Journal of Machine Learning Research ? (????) ?-?? Submitted ?/??; Published ??/??

Learning Control Knowledge for Forward Search Planning

Sungwook Yoon SUNGWOOK.YOON@ASU.EDU
Computer Science and Engineering Department, Arizona State University
Alan Fern AFERN@EECS.ORST.EDU
School of Electrical Engineering and Computer Science, Oregon State University
Robert Givan GIVAN@PURDUE.EDU
School of Electrical and Computer Engineering, Purdue University

Editor: Leslie Pack Kaelbling

Abstract

A number of today's state-of-the-art planners are based on forward state-space search. The impressive performance can be attributed to progress in computing domain independent heuristics that perform well across many domains. However, it is easy to find domains where such heuristics provide poor guidance, leading to planning failure. Motivated by such failures, the focus of this paper is to investigate mechanisms for learning domain-specific knowledge to better control forward search in a given domain. While there has been a large body of work on inductive learning of control knowledge for AI planning, there is a void of work aimed at forward-state-space search. One reason for this may be that it is challenging to specify a knowledge representation for compactly representing important concepts across a wide range of domains. One of the main contributions of this work is to introduce a novel feature space for representing such control knowledge. The key idea is to define features in terms of information computed via relaxed plan extraction, which has been a major source of success for non-learning planners. This gives a new way of leveraging relaxed planning techniques in the context of learning. Using this feature space, we describe three forms of control knowledge—reactive policies (decision list rules and measures of progress) and linear heuristics—and show how to learn them and incorporate them into forward state-space search. Our empirical results show that our approaches are able to surpass state-of-the-art non-learning planners across a wide range of planning competition domains.

Keywords: Planning, Machine Learning, Knowledge Representation, Search

1. Introduction

In recent years, forward state-space search has become a popular approach in AI planning, leading to state-of-the-art performance in a variety of settings, including classical STRIPS planning (Hoffmann and Nebel, 2001; Vidal, 2004), optimal planning (Helmert et al., 2007), temporal-metric planning (Do and Kambhampati, 2003), nondeterministic planning (Bryce and Kambhampati, 2006), and oversubscribed planning (Benton et al., 2006), among others. Given that forward state-space search is one of the oldest techniques available for planning, and many other search spaces and approaches have been developed, this state-of-the-art performance is somewhat surprising. One of the key reasons for the success is the development of powerful domain-independent heuristics that work well on many AI planning domains. Nevertheless, it is not hard to find domains where these heuristics do not work well, resulting in planning failure. We are motivated by these failures and

in this study, we investigate machine learning techniques that find domain-specific control knowledge that can improve or speed-up forward state-space search in a non-optimal, or satisficing, planning setting.

As outlined in Section 3 there is a large body of work on learning search-control knowledge for AI planning domain. However, despite the significant effort, none of these approaches has been demonstrated to be competitive with state-of-the-art non-learning planners across a wide range of planning domains. There are at least two reasons for the performance gap between learning and non-learning planners. First, most prior work on learning control knowledge has been in the context of non-state-of-the-art planning approaches such as partial-order planning, means-ends analysis, among others. In fact, we are only aware of two recent efforts (Botea et al., 2005; Coles and Smith, 2007) that learn control knowledge for forward state-space search planners. Even these approaches have not demonstrated the ability to outperform the best non-learning planners as measured on planning competition domains. Second, it is a challenge to define a hypothesis space for representing control knowledge that is both rich enough for a wide variety of planning domains, yet compact enough to support efficient and reliable learning. Indeed, a common shortcoming of much of the prior work is that the hypothesis spaces, while adequate for the small number of domains investigated, were not rich enough for many other domains.

The primary goal of this work is to contribute toward reversing the performance gap between learning and non-learning planners. In this work, we do this by addressing each of the above two issues. First, our system is based on the framework of forward state-space search, in particular, being built upon the state-of-the-art planner FF (Hoffmann and Nebel, 2001). Second, we propose a novel hypothesis space for representing useful heuristic features of planning states. We show how to use this feature space as a basis for defining and learning several forms of control knowledge that can be incorporated into forward state-space search. The result is a learning-based planner that learns control knowledge for a planning domain from a small number of solved problems and is competitive with and often better than state-of-the-art non-learning planners across a substantial set of benchmark domains and problems.

A key novelty of our proposed feature space is that it leverages the computation of relaxed plans, which are at the core of the computation of modern forward-search heuristics (Bonet and Geffner, 2001; Hoffmann and Nebel, 2001). Relaxed plans are constructed by ignoring, to varying degrees, the delete/negative effects of actions and can be computed very efficiently. The length of these plans can then serve as an informative heuristic (typically non-admissible) for guiding state-space search. In addition to their length, relaxed plans contain much more information about a search state that is ignored by most forward-search planners.[1] Our proposed feature space gives one way of using this information by viewing the relaxed plan as a structure for defining potentially useful features of the current state. As an example of the utility of our feature space, note that the fact that relaxed planning ignores delete effects is the main reason that the length sometimes dramatically underestimates the true distance to goal, leading to poor heuristic guidance (note that relaxed-plan length can also overestimate the distance to goal). Our feature space is able to partially capture information about the delete effects ignored in a relaxed plan, which can be used to learn knowledge that partially compensates for the underestimation.

We use the relaxed-plan feature space to learn two forms of knowledge for controlling forward state-space planning. In each case, the knowledge is learned based on a set of training problems from a domain, each labeled by a solution. First, we consider learning knowledge in the form of linear heuristic functions. In particular, we learn heuristics that are linear combinations of relaxed-plan features, with one of those features being the

1. One exception is Vidal (2004) where relaxed plans are also used to extract macro actions.

relaxed-plan length. Thus, our heuristic learner can be viewed as an approach for automatically correcting deficiencies of the usual relaxed-plan length heuristic, by augmenting it with a weighted combination of additional features selected from the large space of possible relaxed-plan features.

As a second form of control knowledge, we investigate reactive policies. Learning reactive policies for planning domains has been studied by several researchers (Khardon, 1999; Martin and Geffner, 2000; Yoon et al., 2002, 2005). However, all of these studies have used the learned policies as stand-alone search-free planners that simply execute the linear sequence of actions selected by the policies. While this non-search approach is efficient and has been shown to work well in a number of domains, it often fails due to flaws in the policy that arise due to imperfect learning. Nevertheless, such policies capture substantial information about the planning domain, which we would like to exploit in a more robust way. In this work, we propose and evaluate a simple way of doing this by integrating learned policies into forward search. At each search node (i.e. state), we execute the learned policy for a fixed horizon and add all of the states encountered to the search queue. In this way, flaws in the policy can be overcome by search, while the search efficiency can be substantially improved by quickly uncovering good "deep states" that are found by the policy. We evaluate this idea using two representations for learned policies both of which make use of relaxed-plan features—decision lists of rules similar to Yoon et al. (2002) and measures of progress (Yoon et al., 2005)—both of which make use of relaxed-plan features.

In our experiments, we learned and evaluated both forms of control knowledge on benchmark problems from recent planning competitions. We learned on the first 15 problems in each domain and tested on the remaining problems. The results are very much positive. Forward state-space search with the learned control knowledge outperforms state-of-the-art planners, in most of the competition domains. We also demonstrate the utility of our relaxed-plan feature space by considering feature spaces that ignore parts of the relaxed-plan information, showing that using the relaxed-plan information leads to the best performance.

The remainder of the paper is structured as follows. In Section 2, we introduce the problem of learning domain-specific control knowledge for planning and in Section 3 we overview some of the prior work in this area. In Section 4, we describe the two general forms of control knowledge, heuristics and reactive policies, that we consider learning in this work and how we will use that knowledge to guide forward state-space search. In Section 5, we will describe a relaxed-plan feature space that will serve as our basis for representing both learned heuristics and policies. In Sections 6 and 7 we will describe specific representations for policies and heuristics, in terms of the relaxed-plan features, and give learning algorithms for these forms of control knowledge. In Section 8, we demonstrate the effectiveness of our proposal in this study through empirical results on benchmark planning domains. In Section 9, we summarize and discuss potential future extensions to this work.

