similarly to the above, such inequality constraints may sometimes help, but may also increase the difficulty of Aleph's search for good rules.

As the two options can be independently switched on or off, we have a total of four condition language variants. We will refer to these by **P**, **M**, $\mathbf{P}^{\neq}$, and $\mathbf{M}^{\neq}$ in the obvious manner.

We restrict negative condition literals, including literals of the form $x \neq y$, to use **bound variables** only: In any rule $r[Y]$ learned, whenever variable x occurs in a negative condition literal, then x must also occur in either a positive condition literal or in the rule's head.[5] Intuitively, this prevents negative literals from having excessive coverage by instantiating an unbound variable with all values that do *not* occur in a state (e.g., "$\neg at[x, y]$" collects all but one city y for every object x). Note that, in our context, head variables are considered to be bound as their instantiation will come from the ground action a whose "bad" or "good" nature we will be checking.

Aleph furthermore allows various forms of fine-grained control over its search algorithm. We used the default setting for all except two parameters. First, the rule length bound restricts the search space to conditions with at most L literals. We empirically found this parameter to be of paramount importance for the runtime performance of learning. Furthermore, we found that $L = 6$ was an almost universally good "magic" setting of this parameter in our context: $L > 6$ rarely ever lead to better-performing rules, i.e., to rules with more pruning power than those learned for $L = 6$; and $L < 6$ very frequently lead to much worse-performing rules. We thus fixed $L = 6$, and use this setting throughout the experiments reported. Second, minimum coverage restricts the search space to rules that cover at least C examples from E^-. We did not run extensive experiments examining this parameter, and fixed it to $C = 2$ to allow for a maximally fine-grained representation of the training examples (refraining only from inserting a rule for the sake of a single state/action pair).

[5] We implemented this restriction via the "input/output" tags Aleph allows in the lists of potential rule heads and body literals. We did not use these tags for any other purpose than the one described, so we omit a description of their more general syntax and semantics.

6 Using the Pruning Rules

Given a domain instance Π, a state s during forward search on Π, and an action a applicable to s, we need to test whether R covers (s, a). If the answer is "no", proceed as usual; if the answer is "yes", prune a, i.e., do not generate the resulting state.

The issue here is computational efficiency: We have to pose the question "does R cover (s, a)?" not only for every state s during a combinatorial search, but even for every action a applicable to s. So it is of paramount importance for that test to be fast. Indeed, we must avoid the infamous utility problem, identified in early work on learning for planning, where the overhead of evaluating the learned knowledge would often dominate the potential gains.

Unfortunately, the problem underlying the test is **NP**-complete: For rule heads with no variables, and rule bodies with only positive literals, we are facing the well-known problem of evaluating a Boolean conjunctive query (the rule body) against a database (the state). More precisely, the problem is **NP**-complete when considering arbitrary-size rule bodies ("combined complexity" in database theory). When fixing the rule body size, as we do in our work (remember that $L = 6$), the problem becomes polynomial-time solvable ("data complexity"), i.e., exponential in the fixed bound. For our bound 6, this is of course still way too costly with a naïve solution enumerating all rule groundings. We employ backtracking in the space of partial groundings, using unification to generate only partial groundings that match the state and ground action in question. In particular, a key advantage in practice is that, typically, many of the rule variables occur in the head and will thus be fixed by the ground action a already, substantially narrowing down the search space.

