Learning Weighted Rule Sets for Forward Search Planning

Yuehua Xu
School of EECS
Oregon State University
xuyu@eecs.oregonstate.edu

Alan Fern
School of EECS
Oregon State University
afern@eecs.oregonstate.edu

Sungwook Yoon
Palo Alto Research Center
sungwook.yoon@parc.com

Abstract

In many planning domains, it is possible to define and learn good rules for reactively selecting actions. This has lead to work on learning rule-based policies as a form of planning control knowledge. However, it is often the case that such learned policies are imperfect, leading to planning failure when they are used for greedy action selection. In this work, we seek to develop a more robust form of rule-based control knowledge, attempting to leverage the perceived utility of rules while allowing for imperfection. Specifically, we consider learning sets of weighted action-selection rules for a target planning domain, which are used to assign numeric scores to potential state transitions. These scores can then be used to guide forward search strategies for solving problems from the target domain. This approach allows for information from multiple rules to be combined to help maintain robustness to errors. Our learning approach is based on a combination of a heuristic rule learner and RankBoost, an efficient boosting-style algorithm for learning ranking functions. We further show how to improve performance by incorporating FF's heuristic and tuning the rule weights learned by RankBoost using a perceptron-style algorithm. Our initial empirical results show significant promise for this approach in a number of domains.

Introduction

Heuristic search is a fundamental approach to solve planning problems, on which a number of state-of-the-art planners have been built (Bonet and Geffner 1999; Hoffmann and Nebel 2001). The success of these planners is due to the development of domain-independent heuristics that work well across different planning domains. However, there still exist many domains where the domain-independent heuristics are deficient, inspiring investigation on learning mechanisms for heuristic search planning.

Recent work (Yoon, Fern, and Givan 2008; Xu, Fern, and Yoon 2009) has made progress on learning domain-specific heuristics for a target domain, based on prior planning experience. In particular, (Yoon, Fern, and Givan 2008) applied linear regression to learn a linear approximation of the difference between the relaxed plan length heuristic and the observed distances-to-goal of the states on example solution paths. That work also defined a feature space and an approach for learning the features used in the linear approximation. More recently, (Xu, Fern, and Yoon 2009) tightly combine weight learning with the actual search procedure, tuning the feature weights to correct observed search errors on training problems. That work, however, did not include a feature learning mechanism, requiring features to be provided to it before learning. The work in this paper extends (Xu, Fern, and Yoon 2009) to automatic feature learning.

While the above prior work has shown promising results, they are limited to learning control knowledge in the form of heuristics, i.e. ranking functions that only capture information about states. Rather, in this work, we will investigate an alternative approach of learning ranking functions on *state transitions* that can take information about both states and actions into account. Our approach is motivated by prior work (Khardon 1999; Martin and Geffner 2000; Yoon, Fern, and Givan 2002), where decision-lists of action-selection rules were learned in order to define reactive policies for planning domains. Given a good reactive policy, the planning problems from the corresponding domain can be quickly solved without search. While action-selection rules are an intuitively appealing form of control knowledge and give good results in some domains, experience has shown that in many domains the learned reactive policies are often imperfect and result in poor planning performance.

In this paper, we attempt to learn and use action selection rules in a more robust way. In particular, we use sets of weighted rules to define a ranking function on state transitions, which can then be used to guide a search process. Rules are viewed as binary features on transitions, so that the ranking of a potential transition is equal to the sum of active-rule weights for that transition. By combining multiple action selection rules, robustness is improved. In this paper, we focus on learning and using such weighted rule sets to efficiently solve planning problems in the context of greedy search. It is important to note, however, that all of the algorithms in this paper can be easily generalized to the case of breadth-first beam search, which allows for the amount of search to be controlled through the beam width parameter.

It is worth contrasting our approach with other recent work that attempts to overcome the brittleness of learned rule-based policies by integrating them directly into best-

first search (Yoon, Fern, and Givan 2008). In this approach, when a node is expanded during the search, instead of adding just its successors to the priority queue, the reactive policy is executed for a number of steps and nodes generated on the trajectory along with their neighbors are added to the queue. By incorporating reactive policies into search, the action selection rules are used in a more robust way and in a number of domains improvement was observed over best-first search without the policy.

One aspect of the approach of (Yoon, Fern, and Givan 2008) is that the policy is learned completely independently of the search process. In particular, there is no mechanism for directly trying to improve a policy to correct for observed search errors. In contrast, the primary learning approach in this paper is directly driven by observed search errors, focusing its attention on parts of the search space that are observed to be most difficult. A second aspect of (Yoon, Fern, and Givan 2008) is that it is difficult to place a bound on the time required for the search to uncover a solution when it solves a problem. In contrast, one of our primary goals is to learn control knowledge and use search strategies that allow for clear and practical bounds on the runtime in cases where the solution is found at a particular depth. This is the reason for our focus on breadth-first beam search, which has the desired property, and specifically greedy search (i.e. beam width equals to one) in this paper.

The remainder of the paper is organized as follows. First, we give our problem setup for learning rule-based ranking problems. Second, we formulate our learning problem as a ranking problem and describe how to apply a boosting-style algorithm to solve it. Next, we present how to integrate this initial solution into the search process, followed by the description of our rule learner. We finally present experimental results and conclude.