2. Problem Setup

In this paper, we focus our attention on learning control knowledge for deterministic STRIPS planning domains. Below we first give a formal definition of the types of planning domains we consider and then describe the learning problem.

2.1 Planning Domains

A deterministic planning domain $\mathcal{D}$ defines a set of possible actions $\mathcal{A}$ and a set of states $\mathcal{S}$ in terms of a set of predicate symbols $\mathcal{P}$, action types Y, and objects O. Each state

in $\mathcal{S}$ is a set of facts, where a fact is an application of a predicate symbol in $\mathcal{P}$ to the appropriate number of objects from O. There is an action in $\mathcal{A}$, for each way of applying the appropriate number of objects in O to an action type symbol in Y. Each action $a \in \mathcal{A}$ consists of: 1) an action name, which is an action type applied to the appropriate number of objects, 2) a set of precondition state facts $\text{Pre}(a)$, 3) two sets of state facts $\text{Add}(a)$ and $\text{Del}(a)$ representing the add and delete effects respectively. As usual, an action a is applicable to a state s iff $\text{Pre}(a) \subseteq s$, and the application of an (applicable) action a to s, denoted $a(s)$, results in the new state $a(s) = (s \setminus \text{Del}(a)) \cup \text{Add}(a)$.

Given a planning domain, a planning problem $\mathbb{P}$ from the domain is a tuple (s, A, g), where $A \subseteq \mathcal{A}$ is a set of applicable actions, $s \in \mathcal{S}$ is the initial state, and g is a set of state facts representing the goal. A solution plan for a planning problem is a sequence of actions $(a_1, \ldots, a_h)$, where the sequential application of the sequence starting in state s leads to a goal state s' where $g \subseteq s'$. Later in the paper, in Section 7, when discussing measures of progress, it will be useful to talk about reachability and deadlocks. We say that a planning problem (s, A, g) is reachable from problem (s_0, A, g) iff there is some action sequence in A^* that leads from s_0 to s. We say that a planning problem $\mathbb{P}$ is deadlock free iff all problems reachable from $\mathbb{P}$ are solvable.

2.2 Learning Control Knowledge from Solved Problems

The bi-annual International Planning Competition (IPC), has played a large role in the recent progress observed in AI planning. Typically the competition is organized around a set of planning domains, with each domain providing a sequence of planning problems, often in increasing order of difficulty. Despite the fact that the planners in these competitions experience many similar problems from the same domain, to our knowledge only one of them, Macro-FF (Botea et al., 2005), has made any attempt to learn from previous experience in a domain to improve performance on later problems.[2] Rather they solve each problem as if it were the first time the domain had been encountered. The ability to effectively transfer domain experience from one problem to the next would provide a tremendous advantage. Indeed, the potential benefit of learning domain-specific control knowledge can be seen by the impressive performance of planners such as TL Plan (Bacchus and Kabanza, 2000) and SHOP (Nau et al., 1999), where human-written control knowledge is provided for each domain. However, to date, most "learning to plan" systems have lagged behind the state-of-the-art non-learning domain-independent planners. One of the motivations and contributions of this work is to move toward reversing that trend.

The input to our learner will be a set of problems for a particular planning domain along with a solution plan to each problem. The solution plan might have been provided by a human or automatically computed using a domain-independent planner (for modestly sized problems). The goal is to analyze the training set to extract control knowledge that can be used to more effectively solve new problems from the domain. Ideally, the control knowledge allows for the solutions of large, difficult problems that could not be solved within a reasonable time limit before learning.

As a concrete example of this learning setup, in our experiments, we use the problem set from recent competition domains. We first use a domain-independent planner, in our case FF (Hoffmann and Nebel, 2001), to solve the low-numbered planning problems in the set (typically corresponding to the easier problems). The solutions are then used by our learner to induce control knowledge. The control knowledge is then used to solve the remaining, typically more difficult, problems in the set. Our objective here is to obtain fast, satisficing

2. Macro-FF learned on training problems from each domain provided by the organizers before the competition, rather than learning during the actual competition itself.

planning through learning. The whole process is domain independent, with knowledge transferring from easy to hard problems in a domain. Note that although learning times can be substantial, the learning cost can be amortized over all future problems encountered in the domain.

3. Prior Work

There has been a long history of work on learning-to-plan, originating at least back to the original STRIPS planner (Fikes et al., 1972), which learned triangle tables or macros that could later be exploited by the planner. For a collection and survey of work on learning in AI planning see Minton (1993) and Zimmerman and Kambhampati (2003).

A number of learning-to-plan systems have been based on the explanation-based learning (EBL) paradigm, e.g. Minton et al. (1989) among many others. EBL is a deductive learning approach, in the sense that the learned knowledge is provably correct. Despite the relatively large effort invested in EBL research, the best approaches typically did not consistently lead to significant gains, and even hurt performance in many cases. A primary way that EBL can hurt performance is by learning too many, overly specific control rules, which results in the planner spending too much time simply evaluating the rules at the cost of reducing the number of search nodes considered. This problem is commonly referred to as the EBL utility problem (Minton, 1988).

Partly in response to the difficulties associated with EBL-based approaches, there have been a number of systems based on inductive learning, perhaps combined with EBL. The inductive approach involves applying statistical learning mechanisms in order to find common patterns that can distinguish between good and bad search decisions. Unlike EBL, the learned control knowledge does not have guarantees of correctness, however, the knowledge is typically more general and hence more effective in practice. Some representative examples of such systems include learning for partial-order planning (Estlin and Mooney, 1996), learning for planning as satisfiability (Huang et al., 2000), and learning for the Prodigy means-ends framework (Aler et al., 2002). While these systems typically showed better scalability than their EBL counterparts, the evaluations were typically conducted on only a small number of planning domains and/or small number of test problems. There is no empirical evidence that such systems are robust enough to compete against state-of-the-art non-learning planners across a wide range of domains.

More recently there have been several learning-to-plan systems based on the idea of learning reactive policies for planning domains (Khardon, 1999; Martin and Geffner, 2000; Yoon et al., 2002). These approaches use statistical learning techniques to learn policies, or functions, that map any state-goal pair from a given domain to an appropriate action. Given a good reactive policy for a domain, problems can be solved quickly, without search, by iterative application of the policy. Despite its simplicity, this approach has demonstrated considerable success. However, these approaches have still not demonstrated the robustness necessary to outperform state-of-the-art non-learning planners across a wide range of domains.

Ideas from reinforcement learning have also been applied to learn control policies in AI planning domains. Relational reinforcement learning (RRL) (Dzeroski et al., 2001), utilized Q-learning with a relational function approximator, and demonstrated good empirical results in the Blocksworld. The Blocksworld problems they considered were complex from a traditional RL perspective due to the large state and action spaces, however, they were relatively simple from an AI planning perspective. This approach has not yet shown scalability to the large problems routinely tackled by today's planners. A related approach, utilized a more powerful form of reinforcement learning, known as approximate policy iteration, and demonstrated good results in a number of planning competition domains (Fern

et al., 2006). Still the approach failed badly on a number of domains and overall does not yet appear to be competitive with state-of-the-art planners on a full set of competition benchmarks.

The most closely related approaches to ours are recent systems for learning in the context of forward state-space search. Macro-FF (Botea et al., 2005) and Marvin (Coles and Smith, 2007) learn macro action sequences that can then be used during forward search. Macro-FF learns macros from a set of training problems and then applies them to new problems. Rather, Marvin is an online learner in the sense that it acquires macros during search in a specific problem that are applied at later stages in the search. As evidenced in the recent planning competitions, however, neither system dominates the best non-learning planners.

Finally, we note that researchers have also investigated domain-analysis techniques, e.g. (Gerevini and Schubert, 2000; Fox and Long, 1998), which attempt to uncover structure in the domain by analyzing the domain definition. These approaches have not yet demonstrated the ability to improve planning performance across a range of domains.

4. Control Knowledge for Forward State-Space Search

In this section, we describe the two general forms of control knowledge that we will study in this work: heuristic functions and reactive policies. For each, we describe how we will incorporate them into forward state-space search in order to improve planning performance. Later in the paper, in Sections 6 and 7, we will describe specific representations for heuristics and policies and give algorithms for learning them from training data.