For the sake of clarity, let us fill in a few details. Say that s is a state, $a[o_1, \ldots, o_k]$ is a ground action, and $\neg a[x_1, \ldots, x_k] \Leftarrow l_1[X_1] \wedge \cdots \wedge l_n[X_n]$ is a pruning rule for the respective action schema. We view the positive respectively negative body literals as sets of atoms, denoted L_P respectively L_N. With $\alpha := \{(x_1, o_1), \ldots, (x_k, o_k)\}$, we set $L_P := \alpha(L_P)$ and $L_N := \alpha(L_N)$, i.e., we apply the partial assignment dictated by the ground action to every atom. We then call the following recursive procedure:

if $L_P \neq \emptyset$ **then**
  select $l \in L_P$
  for all $q \in s$ unifiable with l via partial assignment β **do**
    if recursive call on $\beta(L_P \setminus \{l\})$ and $\beta(L_N)$ succeeds **then**
      succeed
    endif
  endfor
  fail
else /* $L_P = \emptyset$ */
  if $L_N \cap s = \emptyset$ **then** succeed **else** fail **endif**
endif

The algorithm iteratively processes the atoms in L_P. When we reach L_N, i.e., when all positive body literals have already been processed, all variables must have been instantiated because negative literals use bound variables only (cf. previous section). So the negative part of the condition is now a set of ground atoms and can be tested simply in terms of its intersection with the state s.

We use two simple heuristics to improve runtime. Within each rule condition, we order predicates with higher arity up front so that many variables will be instantiated quickly. Across rules, we dynamically adapt the order of evaluation. For each rule r we maintain its "success count", i.e., the number of times r fired (pruned out an action). Whenever r fires, we compare its success count with that of the preceding rule r'; if the count for r is higher, r and r' get switched. This simple operation takes constant time but can be quite effective.

7 Experiments

We use the benchmark domains from the learning track of IPC'11. All experiments were run on a cluster of Intel E5-2660 machines running at 2.20 GHz. We limited runtime for training data generation to 15 minutes (per task), and for rule learning to 30 minutes (per domain, configuration, and action schema). To obtain the training data, we manually played with the generator parameters to find maximally large instances for which the learning process was feasible within these limits. We produced 8–20 training instances per domain and training data variant (i.e., **conservative** vs. **greedy**). Handling sufficiently large training instances turned out to be a challenge in Gripper, Rovers, Satellite and TPP. For example, in Gripper the biggest training instances contain 3 grippers, 3 rooms and 3 objects; for Rovers, our training instances either have a single rover, or only few waypoints/objectives. We ran all four condition language variants – **P**, **M**, $\mathbf{P}^{\neq}$, and $\mathbf{M}^{\neq}$ – on the same training data. We show data only for the language variants with inequality constraints, i.e., for $\mathbf{P}^{\neq}$ and $\mathbf{M}^{\neq}$), as these generally performed better.

	all-operators								preferred-operators							
	Conservative				Greedy				Conservative				Greedy			
	$P^{\neq}$		$M^{\neq}$		$P^{\neq}$		$M^{\neq}$		$P^{\neq}$		$M^{\neq}$		$P^{\neq}$		$M^{\neq}$	
	#	L	#	L	#	L	#	L	#	L	#	L	#	L	#	L
Barman	14	2.7	5	2.4	17	2.1	17	1.8	7	2.9	5	2.4	8	2.1	8	1.5
Blocksworld	29	4.4	0	—	61	3.8	23	2.7	28	4.3	0	—	46	3.7	21	2.7
Depots	2	4.5	1	4	16	3.3	10	2.8	4	4.8	2	4	12	3.4	9	3.1
Gripper	27	4.9	1	4	26	4.1	23	3.2	20	4.8	9	4	17	4.2	11	3.4
Parking	92	3.4	51	2.8	39	2.6	31	2.2	71	3.3	48	2.8	20	2.6	18	2.1
Rover	30	2.2	18	1.8	45	1.8	36	1.6	3	2	3	2	14	1.7	16	1.7
Satellite	27	3.2	26	3	25	2.6	22	2.2	12	3.4	12	3	9	3	9	2.6
Spanner	1	3	1	3	1	3	1	3	1	3	1	3	1	3	1	3
TPP	13	2.5	10	2.4	18	2.6	21	2.6	6	2.8	5	2.8	11	2.7	12	2.8

Table 1. Statistics regarding the rule sets learned. "#": number of rules; "L": average rule length (number of rule body literals).