Problem Setup

A planning problem is a tuple (s_0, A, g), where s_0 is the initial state, A is a set of actions, and g is a set of state facts representing the goal. A solution plan for a planning problem is a sequence of actions $(a_1, \ldots, a_l)$, where the sequential application of the sequence starting in state s_0 leads to a goal state s^* where $g \subseteq s^*$. In this paper, we view planning problems as directed graphs where the vertices represent states and the edges represent applicable actions, which will result in state transitions. Planning then reduces to graph search for a path from the initial state to the goal.

Greedy Search

Different search strategies can be used to solve the planning problem, with the guidance of a heuristic or ranking function. As an initial evaluation of our rule-based ranking functions, in this paper, we consider the most basic and efficient strategy — greedy search.

Greedy search can be viewed as a special case of breadth-first beam search with beam width one. In greedy search, the beam contains only one unique node, which is ranked highest among all children of its parent. This ranking is usually done via a heuristic function on states, however, in this work the ranking will be done via a function that ranks state transitions from a parent to a child. At each step of greedy search, the current search node is expanded and the child of the transition with the highest rank is then selected to be the current node. This process continues until a goal node is discovered, at which point a solution plan has been found.

Obviously, greedy search prunes away most nodes in the search space and thus the quality of the ranking function is critical to its success. By extending to breadth-first beam search with beam widths greater than one, it is possible to improve robustness to imperfect ranking functions. While we focus on greedy search in this paper, it is important to note that all of the algorithms extend easily to general bread-first search, and will be a topic of future work.

Rule-based Ranking Function

We consider ranking functions that are represented as linear combinations of features. In prior work (Yoon, Fern, and Givan 2008; Xu, Fern, and Yoon 2009), taxonomic syntax was used to define features over state-goal pairs, resulting in a ranking function that for any pair (s, g) evaluated how good s was with respect to the goal g. In this paper, since we are interested in learning rankings functions for state transitions rather than just states, we introduce a new space of features which are defined by action-selection rules and thus can capture the property of state transitions.

Following prior work (Yoon, Fern, and Givan 2008) that used taxonomic syntax to define action-selection rules for reactive policies, each of our action-selection rules has the form:

$$u(z_1, \ldots, z_k) : L_1, L_2, \ldots, L_m \quad (1)$$

where u is a k-argument action type, the z_i are argument variables, and the L_i are literals of the form $z \in E$ where $z \in \{z_1, \ldots, z_k\}$ and E is a taxonomic class expression. Given a state-action pair (s, g), each class expression E represents a set of objects in the state s, so that each literal can be viewed as constraining a variable to take values from a particular set of objects. For example, **holding** is a taxonomic class expression in the Blocksworld domain and it represents the set of blocks that are being held in the current state. More complex class expressions can be built via operations such as intersection, negation, composition, etc. For example, the class expression **ontable** $\cap$ **gontable** represents the set of objects/blocks that are on the table in both the current state and goal. We refer the reader to (Yoon, Fern, and Givan 2008) for details of taxonomic syntax, noting that the details are not essential to the main contribution of this paper. Given a state-goal pair (s, g) and a ground action a where $a = u(o_1, \ldots, o_k)$, the literal $z_j \in E$ is said to be true if and only if o_j is in the set of objects that is represented by E. We say that the rule *suggests* action a for state-goal pair (s, g) if all of the rule literals are true for a relative to (s, g).

Given a rule of the above form, we can define a corresponding feature function f on state transitions, where a state transition is simply a state-goal-action tuple (s, g, a) such that a is applicable in s. The value of $f(s, g, a) = 1$ iff the corresponding rule suggests a for (s, g) and otherwise $f(s, g, a) = 0$. An example rule in the Blocksworld domain

is: **putdown**$(x_1) : x_1 \in$ **holding**, which defines a feature function f, where $f(s, g, a) = 1$ iff $a =$ **putdown**(o) and **holding**$(o) \in s$ for some object o, and is equal to zero for all other transitions.

Assume we have a set of rules giving a corresponding set of feature functions $\{f_i\}$, the ranking function is then a linear combination of these rule-based features

$$F(s, g, a) = \sum_i \alpha_i \cdot f_i(s, g, a)$$

where α_i is the corresponding real-valued weight of f_i. From this it is clear that the rank assigned to a transition is simply the sum of the weights of all rules that suggest that transition. In this way, rules that have positive weights can increase the rank of a transition, and rules with negative weights can decrease the rank.

Learning to Plan

Given a target planning domain, our goal in this paper is to learn a ranking function represented as a weighted rule set that can quickly solve problems in the domain using greedy search. The learning problem provides a training set of pairs $\{(x_i, y_i)\}$, where each $x_i = (s_{i0}, A, g_i)$ is a planning problem from the target planning domain and each $y_i = (s_{i0}, s_{i1}, \ldots, s_{il})$ is a sequence of states corresponding to the solution path from the initial state s_{i0} to a goal $s_{il} \supseteq g_i$. Here we have assumed, without loss of generality, that all solutions have length l to simplify notation.