4.1 Heuristic Functions

The first and most traditional forms of control knowledge we consider are heuristic functions. A heuristic function $H(s, A, g)$ is simply a function of a state s, action set A, and goal g that estimates the cost of achieving the goal from s using actions in A. If a heuristic is accurate enough, then greedy application of the heuristic will find the goal without search. However, when a heuristic is less accurate, it must be used in the context of a search procedure such as best-first search, where the accuracy of the heuristic impacts the search efficiency. In our experiments, we will utilize best-first search, which has often been demonstrated to be an effective, though sub-optimal, search strategy in forward state-space planning. Note that by best-first search, here we mean a search that is guided by only the heuristic value, rather than the path-cost plus heuristic value. This search is also called *greedy* best-first search. In this paper, when we use the term best-first search, it means greedy best-first search and we will add *greedy* in front of best-first search to remind the readers, as necessary, in the following texts.

Recent progress in the development of domain-independent heuristic functions for planning has led to a new generation of state-of-the-art planners based on forward state-space heuristic search (Bonet and Geffner, 2001; Hoffmann and Nebel, 2001; Nguyen et al., 2002). However, in many domains these heuristics can still have low accuracy, e.g. significantly underestimating the distance to goal, resulting in poor guidance during search. In this study, we will attempt to find regular pattern of heuristic inaccuracy (either due to over or under estimation) through machine learning and compensate the heuristic function accordingly.

We will focus our attention on linear heuristics that are represented as weighted linear combinations of features, i.e. $H(s, A, g) = \Sigma_i w_i \cdot f_i(s, A, g)$, where the w_i are weights and the f_i are functions. In particular, for each domain we would like to learn a distinct set of features and their corresponding weights that lead to good planning performance in that domain. Note that some of the feature functions can correspond to existing domain-

independent heuristics, allowing for our learned heuristics to exploit the useful information they already provide, while overcoming deficiencies by including additional features. The representation that we use for features is discussed in Section 5 and our approach to learning linear heuristics over those features is described in Section 6.

In all of our experiments, we use the learned heuristics to guide (greedy) best-first search when solving new problems.

4.2 Reactive Policies in Forward Search

The second general form of control knowledge that we consider in this study is of reactive policies. A reactive policy is a computationally efficient function $\pi(s, A, g)$, possibly stochastic, that maps a planning problem (s, A, g) to an action in A. Given an initial problem (s_0, A, g), we can use a reactive policy π to generate a *trajectory* of pairs of problems and actions $(((s_0, A, g), a_0), ((s_1, A, g), a_1), ((s_2, A, g), a_2) \ldots)$, where $a_i = \pi(s_i, A, g)$ and $s_{i+1} = a_i(s_i)$. Ideally, given an optimal or near-optimal policy for a planning domain, the trajectories represent high-quality solution plans. In this sense, reactive policies can be viewed as efficient domain-specific planners that avoid unconstrained search. Later in the paper, in Section 7, we will introduce two formal representations for policies: decision rule lists and measures of progress, and describe learning algorithms for each representation. Below we describe some of the prior approaches to using policies to guide forward search and the new approach that we propose in this work.

The simplest approach to using a reactive policy as control knowledge is to simply avoid search altogether and follow the trajectory suggested by the policy. There have been a number of studies (Khardon, 1999; Martin and Geffner, 2000; Yoon et al., 2002, 2005; Fern et al., 2006) that consider using learned policies in this way in AI planning context. While there have been some positive results, for many planning domains the results have been mostly negative. One reason for these failures is that inductive, or statistical, policy learning can result in imperfect policies, particularly with limited training data. Although these policies may select good actions in many states, the lack of search prevents them from overcoming the potentially numerous bad action choices.

In an attempt to overcome the brittleness of simply following imperfect reactive policies, previous researchers have considered more sophisticated methods of incorporating imperfect policies into search. Two such methods include discrepancy search (Harvey and Ginsberg, 1995) and policy rollout (Bertsekas and Tsitsiklis, 1996). Unfortunately, our initial investigation showed that in many planning competition domains these techniques were not powerful enough to overcome the flaws in our learned polices. With this motivation we developed a novel approach that is easy to implement and has proven to be quite powerful.

The main idea is to use reactive policies during the node expansion process of a heuristic search, which in our work is greedy best-first search. Typically in best-first search, only the successors of the current node being expanded are added to the priority queue, where priority is measured by heuristic value. Rather, our approach first executes the reactive policy for h steps from the node being expanded and adds the nodes of the trajectory along with their neighbors to the queue. In all of our experiments, we used a value of $h = 50$, though we found that the results were quite stable across a range of h (we sampled a range from 30 to 200).

Note that when $h = 0$ we get standard (greedy) best-first search. In cases, where the policy can solve a given problem from the current node being expanded, this approach will solve the problem without further search provided that h is large enough. Otherwise, when the policy does not directly lead to the goal, it may still help the search process by putting heuristically better nodes in the search queue in a single node expansion. Without the policy (i.e. $h = 0$) such nodes would only appear in the queue after many node expansions.

Intuitively, given a reasonably good heuristic, this approach is able to leverage the good choices made by a policy, while overcoming the flaws. While this technique for incorporating policies into search is simple, our empirical results, show that it is very effective, achieving better performance than either pure heuristic search or search-free policy execution.

5. A Relaxed-Plan Feature Space

A key challenge toward learning control knowledge in the form of heuristics and policies is to develop specific representations that are rich enough to capture important properties of search nodes. In this section, we describe a novel feature space for representing such properties. This feature space will be used as a basis for our policy and heuristic representations described in Sections 6 and 7. Note that throughout, for notational convenience we will describe each search node by its implicit planning problem (s, A, g), where s is the current state of the node, g is the goal, and A is the action set.

Each feature in our space is represented via an expression in taxonomic syntax, which as described in Section 5.4, provides a language for describing sets of objects with common properties. Given a search node (s, A, g) and a taxonomic expression C, the value of the corresponding feature is computed as follows. First, a database of atomic facts $D(s, A, g)$ is constructed, as described in Section 5.3, which specifies basic properties of the search node. Next, we evaluate the class expression C relative to $D(s, A, g)$, resulting in a class or set of objects. These sets, or features, can then be used as a basis for constructing control knowledge in various ways—e.g. using the set cardinalities to define a numeric feature representation of search nodes.

Our feature space, is in the spirit of prior work (Martin and Geffner, 2000; Yoon et al., 2002; Fern et al., 2006) that also used taxonomic syntax to represent control knowledge. However, our approach is novel in that we construct databases $D(s, A, g)$ that contain not only facts about the current state and goal, but also facts derived via a bounded reasoning process known as relaxed planning. Prior work, considered only databases that included information about the current state and goal. By defining features in terms of taxonomic expressions built from our extended databases, we are able to capture properties that are difficult to represent in terms of the state and goal predicates alone.

In the remainder of this section, we first review the idea of relaxed planning, which is central to our feature space. Next, we describe the construction of the database $D(s, A, g)$ for search nodes. Finally, we introduce taxonomic syntax, which is used to build complex features on top of the database.

5.1 Relaxed Plans

Given a planning problem (s, A, g), we define the corresponding relaxed planning problem to be the problem (s, A^+, g) where the new action set A^+ is created by copying A and then removing the delete list from each of the actions. Thus, a relaxed planning problem is a version of the original planning problem where it is not necessary to worry about delete effects of actions. A relaxed plan for a planning problem (s, A, g) is simply a plan that solves the relaxed planning problem.

Relaxed planning problems have two important characteristics. First, although a relaxed plan may not necessarily solve the original planning problem, the length of the shortest relaxed plan serves as an admissible heuristic for the original planning problem. This is because preconditions and goals are defined in terms of positive state facts, and hence removing delete lists can only make it easier to achieve the goal. Second, in general, it is computationally easier to find relaxed plans compared to solving general planning problems. In the worst case, this is apparent by noting that the problem of plan existence

can be solved in polynomial time for relaxed planning problems, but is PSPACE-complete for general problems. However, it is still NP-hard to find minimum-length relaxed plans (Bylander, 1994). Nevertheless, practically speaking, there are very fast polynomial time algorithms that typically return short relaxed plans whenever they exist, and the lengths of these plans, while not admissible, often provide good heuristics.