Table 1 shows statistics about the learned rule sets. One clear observation is that fewer rules tend to be learned when using preferred-operators training data. This makes sense simply as that training data is smaller. A look at rule length shows that rules tend to be short except in a few cases. A notable extreme behavior occurs in Spanner, where we learn a single three-literal pruning rule, essentially instructing the planner to not leave the room without taking along all the spanners. As it turns out, this is enough to render the benchmark trivial for heuristic search planners. We get back to this below.

We implemented parsers for our pruning rules, and usage during search, in FF [10] and Fast Downward (FD) [9]. We report data only for FD; that for FF is qualitatively similar. To evaluate the effect of our rules when using/not using the native pruning, as "base planners" we run FD with h^{FF} in single-queue lazy greedy best-first search (**FD1**), respectively in the same configuration but with a second open list for states resulting from preferred operators (**FD2**). To evaluate the effect of our rules on a representation of the state of the art in runtime, we run (the FD implementation of) the first search iteration of **LAMA** [15], which also is a dual-queue configuration where one open list does, and one does not, use the native pruning. As we noticed that, sometimes, FD's *boosting* (giving a higher preference to the preferred-operators queue), is detrimental to performance, we also experimented with configurations not using such boosting.

In both dual-queue configurations, we apply our learned pruning rules only to the preferred-operators queue, keeping the other "complete" queue intact. The **preferred-operators** training data is used in these cases. For FD1, where we apply the rules to a single queue not using preferred operators, we use the **all-operators** training data.

For the experiments on test instances, we used runtime (memory) limits of 30 minutes (4 GB). We used the original test instances from IPC'11 for all domains except Gripper and Depots, where LAMA was unable to solve more than a single instance (with or without our rules). We generated smaller test instances using the generators provided, using about half as many crates than the IPC'11 test instances in Depots, and cutting all size parameters by about half in Gripper.

Table 2 gives a summary of the results. Considering the top parts of the tables (FD-default with boosting where applicable), for 4 out of 9 domains with FD1, for 4 domains with FD2, and for 4 domains with LAMA, the best coverage is obtained by one of our rule-pruning configurations. Many of these improvements are dramatic: 2 domains (FD1: Barman and Spanner), 3 domains (FD2: Barman, Blocksworld, and Parking), respectively 1 domain (LAMA: Barman). When switching the boosting off in FD2 and LAMA, a further dramatic improvement occurs in Satellite (note also that, overall, the baselines suffer a lot more from the lack of boosting than those configurations using our pruning rules). Altogether, our pruning rules help in different ways for different base planners, and can yield dramatic improvements in 5 out of the 9 IPC'11 domains.

The Achilles heel lies in the word "can" here: While there are many great results, they are spread out across the different configurations. We did not find a single configuration that combines these advantages. Furthermore, on the two domains where our pruning techniques are detrimental – Rovers and TPP – we lose dramatically, so that, for the default (boosted) configurations of FD2 and LAMA, in overall coverage we end up doing substantially worse.

In other words, our pruning techniques (a) have high variance and are sensitive to small configuration details, and (b) often are highly complementary to standard heuristic search planning techniques. Canonical remedies for this are *auto-tuning*, learning a configuration per-domain, and/or *portfolios*, employing combinations of configurations. Indeed, from that perspective, both (a) and (b) could be good news, especially as other satisficing heuristic search planning techniques have a tendency to be strong in similar domains.