Given such a training set the learning goal is to learn a set of rules along with their weights so that when used to rank state transitions, greedy search will uncover the solutions in the training set (or some variant of them, see next section). There is an implicit assumption that the set of training problems are representative of the problem distribution to be encountered in the future, so that learning will be biased toward the most relevant problem space. In addition, the solutions in the training set should reflect good solutions, since learning will attempt to mimic those solutions.

Learning Weighted Rule Set

In this section, we first formulate the learning problem for planning as a ranking problem, and then describe a boosting-style algorithm to learn the weighted rule set.

The Ranking Problem

A ranking learning problem consists of a set of instances I and a feedback function $\phi : I \times I \to \{-1, 0, 1\}$. Given any $v_1 \in I$ and $v_2 \in I$, $\phi(v_1, v_2) = 1$ indicates that v_1 should be ranked higher than v_2, while $\phi(v_1, v_2) = -1$ indicates that v_2 should be ranked higher than v_1. If $\phi(v_1, v_2) = 0$, there is no constraint on the ordering of v_1 and v_2. For the ranking problem, the learning objective is to learn a ranking function F over I that minimizes the number of misranked pairs of nodes relative to the feedback function (Freund et al. 2003). Here we say that F misranks a pair if $\phi(v_1, v_2) = 1$ and $F(v_1) \leq F(v_2)$, or $\phi(v_1, v_2) = -1$ and $F(v_1) \geq F(v_2)$. The hope is that such a learned function will generalize to correctly rank instances outside of the training instances in I. It is typical to learn linear ranking functions of the form $F(v) = \sum_i \alpha_i \cdot f_i(v)$, where the f_i are real valued feature functions that assign scores to instances in I. The α_i are real-valued weights indicating the influence of each feature. In this paper, the instance space I will contain state transitions from a planning domain and our features will be the rule-based features described above.

Our first approach to learning a ranking function for planning is to convert it to a traditional ranking problem as described below. Given a training set $\{(x_i, y_i)\}$ for a planning domain, we consider each node s_{ij} along each solution trajectory y_i and let $C_{i(j+1)}$ be the set of all candidate transitions out of s_{ij}, where a transition from s to s' via action a will be denoted by (s, a, s'). The set of instances for the ranking problem is then $I = \cup_i \cup_{j=1}^{l} C_{ij}$. For simplicity and without loss of generality, assume that for any s_{ij} there is exactly one transition $t_{ij} = (s_{i(j-1)}, a, s_{ij})$ to s_{ij} in C_{ij}, which is the transition observed in y_i. The simplest way to define the feedback function is to require that t_{ij} be ranked higher than other transitions in C_{ij}. Specifically, $\phi(t_{ij}, t) = 1$ and $\phi(t, t_{ij}) = -1$ for any $t \in C_{ij}$ such that $t \neq t_{ij}$ and let ϕ equal to 0 for all other pairs. Any ranking function that does not make any errors on this ranking problem will allow for greedy search to produce all solutions in the training set.

In real applications, however, finding such a perfect ranking function is often impractical. Especially in automated planning, where there can be many equally good solution trajectories to a planning problem other than those in the training set, e.g. by exchanging the ordering of certain actions. In such cases, it becomes infeasible to require that the specific transitions observed in the training data be ranked higher than all other transitions in C_{ij} since many of those other transitions are equally good. To deal with this issue we attempt to determine which other transitions in C_{ij} are also good transitions. To do this we use the heuristic algorithm described in (Veloso, Pérez, and Carbonell 1991) to transform the given solution trajectories to partially ordered plans. The partially ordered plans contain the same set of actions as the totally ordered plan given by the y_i but only include the necessary constraints on the action-ordering. Therefore, every partially ordered plan represents a set of solution paths.

With the partially ordered plans, we can now define the feedback function in a more general way. Let $\delta(t, x_i)$ be a boolean function that determines whether a given transition $t = (s, a, s')$ is on the partially ordered plan for x_i. $\delta(t, x_i) =$ **true** indicates that there exists a solution path consistent with the partially order plan that goes through t. Given this we can arrive at an improved feedback function ϕ as given below.

- For each pair $(t_1, t_2) \in C_{ij} \times C_{ij}$, if $\delta(t_1, x_i) =$ **true** and $\delta(t_2, x_i) =$ **false**, then $\phi(t_1, t_2) = 1$ and $\phi(t_2, t_1) = -1$;
- For all other cases, $\phi(t_1, t_2) = 0$.

This function indicates that for every state on the a solution path y_i, its outgoing transitions that can lead to a solution, according to the partially ordered plan, should be ranked higher than those can not. Assume that a ranking

function similar to the feedback function is found, it is reasonable to believe that it will allow for greedy search to perform well, though this is unfortunately not guaranteed. The reason for this is that the only transitions included in I are those that originate at nodes on the totally ordered training solutions (i.e. the union of transitions in the C_{ij}). Thus, the learned ranking function might lead a greedy search to take a transition that leads off of the training solution, where it has not been trained and hence no guarantees can be made about its performance. One way to solve this problem would be to include all possible transitions in the set of instances I and to attempt to rank all transitions consistent with a partially ordered plan higher than all others. Unfortunately, there can be an exponentially large set of transitions consistent with a partially order plan, making this option intractable in general. Thus, for our first approach, we simply accept this potential pitfall of working with the reduced transition set I described above. We will address this pitfall in our second iterative approach described in the next section.