The above observations have been used to realize a number of state-of-the-art planners based on heuristic search. HSP (Bonet and Geffner, 2001) uses forward state-space search guided by an admissible heuristic that estimates the length of the optimal relaxed plan. FF (Hoffmann and Nebel, 2001) also takes this approach, but unlike HSP, estimates the optimal relaxed-plan length by explicitly computing a relaxed plan. FF's style of relaxed plan computation is linear with the length of the relaxed plan, thus fast, but the resulting heuristics can be inadmissible. Our work builds on FF, using the same relaxed-plan construction technique, which we briefly describe below.

FF computes relaxed plans using a relaxed plan graph (RPG). An RPG is simply the usual plan graph created by Graphplan (Blum and Furst, 1995), but for the relaxed planning problem rather than the original problem. Since there are no delete lists in the relaxed plan, there will be no mutex relations in the plan graph. The RPG is a leveled graph alternating between action levels and state-fact levels, with the first level containing the state facts in the initial state. An action level is created by including any action whose preconditions are satisfied in the preceding state-fact level. A state-fact level is created by including any fact that is in the previous fact level or in the add list of an action in the preceding action level. RPG construction stops when a fixed point is reached or the goal facts are all contained in the most recent state-fact level. After constructing the RPG for a planning problem, FF starts at the last RPG level and uses a backtrack-free procedure that extracts a sequence of actions that correspond to a successful relaxed plan. All of this can be done very efficiently, allowing for fast heuristic computation.

While the length of FF's relaxed plan often serves as an effective heuristic, for a number of planning domains, ignoring delete effects leads to severe underestimates of the distance to a goal. The result is poor guidance and failure on all but the smallest problems. One way to overcome this problem would be to incorporate partial information about delete lists into relaxed plan computation, e.g., by considering mutex relations. However, to date, this has not born out as a practical alternative. Another possibility is to use more information about the relaxed plan than just its length. For example, Vidal (2004) uses relaxed plans to construct macro actions, which help the planner overcome regions of the state space where the relaxed-plan length heuristic is flat. However, that work still uses length as the sole heuristic value. In this work, we give a novel approach to leveraging relaxed planning, in particular, we use relaxed plans as a source of information from which we can compute complex features that will be used to learn heuristic functions and policies. Interestingly, as we will see, this approach will allow for features that are sensitive to delete lists of actions in relaxed plans, which can be used to help correct for the fact that relaxed plans ignore delete effects.

5.2 Example of using Relaxed Plan Features for Learning Heuristics

As an example of how relaxed plans can be used to define useful features, consider a problem from the Blocksworld in Figure 1. Here, we show two states S_1 and S_2 that can be reached from the initial state by applying the actions putdown(A) and stack(A, B) respectively. From each of these states we show the optimal relaxed plans for achieving the goal. For these states, the relaxed-plan length heuristic is 3 for S_2 and 4 for S_1, suggesting that S_2 is the better state. However, it is clear that, in fact, S_1 is better.

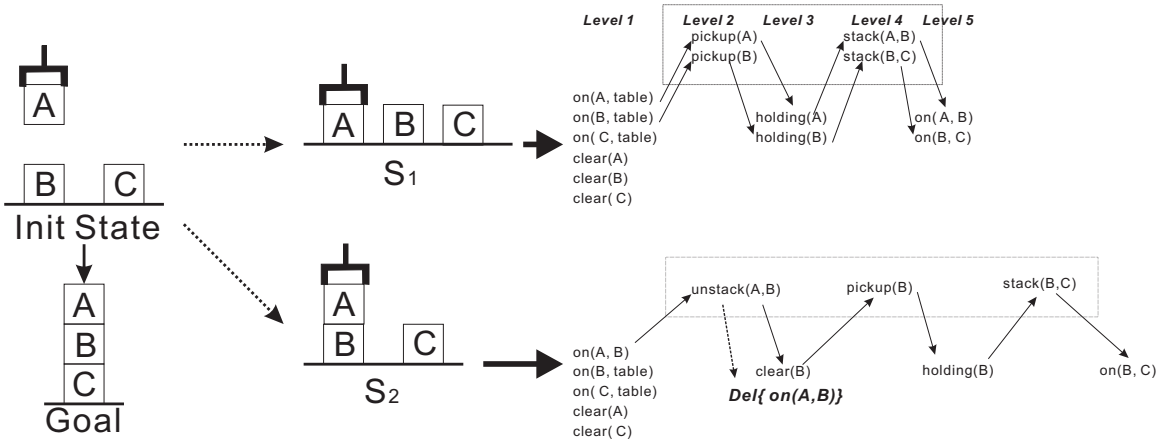

Figure 1: Blocksworld Example

Notice that in the relaxed plan for S_2, on(A, B) is in the delete list of the action unstack(A, B) and at the same time it is a goal fact. One can improve the heuristic estimation by adding together the relaxed-plan length and a term related to such deleted facts. In particular, suppose that we had a feature that computed the number of such "on" facts that were both in the delete list of some relaxed plan action and in the goal, giving a value of 0 for S_1 and 1 for S_2. We could then weight this feature by two and add it to the relaxed-plan length to get a new heuristic. This would assign a value of 4 for S_1 and 5 for S_2, correctly ranking the states. Using taxonomic syntax, one can define such a feature as the cardinality of a certain class expression over a database of facts defined in the next section.

While this is an over-simplified example, it is suggestive as to the utility of features derived from relaxed plans. Below we describe a domain-independent feature space that can be instantiated for any planning domain. Our experiments show that these features are useful across a range of domains used in planning competitions.

5.3 Constructing Databases from Search Nodes

Recall that each feature in our feature space corresponds to a taxonomic syntax expression (see next section) built from the predicate symbols in databases of facts constructed for each search node encountered. We will denote the database for search node (s, A, g) as $D(s, A, g)$, which will simply contain a set of ground facts over some set of predicate symbols and objects derived from the search node. Whereas prior work defined $D(s, A, g)$ to include only facts about the goal g and state s, we will also include facts about the relaxed plan corresponding to problem (s, A, g), denoted by $(a_1, \ldots, a_n)$. Note that in this work we use the relaxed plan computed by FF's heuristic calculation.

Given any search node (s, A, g) we now define $D(s, A, g)$ to be the database that contains the following facts:

- All of the state facts in s.
- The name of each action a_i in the relaxed plan. Recall that each name is an action type Y from domain definition $\mathcal{D}$ applied to the appropriate number of objects, e.g., unstack(A, B).

on(A, B), on(B, table), on(C, table), clear(A), clear(C), armempty()
unstack(A, B), pickup(B), stack(B, C)
aholding(A), **a**clear(B), **a**holding(B), **a**on(B, C)
don(A, B), **d**armempty(), **d**armempty(), **d**on(B, table), **d**holding(B), **d**clear(C)
gon(A, B), **g**on(B, C)
con(A, B), **c**on(C,table)

Figure 2: Database for state S_2 in Figure 1

- For each state fact in the add list of some action a_i in the relaxed plan, add a fact to the database that is the result of prepending an **a** to the fact's predicate symbol. For example, in Figure 1, for state S_2 the fact holding(B) is in the add list of pickup(B), and thus we would add the fact **a**holding(B) to the database.
- Likewise for each state fact in the delete list of some a_i, we prepend a **d** to the predicate symbol and add the resulting fact to the database. For example, in Figure 1, for S_2, we would add the fact **d**on(A, B).
- For each state fact in the goal g, we prepend a **g** to the predicate symbol and add it to the database. For example, in Figure 1, we would add the facts **g**on(A, B) and **g**on(B, C).
- For each predicate symbol that appears in the goal, we prepend a **c** to the predicate symbol and add a corresponding fact to the database whenever it is true in the current state s and appears in the goal. For example, the predicate **c**on represents the relation "correctly on", and in Figure 1, for state S_2 we would add the fact **c**on(A,B) since A is currently on B and is supposed to be on B in the goal. The '**c**' predicates provide a useful mechanism for expressing concepts that relate the current state to the goal.