A comprehensive investigation of auto-tuning and portfolios is beyond the scope of this paper, but to give a first impression we report preliminary data in Table 2 (bottom right), based on the configuration space $\{\text{FD1, FD2, LAMA}\} \times \{\mathbf{P}, \mathbf{M}, \mathbf{P}^{\neq}, \mathbf{M}^{\neq}\} \times \{\text{boost, no-boost}\}$. For "AutoTune", we created medium-size training data (in between training data and test data size) for each domain, and selected the configuration minimizing summed-up search time on that data. For "Portfolios", we created sequential portfolios of four configurations, namely FD1 Cons $P^{\neq}$, FD2 base planner (boosted), LAMA Cons $P^{\neq}$ (boosted), and LAMA Greedy $M^{\neq}$ not boosted. For "Seq-Uniform" each of these gets 1/4 of the runtime (i.e., 450 seconds); for "Seq-Hand", we played with the runtime assignments a bit, ending up with 30, 490, 590, and 690 seconds respectively. Despite the comparatively little effort invested, these auto-tuned and portfolio planners perform vastly better than any of the components, including LAMA.

Regarding rule content and its effect on search, the most striking, and easiest to analyze, example is Spanner. Failing to take a sufficient number of spanners to tighten all nuts is the major source of search with delete relaxation heuristics. Our single learned rule contains sufficient knowledge to get rid of that, enabling FD1 to solve every instance in a few seconds. This does not work for FD2 and LAMA because their preferred operators prune actions taking spanners (the relaxed plan makes do with a single one), so that the combined pruning (preferred operators *and* our rule) removes the plan. We made an attempt to remedy this by pruning with our rules on one queue and with preferred operators on the other, but this did not work either (presumably because, making initial progress on the heuristic value, the preferred operators queue gets boosted). The simpler and more successful option is to use a portfolio, cf. above.

	FD1 (h^{FF})									FD2 (dual queue h^{FF}+ preferred operators)																			
	base pl.	Cons P$^{\neq}$		Cons M$^{\neq}$		Greedy P$^{\neq}$		Greedy M$^{\neq}$		base planner			Cons P$^{\neq}$				Cons M$^{\neq}$				Greedy P$^{\neq}$				Greedy M$^{\neq}$			
	C	C	¬S	C	¬S	C	¬S	C	¬S	C	T	E	C	T	E	RT	C	T	E	RT	C	T	E	RT	C	T	E	RT
Barman (30)	0	**27**	0	0	0	0	0	0	0	14	609.6	271972	13	12.9	28.9	63%	23	17.1	39.2	57%	**27**	1.0	1.4	47%	21	1.6	2.3	45%
Blocksworld (30)	0	0	0	0	0	0	18	**1**	0	19	37.4	19916	18	0.6	1.0	54%	19	1.2	1.0	0%	1	0.0	0.0	85%	**27**	3.6	3.4	17%
Depots (30)	**13**	**13**	0	**13**	0	**13**	12	**13**	11	18	48.2	111266	18	0.7	1.1	33%	18	0.8	1.0	20%	**23**	1.6	2.1	18%	21	3.2	3.5	18%
Gripper (30)	13	0	0	**15**	0	0	23	0	20	**29**	3.9	2956	19	0.0	0.1	95%	26	0.0	0.1	90%	19	0.0	0.3	96%	17	0.0	0.2	84%
Parking (30)	1	3	0	**4**	0	0	30	0	30	7	642.5	16961	8	0.5	0.5	6%	6	0.8	0.8	5%	**25**	35.5	15.2	2%	14	15.3	11.8	1%
Rover (30)	0	0	29	0	3	0	1	0	0	**30**	41.9	22682	11	0.0	0.1	91%	12	0.0	0.1	91%	3	0.0	0.1	94%	13	0.0	0.1	83%
Satellite (30)	0	0	0	0	0	0	0	0	1	**3**	752.3	51741	0	—	—	—	0	—	—	—	2	0.5	0.7	54%	0	—	—	—
Spanner (30)	0	**30**	0	**30**	0	**30**	0	**30**	0	0	—	—	0	—	—	—	0	—	—	—	0	—	—	—	0	—	—	—
TPP (30)	0	0	0	0	0	0	0	0	0	**29**	232.5	13057	0	—	—	—	0	—	—	—	0	—	—	—	0	—	—	—
$\sum$ (270)	27	**73**	29	62	3	43	84	44	62	**149**			87				104				100				113			
										no FD preferred operators boosting																		
Satellite (30)										2	1009,0	68253	0	—	—	—	12	1,1	11,1	92%	0	—	—	—	**16**	3,4	23,2	84%
$\sum$ (270)										53			50				65				**80**				72			