Learning with RankBoost

By converting our learning-to-plan problems to ranking problems we can now consider apply existing learning algorithms for ranking. RankBoost is a particularly effective boosting-style algorithm that combines a set of weak learners in order to accurately rank a set of instances (Freund et al. 2003). Given a set of instances I and a feedback function $\phi : I \times I \rightarrow \{-1, 0, 1\}$, RankBoost defines a probability distribution D over $I \times I$.

$$D(v_1, v_2) = Z \cdot max\{0, \phi(v_1, v_2)\}, v_1 \in I, v_2 \in I \quad (2)$$

where Z is a normalization factor chosen such that D will be a distribution. The learning objective of RankBoost is to find a ranking function F that minimizes the ranking loss,

$$rLoss_D(F) = \sum_{v_1, v_2} D(v_1, v_2) \cdot \psi(F(v_1) \leq F(v_2))$$

where $\psi(\cdot)$ returns 1 if the term inside is true and returns 0 otherwise. When $\phi(v_1, v_2) = -1$ or 0, $D(v_1, v_2) = 0$. When $\phi(v_1, v_2) = 1$, $D(v_1, v_2) = Z$. Therefore, the ranking loss can also be written as $rLoss_D(F) = Z \cdot \sum_{\phi(v_1, v_2)=1} \psi(F(v_1) \leq F(v_2))$. Note that $\phi(v_1, v_2) = -1$ implies $\phi(v_2, v_1) = 1$. All pairs of instances that have a non-trivial ranking ordering are considered in the loss function. Since Z is a constant, the learning objective of RankBoost equals to minimizing the number of pairs of instances that are misranked by F according to the feedback function ϕ.

RankBoost is an iterative algorithm that adds one feature to a linear ranking function on each iteration so as to continually improve the rank loss. To do this, each round calls a weak learning algorithm to learn a feature that is focused on correctly ranking instance pairs have been most difficult to rank correctly in previous iterations. In our case, the weak learner will learn rule-based features and after n iterations of RankBoost we will have a weighted rule set of size n.

More specifically, as shown in Figure 1, the RankBoost algorithm maintains a distribution over all pairs of instances,

RankBoost (I, D, k)
// I is the set of instances.
// D is the input distribution over $I \times I$.
// k is the number of iterations.
$D_1 = D$
for $i = 1, 2, \ldots, k$:
  $f_i \leftarrow$ **Rule-Learner** (I, D_i)
  // Learning a ranking feature using distribution D_i
  Choose $\alpha_i \in R$
  for each pair $(v_1, v_2) \in I \times I$
    $D_{i+1}(v_1, v_2) = \frac{D_i(v_1, v_2) exp(\alpha_i (f_i(v_2) - f_i(v_1)))}{Z_i}$
  where Z_i is a normalization factor
return $F = \sum_{i=1}^{k} \alpha_i \cdot f_i$

Figure 1: The RankBoost algorithm (Freund et al. 2003).

indicating the importance of these pairs to being ranked correctly by the next learned feature. Initially it is the distribution D defined in Equation 2. This distribution is passed to the weak learner, which attempts to return a feature f_i that achieves a good ranking loss with respect to the current distribution. After learning f_i, RankBoost selects a weight α_i in order to minimize $Z_i = \sum_{v_1, v_2} D_i(v_1, v_2) \exp(\alpha_i (f_i(v_2) - f_i(v_1)))$. The distribution then gets updated in a way that the distribution value for correctly ranked pairs will be decreased and the distribution value for incorrectly ranked pairs will be increased. As a result, the next learning iteration will emphasize pairs that have been misranked more often in previous iterations.

RankBoost has strong theoretical properties, similar to those of traditional boosting algorithms. In particular, under certain assumption about the weak learning algorithm, RankBoost can be guaranteed to decrease the ranking loss on the training data on each iteration and the generalization error of the learned ranking function can be bounded (Freund et al. 2003). Our specific approach for weak learning and setting the weight values follows that of (Freund et al. 2003) where a specialized formulation was presented for binary features, which is the case for our rule-based features. In particular, the weak learner attempts to learn a feature that maximizes $|r|$, where $r = \sum_{v_1, v_2} D(v_1, v_2)(f(v_1) - f(v_2))$ The corresponding weight α_i for a learned feature is then set to $\alpha_i = \frac{1}{2} \ln(\frac{1+r}{1-r})$. While applying the RankBoost algorithm to our planning framework, we only made two modifications motivated by our application domain of planning.

First, we modified RankBoost so that it could take into account prior knowledge provided by an initial heuristic function over states. This is motivated by the fact that prior work (Yoon, Fern, and Givan 2008; Xu, Fern, and Yoon 2009) has found it quite useful to incorporate state-of-the-art heuristics such as relaxed-plan length (Hoffmann and Nebel 2001) into the learned control knowledge. Space limits preclude details of this modification to RankBoost, but the essential idea is to initialize the learned ranking function to be the provided heuristic, in our case relaxed-plan length, and to modify the initial distribution appropriately. The learned ranking is then equal to the heuristic plus a linear combination of learned features that attempt to correct the heuristic.