Figure 2, shows an example of the database that would be constructed for the state S_2 in Figure 1. Note that taxonomic syntax class expressions will be constructed from the predicates in this database which include the set of planning domain predicates and action types, along with a variant of each planning domain predicate prepended with an '**a**', '**d**', '**g**', or '**c**'. The database captures information about the state and goal using the planning domain predicates, the '**g**' predicates, and '**c**' predicates. It captures information about the relaxed plan using the action type predicates and the '**a**' and '**d**' predicates. Notice that the database does not capture information about the temporal structure of the relaxed plan. Such temporal information may be useful for describing features, and is a natural extension of this work.

5.4 Defining Complex Features with Taxonomic Syntax

For a given search node (s, A, g) with database $D(s, A, g)$ we now wish to define more complex features of the search node. We will do this using taxonomic syntax (McAllester and Givan, 1993) which is a first-order language for writing class expressions C that are used to denote sets of objects. In particular, given a class expression C, the set of objects it represents relative to $D(s, A, g)$ will be denoted by $C[D(s, A, g)]$. Thus, each C can be viewed as defining a feature that for search node (s, A, g) takes the value $C[D(s, A, g)]$. Below we describe the syntax and semantics of the taxonomic syntax fragment we use in this paper.

Syntax. Taxonomic class expression are built from a set of predicates $\mathcal{P}$, where $n(P)$ will be used to denote the arity of predicate P. For example, in our application the set of

predicates will include all predicates used to specify facts in database $D(s, A, g)$ as described above. The set of possible class expressions over $\mathcal{P}$ are given by the following grammar:

$$C := \textbf{a-thing} \mid P_1 \mid C \cap C \mid \neg C \mid (P \;\; C_1 \ldots C_{i-1} \;\; ? \;\; C_{i+1} \ldots C_{\mathrm{n}(P)})$$

where C and C_j are class expressions, P_1 is any arity one predicate, and P is any predicate symbol of arity two or greater. Given this syntax we see that the primitive class expressions are the special symbol **a-thing**, which will be used to denote the set of all objects, and single arity predicates, which will be used to denote the sets of objects for which the predicates are true. One can then obtain compound class expressions via complementation, intersection, or relational composition (the final rule). Before defining the formal semantics of class expressions, we introduce the concept of depth, which will be used in our learning procedures. We define the depth of a class expression C, denoted $\mathrm{depth}(C)$ as follows. The depth of **a-thing** or a single arity predicate is 0, $\mathrm{depth}(C_1 \cap C_2) = 1 + \max(\mathrm{depth}(C_1), \mathrm{depth}(C_2))$, $\mathrm{depth}(\neg C) = 1 + \mathrm{depth}(C)$, and the depth of the expression $(P \;\; C_1 \ldots C_{i-1} \;\; ? \;\; C_{i+1} \ldots C_{\mathrm{n}(P)})$, is $1 + \max\big(\mathrm{depth}(C_1), \ldots, \mathrm{depth}(C_{\mathrm{n}(P)})\big)$. Note that the number of class expressions can be infinite. However, we can limit the number of class expressions under consideration by placing an upper bound on the allowed depth, which we will often do when learning.

Semantics. We now describe the semantics of class expressions, which are defined relative to a finite database D of ground facts over the set of predicates $\mathcal{P}$ and a finite set of constant symbols, or objects. For example, D might correspond to one of the databases described in the previous section. One can simply view the database D as a finite first-order model, or Herbrand interpretation. Given a class expression C and a database D, we use $C[D]$ to denote the set of objects represented by C with respect to D. We also use $P[D]$ to denote the set of tuples of objects corresponding to predicate P in D, i.e. the tuples that make P true.

If $C =$ **a-thing** then $C[D]$ denotes the set of all objects in D. For example, in a database constructed from a Blocksworld state, **a-thing** would correspond to the set of all blocks. If C is a single arity predicate symbol P, then $C[D]$ is the set of all objects in D for which P is true. For example, if D again corresponds to the Blocksworld then clear$[D]$ and ontable$[D]$ denote the sets of blocks that are clear and on the table respectively in D. If $C = C_1 \cap C_2$ then $C[D] = C_1[D] \cap C_2[D]$. For example, (clear$\cap$ontable)$[D]$ denotes the set of blocks that are clear and on the table. If $C = \neg C'$ then $C[D] =$ **a-thing** $- C'[D]$. Finally, for relational composition, if $C = (P \;\; C_1 \ldots C_{i-1} \;\; ? \;\; C_{i+1} \ldots C_{\mathrm{n}(P)})$ then $C[D]$ is the set of all constants c such that there exists $c_j \in C_j[D]$ such that the tuple $(c_1, \ldots, c_{i-1}, c, c_{i+1}, \ldots, c_{\mathrm{n}(R)})$ is in $P[D]$. For example, if D again contains facts about a Blocksworld problem, (on clear ?)$[D]$ is the set of all blocks that are directly under some clear block.

As some additional Blocksworld examples, $C =$ (**c**on **a-thing** ?) describes the blocks that are currently directly under the block that they are supposed to be under in the goal. So if D corresponds to the database in Figure 2 then $C[D] = \{B, \mathrm{table}\}$. Recall that as described in the previous section the predicate **c**on is true of block pairs (x, y) that are correctly on each other, i.e. x is currently on y and x should be on y in the goal. Likewise (**c**on ? **a-thing**) is the set of blocks that are directly above the block they are supposed to be on in the goal and would be interpreted as the set $\{A, C\}$ in database D. Another useful concept is $\neg$(**c**on ? **a-thing**) which denotes the set of blocks that are not currently on their final destination block and would be interpreted as $\{B, \mathrm{table}\}$ with respect to D.

6. Learning Heuristic Functions

Given the relaxed plan feature space, we will now describe how to use that space to represent and learn heuristic functions for use as control knowledge in forward state-space search.

Recall from Section 4.1 that we will utilize the learned heuristics to control (greedy) best-first search in our experiments. Below we first review our heuristic representation followed by a description of our learning algorithm. Recall that heuristic functions are just one of two general forms of control knowledge that we consider in this paper. Our second form, reactive policies, will be covered in the next section.

6.1 Heuristic Function Representation

Recall that a heuristic function $H(s, A, g)$ is simply a function of a state s, action set A, and goal g that estimates the cost of achieving the goal from s using actions in A. In this section, we will consider learning heuristic functions that are represented as weighted linear combinations of functions f_i, i.e. $H(s, A, g) = \Sigma_i w_i \cdot f_i(s, A, g)$. In particular, for each planning domain we would like to learn a distinct set of functions f_i and their corresponding weights that lead to good planning performance in that domain when guided by the resulting linear heuristic function. In this work, each function will correspond to a class expression C_i defined over the relaxed-plan database as described in the previous section, and will be denoted by f_{C_i}. We will take the numeric value of f_{C_i} given a search node (s, A, g) to be the cardinality of C_i with respect to $D(s, A, g)$—that is, $f_{C_i}(s, A, g) = |C_i[D(s, A, g)]|$.

6.2 Heuristic Function Learning

The input to our learning algorithm is a set of planning problems, each paired with an example solution plan, taken from a target planning domain. We do not assume that these solutions are optimal, though there is an implicit assumption that the solutions are reasonably good. Our learning objective is to learn a heuristic function that closely approximates the observed distance-to-goal for each state in the training solutions. To do this, we first create a derived training set $\mathbb{J}$ that contains a training example for each state in the solution set. In particular, for each training problem (s_0, A, g) and corresponding solution trajectory $(s_0, s_1, \ldots, s_n)$ we add to $\mathbb{J}$ a set of n examples $\{((s_i, A, g), n - i) \mid i = 0, \ldots, n - 1\}$, each example being a pair of a planning problem and the observed distance-to-goal in the training trajectory. Given the derived training set $\mathbb{J}$, we then attempt to learn a real valued function $\Delta(s, A, g)$ that closely approximates the difference between the distances recorded in $\mathbb{J}$ and the value of FF's relaxed-plan length (RPL) heuristic, $\text{RPL}(s, A, g)$. We then take the final heuristic function to be $H(s, A, g) = \text{RPL}(s, A, g) + \Delta(s, A, g)$.