	LAMA (first iteration)																				AutoTune	Portfolios	
	base planner			Cons P$^{\neq}$				Cons M$^{\neq}$				Greedy P$^{\neq}$				Greedy M$^{\neq}$					Seq-Uniform	Seq-Hand
	C	T	E	C	T	E	RT	C	T	E	RT	C	T	E	RT	C	T	E	RT	C	C	C
Barman (30)	7	648.1	151749	**30**	23.8	51.1	53%	**30**	5.0	9.7	44%	22	0.8	1.3	38%	21	0.8	1.3	36%	23	30	30
Blocksworld (30)	27	63.5	13093	24	0.7	1.0	45%	27	1.3	1.0	0%	6	0.3	0.6	55%	**30**	14.2	13.5	19%	27	27	28
Depots (30)	23	43.2	37299	22	0.9	1.2	35%	25	0.9	1.0	25%	**26**	7.0	9.9	22%	25	15.3	17.3	22%	23	24	25
Gripper (30)	**29**	6.4	3122	9	0.0	0.0	85%	16	0.0	0.0	87%	21	0.0	0.4	93%	24	0.0	0.2	76%	29	28	29
Parking (30)	26	699.3	3669	10	0.4	0.2	7%	16	1.4	1.2	7%	**29**	10.2	5.5	2%	28	11.3	6.1	2%	28	30	30
Rover (30)	**29**	211.2	28899	9	0.1	0.2	78%	10	0.1	0.2	80%	0	—	—	—	7	0.1	0.1	65%	30	29	29
Satellite (30)	**4**	986.7	34739	0	—	—	—	0	—	—	—	0	—	—	—	0	—	—	—	3	13	16
Spanner (30)	0	—	—	0	—	—	—	0	—	—	—	0	—	—	—	0	—	—	—	30	30	30
TPP (30)	**20**	360.5	13262	0	—	—	—	0	—	—	—	0	—	—	—	0	—	—	—	29	18	18
$\sum$ (270)	**165**			104				124				104				135				222	229	235
	no FD preferred operators boosting																						
Satellite (30)	3	819,7	32301	0	—	—	—	22	4,1	26,4	85%	1	0,4	0,8	73%	**23**	4,2	14,0	78%			
$\sum$ (270)	84			80				106				95				**125**						

Table 2. Performance overview. "C": coverage; "¬S": all solutions pruned out (search space exhausted); "T" search time and "E" number of expanded states (median for base planner, median ratio "base-planner/rules-planner" for planners using our rules); "RT": median percentage of total time spent evaluating rules. For each base planner, best coverage results are highlighted in boldface. By default, FD's preferred operators queue in FD2 and LAMA is boosted; we show partial results switching that boosting off. For explanation of the "AutoTune" and "Portfolios" data, see text.

Regarding conservative vs. greedy training data, consider FD1. As that search does not employ a complete "back-up" search queue, if our pruning is too strict then no solution can be found. The "¬S" columns vividly illustrate the risk incurred. Note that, in Parking, while the greedy rules prune out all solutions on FD1 (the same happens when training them on the preferred-operators training data), they yield dramatic improvements for FD2, and significant improvements for LAMA. It is not clear to us what causes this.

Regarding the overhead for rule evaluation, the "RT" columns for LAMA show that this can be critical in Gripper, Rovers, and Satellite. Comparing this to Table 1 (right half), we do see that Gripper tends to have long rules, which complies with our observation. On the other hand, for example, Parking has more and longer rules than Rovers, but its evaluation overhead is much smaller. Further research is needed to better understand these phenomena.