Our second modification is to further tune the weights of the ranking function returned by RankBoost. In RankBoost, the weight are selected in order to minimize the ranking loss. However, this does not always correspond exactly to the best weights for maximizing planning performance. Thus to help improve the planning performance, we consider using a perceptron-style algorithm to optimize the weights. Initially the weight of each feature f_i is set according to RankBoost. The algorithm then iteratively conducts greedy search and updates the weights to correct the search errors that have been made. For details about the weight learning algorithm, we refer to prior work (Xu, Fern, and Yoon 2009).

Iterative Learning in Search

In last section, we applied the RankBoost algorithm to learn a weighted rule set for solving planning problems. As noted above, theoretically there is no guarantee on the performance of the learned weighted rule set due to the fact that the transitions included in I are only a small fraction of all possible transitions, in particular those transitions that originate from states in the training solution paths. For any transitions outside of I, the learning problem does not specify any constraints on their rankings. Thus, when the learned ranking function leads greedy search to parts of the search space outside of I the search is in truly unchartered territory.

Our approach to help overcome this issue is to integrate the above RankBoost approach with the search process. In particular, the goal is to form ranking problems whose set of transitions I are a better reflection of where a greedy search using the currently learned ranking function is going to go. In particular, it is desirable to include transitions in I where the current ranking function leads greedy search to make mistakes, where a mistake is flagged whenever the greedy search results in a transition that falls outside of the partial order plan of a training example. This allows for learning to focus on such errors and hopefully correct them.

More specifically, Figure 2 gives pseudo-code for our improved approach to learning ranking functions for planning. The top level procedure repeatedly constructs a ranking problem by calling **ConstructRP**, calls RankBoost to learn k new features on it, and then further optimizes the feature/rule weights using our perceptron-style algorithm. The key aspect of this approach is that each ranking problem generated by **ConstructRP** depends on the performance of the currently learned ranking function F when used for greedy search. In particular, given the current ranking function F **ConstructRP** simulates the process of greedy search using F and adds transitions along the simulated path to I along with the corresponding ranking information as specified by δ (see previous section), which determines whether transitions are on the partially-ordered plan of a training problem or not. However, if the greedy search should ever follow an erroneous transition according to δ the search will be artificially forced to follow the highest ranked good transition. In this way the resulting ranking problem focuses on the transitions that are relevant to the current greedy search and in particular those parts where errors are made. Currently there are no convergence results for this learning approach, which is a point of future work. However, this iterative learning approach can significantly improve empirical performance.

IterativeLearning ($\{x_i\}, \delta$, k, H)
// $x_i = (s_{i0}, A, g_i)$: planning problem
// δ : partial-order plan indicator function (see text)
// k : # of RankBoost iterations for each ranking problem
// H : heuristic function to initial ranking function
$F \leftarrow H$ //initialize the ranking function
repeat until no improvement **or** iteration limit
  $(I, D) \leftarrow$ **ConstructRP** ($\{x_i\}, \delta, F$)
  // Construct a ranking problem
  $F' \leftarrow$**RankBoost** (I, D, k)
  $F \leftarrow F + F'$
  $F \leftarrow$**WeightLearning**(F)
return F

ConstructRP ($\{x_i\}, \delta, F$)
$I \leftarrow \emptyset$
$D \leftarrow 0$
for each $x_i = (s_{i0}, A, g_i)$
  $s \leftarrow s_{i0}$
  repeat until $s \supseteq g_i$ // goal achieved
    $C \leftarrow$ all transitions (s, a, s') out of s
    $C_+ \leftarrow \{t \mid t \in C \wedge \delta(t, x_i) = true\}$
    $C_- \leftarrow C - C_+$
    $I \leftarrow I \cup C$
    for each $t_+ \in C_+, t_- \in C_-$
      $D(t_+, t_-) = \frac{1}{Z}(exp(F(t_-) - exp(F(t_+)))$
      // Z is a normalization factor
    $s \leftarrow$ destination of highest ranked transition in C_+
return (I, D)

Figure 2: The iterative learning algorithm.

Learning Action Selection Rules

The RankBoost algorithm assumes the existence of a weaker learner that can be called to produce a ranking feature. In this section, we briefly introduce the rule learner we used. As shown in Figure 1, the input to the rule learner is a pair (I, D), where I is the set of instances and D is a distribution over $I \times I$. In our case, each instance in I is represented as a state-goal-action tuple (s, g, a), on which the rule-based feature can be evaluated. The learning objective is to find a rule that can maximize $|r|$ where $r = \sum_{v_1 \in I, v_2 \in I} D(v_1, v_2)(f(v_1) - f(v_2))$. For this purpose, we adapt the heuristic rule learner described in (Yoon, Fern, and Givan 2008) to find the best rule that can maximize $|r|$. Since the rule space is exponentially large, the approach performs a beam search over possible rules, where the starting rule has no literals in its body and each search step adds one literal to the body. The search terminates when the beam search is unable to further improve $|r|$.