We represent $\Delta(s, A, g)$ as a finite linear combination of functions f_{C_i} with the C_i selected from the relaxed plan feature space, i.e., $\Delta(s, A, g) = \Sigma_i w_i \cdot f_{C_i}(s, A, g)$. Note that the overall representation for $H(s, A, g)$ is a linear combination of features, where the feature weight of $\text{RPL}(s, A, g)$ has been clamped to one. Another design choice could have been to allow the weight of the RPL feature to also be learned, however, an initial exploration showed that constraining the value to be one and learning the residual $\Delta(s, A, g)$ gives moderately better performance in some domains.

Learning the above representation involves selecting a set of class expressions from the above infinite space defined in Section 5 and then assigning weights to the corresponding features. One approach to this problem would be to impose a depth bound on class expressions and then learn the weights (e.g., using least squares) for a linear combination that involves all features whose depths of class expression are within the bound. However, the number of such features is exponential in the depth bound, making this approach impractical for all but very small bounds. Such an approach will also have no chance of finding important features beyond the fixed depth bound. In addition, we would prefer

to use the smallest possible number of features, since the time complexity of evaluating the learned heuristic grows in proportion to the number of selected features. Thus, we consider a greedy learning approach where we heuristically search through the space of features, without imposing apriori depth bounds. The procedure described below is a relatively generic approach that we found to work well, however, alternative more sophisticated search approaches are an important direction for future work.

Figure 3 gives our algorithm for learning $\Delta(s, A, g)$ from a derived training set $\mathbb{J}$. The main procedure **Learn-Delta** first creates a modified training set $\mathbb{J}'$ that is identical to $\mathbb{J}$ except that the distance-to-goal of each training example is changed to the difference between the distance-to-goal and FF's relaxed-plan length heuristic. Each iteration of **Learn-Delta** maintains a set of class expressions Φ, which represents the set of features that are currently under consideration. Initially Φ is equal to the set of expressions of depth 0 and 1. Each iteration of the loop has two main steps. First, we use the procedure **Learn-Approximation** (described below) to select a subset of class expressions from Φ and to compute their feature weights. Second, we create a new candidate feature set Φ that includes higher depth class expressions. This is done by using the selected features as seeds and then calling the procedure **Expand-Features**. This results in a larger candidate feature set, including the seed features, which is again used by **Learn-Approximation** to find a possibly improved approximation. We continue alternating between feature space expansion and learning an approximation until the approximation accuracy does not improve. Here we measure the accuracy of the approximation by the R-square value, which is the fraction of the variance in the data that is explained by the linear approximation.

Learn-Approximation uses a simple greedy procedure. Starting with an empty feature set, on each iteration the feature from Φ that can most improve the R-square value of the current feature set is included in the approximation. This continues until the R-square value can no longer be improved. Given a current feature set, the quality of a newly considered feature is measured by calling the function *lm* from the statistics tool R, which outputs the R-square value and weights for a linear approximation that includes the new feature. After observing no improvement, the procedure returns the most recent set of selected features along with their weights, yielding a linear approximation of $\Delta(s, A, g)$.

The procedure **Expand-Features** creates a new set of class expressions that includes the seed set, along with new expressions generated from the seeds. There are many possible ways to generate an expanded set of features from a given seed C. Here we consider three such expansion functions that worked well in practice. The first function **Relational-Extension** takes a seed expression C and returns all expressions of the form $(P \;\; c_0 \ldots c_{j-1} \;\; C \;\; c_{j+1} \ldots c_{i-1} \;\; ? \;\; c_{i+1} \ldots c_{\mathrm{n}(P)})$, where P is a predicate symbol of arity larger than one, the c_i are all **a-thing**, and $i, j \leq \mathrm{n}(P)$. The result is all possible ways of constraining a single argument of a predicate by C and placing no other constraints on the predicate. For example in Blocksworld, a relational extension of the class expression, holding, is (**g**on holding ?). The extended expression describes the block in the goal state that should be under the block currently being held.

The second procedure for generating new expressions given a class expression C is **Specialize**. This procedure simply generates all class expressions that can be created by replacing a single depth zero or one sub-expression c' of C with the intersection of c' and another depth zero or one class expression. Note that all expressions that are produced by **Specialize** will be subsumed by C. That is, for any such expression C', we have that for any database D, $C'[D] \subseteq C[D]$. As an example, given the Blocksworld expression (on ? **a-thing**), one of the class expression generated by **Specialize** would be (on ? (**a-thing** $\cap$ **g**clear)). The input class expression describes all the blocks on some block, and the example output class expression describes the blocks that are currently on blocks that should be clear in the goal state. Finally, we add the complement of the seed classes into

the expanded feature set. For example in Logisticsworld, for the input class expression (**c**in ? **a-thing**), the complement output is ¬(**c**in ? **a-thing**). The input describes packages that are already in the goal location, and the output describes packages that are not in the goal location.

Learn-Delta $(\mathbb{J}, \mathcal{D})$
// $\mathbb{J}$ is pairs of problem states and plan length from them
// $\mathcal{D}$ is domain definition to enumerate class expressions
$\mathbb{J}' \leftarrow \{((s, A, g), d - \text{RPL}(s, A, g)) \mid ((s, A, g), d) \in \mathbb{J}\}$
d is the plan length, the remaining states in the solution trajectories
$\Phi \leftarrow \{C \mid C$ is a class expression of depth 0 or 1$\}$
repeat until no R-square value improvement observed
$(\Phi', W) \leftarrow$ **Learn-Approximation(**$\mathbb{J}'$**,** Φ**)**
// Φ' is newly selected features, W is the set of weights for Φ'
$\Phi \leftarrow$ **Expand-Features(**$\mathcal{D}$**,** Φ'**)**
Return Φ', W

Learn-Approximation $(\mathbb{J}, \Phi)$
$\Phi' \leftarrow \{\}$ *// return features*
repeat until no improvement in R-square value
$C \leftarrow \arg\max_{C \in \Phi}$ R-square$(\mathbb{J}, \Phi' \cup \{C\})$
// R-square is computed after linear approximation with the features
$\Phi' \leftarrow \Phi' \cup \{C\}$
$W \leftarrow lm(\mathbb{J}, \Phi')$
// lm, least square approximation, returns weights
Return Φ', W

Expand-Features $(\mathcal{D}, \Phi')$
$\Phi \leftarrow \Phi'$ *// return features*
for-each $C \in \Phi'$
$\Phi \leftarrow \Phi \cup$ **Relational-Extension**$(\mathcal{D}, C) \cup$ **Specialize**$(\mathcal{D}, C) \cup \{\neg C\}$
Return Φ

Figure 3: Pseudo-code for learning heuristics: The learning algorithm used to approximate the difference between the relaxed plan length heuristic and the observed plan lengths in the training data.

7. Learning Reactive Policies

In this section, we present two representations and associated learning algorithms for reactive policies: taxonomic decision lists and measures of progress. Recall that Section 4.2 describes in detail our novel approach for using the resulting policies as search control knowledge.

7.1 Taxonomic Decision Lists

We first consider representing and learning policies as taxonomic decision lists. Similar representations have been considered previously (Martin and Geffner, 2000; Yoon et al., 2002), though this is the first work that builds such lists from relaxed-plan-based features.

7.1.1 Representation

A taxonomic decision list policy is a list of taxonomic *action-selection rules*. Each rule has the form

$$a(x_1, \ldots, x_k) : L_1, L_2, \ldots L_m$$

where a is a k-argument action type, the L_i are *literals*, and the x_i are action-argument variables. Each literal has the form $x \in C$, where C is a taxonomic syntax class expression and x is an action-argument variable.