For TPP, where none of the configurations using our rules can solve anything and so Table 1 does not provide any indication what the problem is, observations on smaller examples suggest that solutions otherwise found quickly are pruned: the FD1 search space became larger when switching on the rule usage.

8 Conclusion

We realized a straightforward idea – using off-the-shelf ILP for learning conjunctive pruning rules acting like preferred operators in heuristic search planning – that hadn't been tried yet. The results are quite good, with substantial to dramatic improvements across several domains, yielding high potential for use in portfolios. Together with the simplicity of the approach, this strongly suggests that further research on the matter may be worthwhile. The most immediate open lines in our view are to (a) systematically explore the design of complementary configurations and portfolios thereof, as well as (b) understanding the behavior of the technique in more detail.

Acknowledgments. This work is partially supported by the EU FP7 Programme under grant agreement no. 295261 (MEALS).

REFERENCES

[1] F. Bacchus and F. Kabanza, 'Using temporal logics to express search control knowledge for planning', *AIJ*, **116**(1-2), 123–191, (2000).

[2] A. Botea, M. Enzenberger, M. Müller, and J. Schaeffer, 'Macro-FF: Improving AI planning with automatically learned macro-operators', *JAIR*, **24**, 581–621, (2005).

[3] A. Coles and A. Smith, 'Marvin: A heuristic search planner with online macro-action learning', *JAIR*, **28**, 119–156, (2007).

[4] T. Rosa, S. Jiménez, R. Fuentetaja, and D. Borrajo, 'Scaling up heuristic planning with relational decision trees', *JAIR*, **40**, 767–813, (2011).

[5] T. Rosa and S. McIlraith, 'Learning domain control knowledge for TLPlan and beyond', in *Proc. PAL'11*, (2011).

[6] C. Domshlak, E. Karpas, and S. Markovitch, 'Online speedup learning for optimal planning', *JAIR*, **44**, 709–755, (2012).

[7] A. Fern, S. Yoon, and R. Givan, 'Approximate policy iteration with a policy language bias: Solving relational Markov decision processes', *JAIR*, **25**, 75–118, (2006).

[8] C. Gretton, 'Gradient-based relational reinforcement-learning of temporally extended policies', in *Proc. ICAPS'07*.

[9] M. Helmert, 'The Fast Downward planning system', *JAIR*, **26**, 191–246, (2006).

[10] J. Hoffmann and B. Nebel, 'The FF planning system: Fast plan generation through heuristic search', *JAIR*, **14**, 253–302, (2001).

[11] Y. Huang, B. Selman, and H. Kautz, 'Learning declarative control rules for constraint-based planning', in *Proc. ICML'00*.

[12] R. Khardon, 'Learning action strategies for planning domains', *AIJ*, **113**(1-2), 125–148, (1999).

[13] J. Kvarnström and M. Magnusson, 'TALplanner in the 3rd IPC: Extensions and control rules', *JAIR*, **20**, 343–377, (2003).

[14] S. Richter and M. Helmert, 'Preferred operators and deferred evaluation in satisficing planning', in *Proc. ICAPS'09*.

[15] S. Richter and M. Westphal, 'The LAMA planner: Guiding cost-based anytime planning with landmarks', *JAIR*, **39**, 127–177, (2010).

[16] M. Roberts and A. Howe, 'Learning from planner performance', *AIJ*, **173**(5-6), 536–561, (2009).

[17] Núñez S, D. Borrajo, and C. Linares, 'Performance analysis of planning portfolios', in *Proc. SoCS'12*.

[18] A. Srinivasan. The Aleph manual, 1999.

[19] Y. Xu, A. Fern, and S. Yoon, 'Iterative learning of weighted rule sets for greedy search', in *Proc. ICAPS'10*.

[20] S. Yoon, A. Fern, and R. Givan, 'Learning control knowledge for forward search planning', *Journal of ML Research*, **9**, 683–718, (2008).