Experimental Results

We present experiments in seven STRIPS domains: Blocksworld, Depots, Driverlog, FreeCell, Pipesworld, Pipesworld-with-tankage and Philosopher. These domains were selected in order to facilitate comparison to prior work

on learning search heuristics (Yoon, Fern, and Givan 2008; Xu, Fern, and Yoon 2009). In future work, we also plan to evaluate our performance on the training and testing sets from the learning track of IPC 2009. We set a time cut-off of 30 CPU minutes and considered a problem to be unsolved if a solution was not found within the time cut-off. For our learning algorithms, the maximum number of learned rules is limited to 30. The learning algorithm will terminate when no improvement can be observed or the maximum number of rules has been reached. Note that the actual size of the learned rule set is usually smaller than 30 since the rule learner may output duplicated rules.

For each domain we needed to create a set of training problems and testing problems on which the weighted rule set would be trained and evaluated. In Blocksworld, all problems were generated using the BWSTATES generator (Slaney and Thiébaux 2001), which produces random Blocksworld problems. Thirty problems with 10 or 20 blocks were used as training data, and 30 problems with 20, 30, or 40 blocks were used for testing. For Driverlog, Depots and FreeCell, the first 20 problems are taken from IPC3 and we generated 30 more problems of varying difficulty to arrive at a set of 50 problems, roughly ordered by difficulty. For each domain, we used the first 15 problems for training and the remaining 35 for testing. The other three domains are all taken from IPC4. Each domain includes 50 or 48 problems, roughly ordered by difficulty. In each case, we used the first 15 problems for training and the remaining problems for testing. The solutions for all training problems are generated by running FF and beam search with different beam widths, selecting the best solutions found, and then transforming these solutions to partially ordered plans as described in (Veloso, Pérez, and Carbonell 1991).

Description of Tables

Before presenting the results we will first provide an overview of the information contained in the results tables. Figure 3 compares the performance of different approaches we used as well as algorithms in prior work for learning control knowledge for greedy search (Yoon, Fern, and Givan 2008; Xu, Fern, and Yoon 2009). These algorithms are:

- Yoon08 : three forms of control knowledge were learned in (Yoon, Fern, and Givan 2008) and were all evaluated for their ability to guide greedy search. The table entries labeled Yoon08 give the best performance among the three types of control knowledge as reported in that work. Results for Yoon08 are only given for our three IPC4 domains: Pipesworld, Pipesworld-with-tankage and Philosopher, for which our training and testing sets exactly correspond.
- RPL : greedy search with FF's relaxed plan length heuristic.
- LaSO-BR$_1$: greedy search with the ranking function learned by LaSO-BR$_1$ (Xu, Fern, and Yoon 2009). This is the closet work to our approach, in which the ranking function is also represented as a linear combination of features and used to control greedy search. Furthermore, the weight learning algorithm in that work is exactly the algorithm we use in this work. The only difference is that in the prior work features were not learned by LaSO-BR$_1$ and were of a different form. They used taxonomic class expressions that capture only state information, while our features are defined on state transitions, including information on both state and actions. The features used in that work were hand-selected from sets of class expressions learned by various other techniques in prior studies.
- RB : greedy search with the weighted rule set that is learned by our initial RankBoost algorithm. Here we assume no prior knowledge and generate a ranking problem from which the weighted rule set is learned.
- RB-H : identical to RB except that we view the relaxed-plan length heuristic as prior knowledge and generate the ranking problem based on it.
- IRB : greedy search with the weighted rule set that is learned via our iterative learning approach starting with no prior knowledge. We choose to learn $k = 5$ rules for each ranking problem generated.
- IRB-H: identical to IRB except that we use the relaxed-plan length heuristic as prior knowledge.

In addition to these algorithms, we also include the performance of FF in Figure 3 to give an indication of the difficulty of the testing problems. Note that FF is a state-of-the-art planner that is not restricted to greedy search. Each column of Figure 3 corresponds to an algorithm and each row corresponds to a target planning domain. The planning performance is first evaluated on the number of solved problems. When two algorithms solve the same number of problems, we will use the median plan length of the solved problems to break the tie. The one that has shorter plans is considered better. For our four approaches, we present the best results that were observed during the learning process.

Figure 4 provides more details of the approaches we used. We add a new column "Learning iterations" indicating how many times the rule learner is called, i.e. how many rules are produced in total. Since there exist duplicated rules, we add a set of four columns that are labeled as "Number of unique rules", giving the actual size of the learned rule set that removes duplications. Now each row corresponds to the performance of the weighted rule set learned after the number of iterations specified by that row. Since IRB and IRB-H learned 5 rules for each ranking problem generated in each iteration, we compared the results after every 5 calls to the rule learner. For example, the first row for Blocksworld corresponds to the weighted rule set learned after 5 rules are induced. However, after removing duplications, the actual size of the weighted rule set is 3 for RB and 4 for RB-H. The next row indicates that the size of the weighted rule set is 7 for RB after 10 rules are induced.