Given a search node (s, A, g) and a list of action-argument objects $O = (o_1, \ldots, o_k)$, we say that a literal $x_i \in C$ is satisfied if $o_i \in C[D(s, A, g)]$, that is, object o_i satisfies the constraint imposed by the class expression C. We say that a rule $R = a(x_1, \ldots, x_k) : L_1, L_2, \ldots L_m$ *suggests* action $a(o_1, \ldots o_k)$ in (s, A, g) if each literal in the rule is true given (s, A, g) and O, and the preconditions of the action are satisfied in s. Note that if there are no literals in a rule for action type a, then all legal actions of type a are suggested by the rule. A rule can be viewed as placing mutual constraints on the tuples of objects that an action type can be applied to. Note that a single rule may suggest no action or many actions of one type. Given a decision list of such rules we say that an action is suggested by the list if it is suggested by some rule in the list, and no previous rule suggests any actions. Again, a decision list may suggest no action or multiple actions of one type.

A decision list $\mathbb{L}$ defines a deterministic policy $\pi[\mathbb{L}]$ as follows. If $\mathbb{L}$ suggests no action for node (s, A, g), then $\pi[\mathbb{L}](s, A, g)$ is the lexicographically least action in s, whose preconditions are satisfied; otherwise, $\pi[\mathbb{L}](s, A, g)$ is the least action suggested by $\mathbb{L}$. It is important to note that since $\pi[\mathbb{L}]$ only considers legal actions, as specified by action preconditions, the rules do not need to explicitly encode the preconditions, which allows for simpler rules and learning. In other words, we can think of each rule as implicitly containing the preconditions of its action type.

As an example of a taxonomic decision list policy, consider a simple Blocksworld domain where the goal is to place all of the blocks on the table. The following policy will solve any problem in the domain.

$$\begin{aligned} \text{putdown}(x_1) &: x_1 \in \text{holding} \\ \text{pickup}(x_1) &: x_1 \in (\text{on } ? \ (\text{on } ? \ \textbf{a-thing})) \end{aligned}$$

The first rule will cause the agent to putdown any block that is being held. Otherwise, if no block is being held, then the second rule will pickup a block x_1 that is directly on top of a block that is directly on top of another object (either the table or another block). In particular, this will pickup a block at the top of a tower of height two or more, as desired. [3]

7.1.2 Decision List Learning

Figure 4 depicts the learning algorithm we use for decision list policies. The training set $\mathbb{J}$ passed to the main procedure **Learn-Decision-List** is a multi-set that contains all pairs of

3. If the second rule is changed to pickup$(x_1) : x_1 \in$ (on ? **a-thing**), then the decision rule list may find the loop, since it might try to pick up a block on the table that has just been put down.

```
Learn-Decision-List (𝕁, d, b)
    // 𝕁: set of training instances where each instance is a search node labeled by an action
    // d: the depth limit for class expressions
    // b: beam width, used in search for the rules
𝕃 ← ()
while ( 𝕁 ≠ {} )
    ℝ ← Find-Best-Rule (𝕁, d, b)
    𝕁 ← 𝕁 − {j ∈ 𝕁 | ℝ suggests an action for j}
    𝕃 ← 𝕃 : ℝ; // append rule to end of current list
Return 𝕃
```

```
Find-Best-Rule(𝕁, d, b)
Hvalue-best-rule ← −∞; ℝ ← ()
for-each action type a
   ℝ_a ←Beam-Search(a, 𝕁, d, b)
   if H(𝕁, ℝ_a) > Hvalue-best-rule
      // H(J, R_a) is learning heuristic function in Equation 2
      ℝ ← ℝ_a
      Hvalue-best-rule ← H(𝕁, ℝ_a)
Return ℝ
```

```
Beam-Search(a, 𝕁, d, b)
L_set ← {(x_k ∈ C)|k ≤ n(a), depth(C) ≤ d}
    // the set of all possible literals involving class expressions of depth d or less
beam ← {a(x_1, ..., a_k)} // initial beam contains rule with empty rule body
Hvalue-best ← −∞; Hvalue-best-new ← 0
while (Hvalue-best < Hvalue-best-new)
    Hvalue-best ← Hvalue-best-new
    candidates ← {R, l|l ∈ L_set, R ∈ beam}
        // the set of all possible rules resulting from adding one literal to a rule in the beam
    beam ← set of b best rules in candidates according to heuristic H from Equation 2
    Hvalue-best-new ← H value of best rule in beam
Return best rule in beam
```

Figure 4: Pseudo-code for learning policy

search nodes and corresponding actions observed in the solution trajectories. The objective of the learning algorithm is to find a taxonomic decision list that for each search node in the training set suggests the corresponding action.

The algorithm takes a Rivest-style decision list learning approach (Rivest, 1987) where one rule is learned at a time, from highest to lowest priority, until the resulting rule set "covers" all of the training data. Here we say that a rule *covers* a training example if it suggests an action for the corresponding state. An ideal rule is one that suggests only actions that are in the training data.

The main procedure **Learn-Decision-List** initializes the rule list to () and then calls the procedure **Find-Best-Rule** in order to select a rule that covers many training examples and that correctly covers a high fraction of those examples—i.e. a rule with high coverage and high precision. The resulting rule is then added to the tail of the current decision list,

and at the same time the training examples that it covers are removed from the training set. The procedure then searches for another rule with high coverage and high precision with respect to the reduced training set. The process of selecting good rules and reducing the training set continues until no training examples remain uncovered. Note that by removing the training examples covered by previous rules we force **Find-Best-Rule** to focus on only the training examples for which the current rule set does not suggest an action.

The key procedure in the algorithm is **Find-Best-Rule**, which at each iteration does a search through the exponentially large rule space for a good rule. Recall that each rule has the form

$$a(x_1, \ldots, x_k) : L_1, L_2, \ldots L_m$$

where a is one of the action types and the L_i are of the form $x \in C$. Since this rule space is exponentially large we utilize a greedy beam-search approach. In particular, the main loop of **Find-Best-Rule** loops over each action type a and then uses a beam search to construct a set of literals for that particular a. The best of these rules, as measured by an evaluation heuristic, is then returned. It remains to describe the beam search over literal sets and our heuristic evaluation function.

The input to the procedure **Beam-Search** is the action type a, the current training set, a beam width b, and a depth bound d on the maximum size of class expressions that will be considered. The beam width and the depth bound are user specified parameters that bound the amount of search. We used $d = 2, b = 10$ in all of our experiments. The search is initialized so that the current beam contains only the empty rule, i.e. a rule with head $a(x_1, \ldots, x_k)$ and no literals. On each iteration of the search a candidate rule set is constructed that contains one rule for each way of adding a new literal, with depth bound d, to one of the rules in the beam. If there are n possible literals of depth bound d this will resulting in a set of nb rules. Next the rule evaluation heuristic is used to select the best b of these rules which are kept in the beam for the next iteration, with all other candidates being discarded. The search continues until the search is unable to uncover an improved rule as measured by the heuristic.

Finally, our rule evaluation heuristic $H(\mathbb{J}, R)$ is shown in Equation 2, which evaluates a rule R with respect to a training set $\mathbb{J}$. There are many good choices for heuristics and this is just one that has shown good empirical performance in our experience. Intuitively this function will prefer rules that suggest correct actions for many search nodes in the training set, while at the same time minimizing the number of suggested actions that are not in the training data. We use $R(s, A, g)$ to represent the set of actions suggested by rule R in (s, A, g). Using this, Equation 1, evaluates the "benefit" of rule R on training instance $((s, A, g), a)$ as follows. If the training set action a is not suggested by R then the benefit is zero. Otherwise the benefit decreases with the size of $R(s, A, g)$. That is, the benefit decreases inversely with the number of actions other than a that are suggested. The overall heuristic H is simply the sum of the benefits across all training instances. In this way the heuristic will assign small heuristic values to rules that cover only a small number of examples and rules that cover many examples but suggest many actions outside of the training set.

$$\text{benefit}(((s, A, g), a), R) = \begin{cases} 0 & : \quad a \notin R(s, A, g) \\ \frac{1}{|R(s,A,g)|} & : \quad a \in R(s, A, g) \end{cases} \tag{1}$$

$$H(\mathbb{J}, R) = \sum_{j \in \mathbb{J}} \text{benefit}(j, R) \tag{2}$$

7.2 Measures of Progress

In this section, we describe the notion of measures of progress and how they can be used to define policies and learned from training data.