Performance Evaluation

Figure 3 compares the performance of different planners. In Blocksworld, Depots, Driverlog, Pipesworld-with-tankage and Philosopher, FF solves fewer problems than the best

Problems solved (Median plan length)	FF	Yoon08	RPL	LaSO-BR$_1$	RB	RB-H	IRB	IRB-H
Blocksworld	10 (77)	N/A	13 (3318)	27 (840)	30 (126)	30 (166)	**30 (89)**	30 (118)
Depots	14 (63)	N/A	1 (462)	4 (1526)	15 (661)	11 (129)	0 (-)	**23 (433)**
Driverlog	3 (119)	N/A	0 (-)	0 (-)	0 (-)	3 (2852)	0 (-)	**4 (544)**
FreeCell	29 (90)	N/A	5 (96)	7 (132)	5 (155)	7 (96)	2 (213)	**9 (92)**
Pipesworld	20 (50)	0 (-)	11 (114)	16 (1803)	7 (1360)	17 (1063)	7 (1572)	**17 (579)**
Pipesworld-with-tankage	3 (63)	0 (-)	**6 (119)**	5 (55)	1 (1383)	6 (152)	3 (2005)	5 (206)
Philosopher	0 (-)	0 (-)	0 (-)	6 (589)	33 (875)	**33 (363)**	33 (875)	**33 (363)**

Figure 3: Experimental results for different planners. For each domain, we show the number of solved problems and the median plan length of the solved problems. A dash in the table indicates that the median plan length is not available since none of the problems can be solved. N/A indicates that the result of the planner is not applicable here.

greedy planner with learned knowledge, which is highlighted in Figure 3. For FreeCell and Pipesworld, FF outperforms the learning approaches. In this case, the learned knowledge was not powerful enough to allow for greedy search to outperform FF. Future work will examine the use of our learned heuristics in non-greedy search such as breadth-first beam search, which we expect will significantly improve performance.

Among the learning approaches, IRB-H is the best planner in all domains except for Blocksworld and Pipesworld-with-tankage, solving more problems with fairly good plan length. For Blocksworld, IRB is the best planner and solves all problems with the best median plan length. RB, RB-H and IRB-H also solve all problems indicating that the rule-based control knowledge in this domain is more useful than the relax plan length heuristic and the previously learned heuristic. In Pipesworld-with-tankage, RPL outperforms our planners. But RB-H solves the same number of problems, with the plan quality being a little worse. The results show that the learned weighted rule sets significantly outperform the relax plan length heuristic and the heuristic in LaSO-BR$_1$ that only captures information about states. Overall, IRB-H is the best approach and to the best of our knowledge, these are the best reported results of any method for learning control knowledge for greedy search.

Performance Across Learning Iterations. Figure 4 gives more detailed results of our learning approaches. As the learning goes on, the number of rules produced will be non-decreasing. However, since there exist replicated rules, the size of the weighted rule set may not change. For example, IRB learned only 2 unique rules for FreeCell, regardless of how many times the rule learner is called. This either indicates a failure of our rule learner to adequately explore the space of possible rules, or indicates a limitation of our language for representing rules in this domain. These issues will be investigated in future work.

Note that in general, with some exceptions, the planning performance judged in terms of solved problems and median plan length improves as the number of unique rules increases. For example, RB solves 3 problems with 13 rules but 15 problems with 16 rules for Depots. As an exception, however, consider Philosopher, where IRB-H solves all problems with the first 3 rules learned. When one new rule is added, it can not solve any of those problems. It is very likely that our weighted rule set converges to a bad local minima, either because of the weight learning algorithm or the iteratively boosting algorithm, or both.

Iterative Learning vs. Non-iterative learning. IRB and IRB-H can be viewed as an iterative version of RB and RB-H, respectively. In general, with some exceptions, the iterative versions perform better than the non-iterative versions, particularly for IRB-H. For Blocksworld, all of them have similar performance, while IRB improves the plan length over RB. For Depots, the iterative version IRB fails to solve any problem but contrastingly, IRB-H works much better than non-iterative version RB-H. For other domains, the iterative versions often achieve a better performance.

Effect of Relax Plan Length Heuristic. The only difference between IRB-H and IRB, as well as the difference between RB-H and RB, is that we used the relax plan length heuristic as prior knowledge for the former methods. In Figure 4, it is shown that the relax plan length heuristic did help to significantly improve performance in some domains. For Driverlog, FreeCell, Pipesworld and Pipesworld-with-tankage, RB-H and IRB-H always solve more problems than RB and IRB, with the same number of rules induced. In Depots, RB-H achieves similar performance as RB but IRB-H solves many more problems than IRB. Blocksworld is a domain where IRB and RB works slightly better than IRB-H and RB-H, with better solution length. In Philosopher, IRB-H and RB-H find better solution paths. Overall, there is clear value in using the relaxed-plan length heuristic as prior knowledge.

Summary and Future Work

In this paper, we developed a robust way for using action selection rules for solving planning problems. Based on prior work that learns action selection rules, we formulated a ranking problem and applied a boosting-style algorithm to solve it. The experimental results show that the learned control knowledge significantly improves the robustness of rules over reactive policy. Furthermore, it outperforms both the state-of-the-art relax plan length heuristic and automatically learned heuristics from recent work.

While initial results are promising, this approach still has a lot of space to improve. One important direction is to extend this approach to breadth-first beam search, which allows for more search and possibly more robustness. Another direction is to investigate the rule learner, finding the reason why it fails to induce new unique rules for some planning domains. Also of the interest is to study the convergence of our iterative learning algorithm. Finally, we plan to consider

	Learning iterations	Number of unique rules				Problems solved (Median plan length)			
		RB	RB-H	IRB	IRB-H	RB	RB-H	IRB	IRB-H
Blocksworld	5	3	4	5	5	30(133)	30(151)	**30(125)**	30(160)
	10	7	8	8	10	30(126)	30(166)	**30(89)**	30(118)
Depots	5	4	5	5	4	0(-)	2(8631)	0(-)	**3(115)**
	10	7	9	8	8	3(9194)	4(954)	0(-)	**20(796)**
	15	9	13	10	9	5(5372)	2(113)	0(-)	**16(313)**
	20	11	17	12	12	2(5193)	7(263)	0(-)	**23(433)**
	25	13	20	14	14	3(3188)	5(678)	0(-)	**22(349)**
	30	16	24	16	15	15(661)	11(129)	0(-)	**19(314)**
Driverlog	5	4	5	4	5	0(-)	1(1893)	0(-)	**3(8932)**
	10	6	5	5	8	0(-)	**3(2852)**	0(-)	1(2818)
	15	7	6	6	10	0(-)	0(-)	0(-)	**3(544)**
	20	8	8	6	10	0(-)	1(4309)	0(-)	**4(544)**
	25	9	9	6	10	0(-)	3(4082)	0(-)	**4(544)**
	30	10	11	6	10	0(-)	1(632)	0(-)	**3(544)**
FreeCell	5	2	4	2	5	2(213)	6(104)	2(213)	**9(94)**
	10	4	7	2	5	2(186)	5(112)	2(213)	**7(95)**
	15	7	10	2	6	3(103)	6(143)	2(213)	**9(94)**
	20	11	15	2	6	5(155)	7(96)	2(213)	**9(94)**
	25	11	19	2	6	3(334)	5(90)	2(213)	**9(92)**
Pipesworld	5	3	4	3	3	4(382)	**17(1063)**	7(1572)	16(279)
	10	7	7	7	7	2(7845)	8(1821)	2(335)	**11(307)**
	15	9	9	9	9	5(2929)	12(1599)	2(335)	**17(595)**
	20	9	12	10	13	3(1369)	11(1423)	5(511)	**17(579)**
	25	11	16	11	15	4(998)	11(2561)	6(883)	**17(595)**
	30	12	19	11	18	7(1360)	12(1423)	6(990)	**15(366)**
Pipesworld-with-tankage	5	5	5	4	4	0(-)	3(296)	3(2006)	**4(126)**
	10	6	8	5	9	0(-)	3(100)	1(5372)	**4(148)**
	15	9	9	5	10	0(-)	**4(98)**	1(4735)	4(134)
	20	12	12	7	10	1(1383)	**6(152)**	1(4735)	5(350)
	25	16	17	7	10	1(1383)	4(449)	3(2940)	**5(206)**
Philosopher	5	3	3	3	3	0(-)	**33(363)**	0(-)	**33(363)**
	10	3	3	4	4	0(-)	**33(363)**	33(875)	0(-)
	15	4	5	4	4	0(-)	**33(363)**	33(875)	0(-)
	20	5	5	5	4	33(875)	**33(363)**	33(875)	0(-)

Figure 4: Experimental results for different planners and different weighted rule sets. For each domain, we show the number of unique rules that are learned after the corresponding number of learning iterations. The performance of each learned rule set is given by the number of solved problems, together with the median plan length of the solved problems. A dash in the table indicates that the median plan length is not available while none of the problems can be solved.

more general ranking functions, e.g. a ranking function that combines weighted rule sets with other features that are defined on states.

References

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proceedings of the European Conference on Planning*, 360–372.

Freund, Y.; Iyer, R.; Schapire, R. E.; and Singer, Y. 2003. An efficient boosting algorithm for combining preferences. *Journal of Machine Learning Research* 4:933–969.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:263–302.

Khardon, R. 1999. Learning action strategies for planning domains. *Artificial Intelligence* 113:125–148.

Martin, M., and Geffner, H. 2000. Learning generalized policies in planning domains using concept languages. In *Proceedings of the 7th International Conference on Knowledge Representation and Reasoning*.

Slaney, J., and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence* 125:119–153.

Veloso, M. M.; Pérez, M. A.; and Carbonell, J. G. 1991. Nonlinear planning with parallel resource allocation. In *Workshop on Innovative Approaches to Planning, Scheduling and Control*, 207–212.

Xu, Y.; Fern, A.; and Yoon, S. 2009. Learning linear ranking functions for beam search with application to planning. *Journal of Machine Learning Research* 10:1349–1388.

Yoon, S.; Fern, A.; and Givan, R. 2002. Inductive policy selection for first-order MDPs. In *In Proceedings of Eighteenth Conference in Uncertainty in Artificial Intelligence*.

Yoon, S.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *Journal of Machine Learning Research* 9:683–718.