7.2.1 Representation

Good plans can often be understood as seeking to achieve specific subgoals en route to the goal. In an object-oriented view, these subgoals, along with the goal itself, can be seen as properties of objects, where in each case we wish to increase the number of objects with the given property. Taking this view, we consider control knowledge in the form of compact descriptions of object classes (properties) that a controller is to select actions to enlarge. For example in Blocksworld, in an optimal trajectory, the number of blocks well placed from the table up never decreases, or in Logisticsworld the number of solved packages never decreases in an optimal plan.

Good trajectories also often exhibit locally monotonic properties: properties that increase monotonically for identifiable local periods, but not all the time during the trajectory. For example, in Blocksworld, consider a block "solvable" if its desired destination is clear and well placed from the table up. Then, in good trajectories, while the number of blocks well placed from the table up stays the same, the number of solvable blocks need never decrease locally; but, globally, the number of solvable blocks may decrease as the number of blocks well placed from the table up increases. Sequences of such properties can be used to define policies that select actions in order to improve the highest-priority property possible, while preserving higher-priority properties.

Previously, the idea of monotonic properties of planning domains have been identified by Parmar (2002) as "measures of progress" and we inherit the term and expand the idea to ensembles of measures where the monotonicity is provided via a prioritized list of functions. Let $F = (F_1, \ldots, F_n)$ be an ordered list where each F_i is a function from search nodes to integers. Given an F we define an ordering relation on search nodes (s, A, g) and (s', A, g) as $F(s, A, g) \succ F(s', A, g)$ if $F_i(s, A, g) > F_i(s', A, g)$ while $F_j(s, A, g) = F_j(s', A, g)$ for all $j < i$. F is a strong measure of progress for planning domain $\mathcal{D}$ iff for any reachable problem (s, A, g) of $\mathcal{D}$, either $g \subseteq s$ or there exists an action a such that $F(a(s), A, g) \succ F(s, A, g)$. This definition requires that for any state not satisfying the goal there must be an action that increases some component heuristic F_i while maintaining the preceding, higher-priority components. In this case we say that such an action has priority i. Note that we allow the lower priority heuristics that follow F_i to decrease so long as F_i increases. If an action is not able to increase some F_i, while maintaining all higher-priority components, we say that the action has null priority. In this work we represent our prioritized lists of functions $F = (F_1, \ldots, F_n)$ using a sequence of class expressions $\mathbb{C} = (C_1, \ldots, C_n)$ and just as was the case for our heuristic representation we take the function values to be the cardinalities of the corresponding sets of objects, that is, $F_i(s, A, g) = |C_i[D(s, A, g)]|$.

Given a prioritized list $\mathbb{C}$, we define the corresponding policy $\pi[\mathbb{C}]$ as follows. Given search node (s, A, g), if all legal actions in state s have null priority, then $\pi[\mathbb{C}](s, A, g)$ is just the lexicographically least legal action. Otherwise $\pi[\mathbb{C}](s, A, g)$ is the lexicographically least legal action that achieves highest priority among all other legal actions.

As a simple example, consider again a Blocksworld domain where the objective is to always place all the blocks on the table. A correct policy for this domain is obtained using a prioritized class expression list (C_1, C_2) where $C_1 = \neg$(on ? (on ? **a-thing**)) and $C_2 = \neg$holding. The first class expression causes the policy to prefer actions that are able to increase the set of objects that are *not* above at least two other objects (objects directly on the table are in this set). This expression can always be increased by picking up a block from a tower of height two or greater when the hand is empty. When the hand is

not empty, it is not possible to increase C_1 and thus actions are preferred that increase the second expression while not decreasing C_1. The only way to do this is to putdown the block being held on the table, as desired.

7.2.2 Learning Measures of Progress

Figure 5 describes the learning algorithm for measures of progress. The overall algorithm is similar to our Rivest-style algorithm for learning decision lists. Again each training example is a pair $((s, A, g), a)$ of a search node and the corresponding action selected in that node. Each iteration of the main procedure **Learn-Prioritized-Measures** finds a new class expression, or measure, that is added to the tail of the prioritized list and then removes any newly "covered" examples from the training set. Here we say that a measure C covers a training example $((s, A, g), a)$ if $|C(D[s, A, g])| \neq |C(D[a(s), A, g])|$. It covers the example positively, if $|C(D[s, A, g])| < |C(D[a(s), A, g])|$ and covers it negatively otherwise. Intuitively if a class expression positively covers an example then it increases across the state transition caused by the action of the example. Negative coverage corresponds to decreasing across the transition. We stop growing the prioritized list when we are unable to find a new measure with positive heuristic value.

The core of the algorithm is the procedure **Find-Best-Measure**, which is responsible for finding a new measure of progress that positively covers as many training instances as possible, while avoiding negative coverage. To make the search more tractable we restrict our attention to class expressions that are intersections of class expressions of depth d or less, where d is a user specified parameter. The search over this space of class expressions is conducted using a beam search of user specified width b which is initialized to a beam that contains only the universal class expression **a-thing**. We used $d = 2, b = 10$ in all of our experiments. Given the current beam of class expressions, the next set of candidates contains all expressions that can be formed by intersecting an element of the beam with a class expression of depth d or less. The next beam is then formed by selecting the best b candidates as measured by a heuristic. The search ends when it is unable to improve the heuristic value, upon which the best expression in the beam is returned.

To guide the search we use a common heuristic shown in Equation 3, which is known as weighted accuracy (Furnkranz and Flach, 2003). This heuristic evaluates an expression by taking a weighted difference between the number of positively covered examples and negatively covered examples. The weighting factor ω measures the relative importance of negative coverage versus positive coverage. In all of our experiments, we have used $\omega = 4$ which results in a positive value when the positive coverage is at least four times the negative coverage.

$$H_m(\mathbb{J}, C, \omega) = |\{j | C \text{ covers positively } j \in \mathbb{J}\}| - \omega \times |\{j | C \text{ covers negatively } j \in \mathbb{J}\}| \quad (3)$$

We note that one shortcoming of our current learning algorithm is that it can be fooled by properties that monotonically increase along all or many trajectories in a domain, even those that are not related to distinguishing between good and bad plans. For example, consider a domain with a class expression C, where $|C|$ never decrease and frequently increases along any trajectory. Our learner will likely output this class expression as a solution, although it does not in any way distinguish good from bad trajectories. In many of our experimental domains, such properties do not seem to exist, or at least are not selected by our learner. However, in Philosopher from IPC 4, this problem did appear to arise and hurt the performance of policies based on measures of progress.

There are a number of possible approaches for dealing with this pitfall. For example, one idea would be to generate a set of random (legal) trajectories and reward class expressions that can distinguish between the random and training trajectories.

Learn-Prioritized-Measures $(\mathbb{J}, d, b, \omega)$
// $\mathbb{J}$: states and corresponding actions in the solution plan trajectories
// d: the depth limit of class expressions
// b is the beam width for the search for best measures
// ω is the weight used for the heuristic defined in Equation 3

$\mathbb{C} \leftarrow ()$ // $\mathbb{C}$ *is the list of prioritized measures*
while ($\mathbb{J} \neq \{\}$)
  $C \leftarrow$ **Find-Best-Measure** $(\mathbb{J}, d, b, \omega)$
  if $H_m(\mathbb{J}, C) \leq 0$ // *see Equation 3 for* H_m
    Return $\mathbb{C}$
  $\mathbb{J} \leftarrow \mathbb{J} - \{j \in \mathbb{J} \mid C \text{ covers } j\}$
  $\mathbb{C} \leftarrow \mathbb{C} : C$ // *append new expression to list*

Return $\mathbb{C}$

Find-Best-Measure$(\mathbb{J}, d, b, \omega)$

beam $\leftarrow$ {**a-thing**}
Hvalue-best $\leftarrow -\infty$
Hvalue-best-new $\leftarrow 0$
while (**Hvalue-best-new** $>$ **Hvalue-best**)
  Hvalue-best $\leftarrow$ **Hvalue-best-new**
  candidates $\leftarrow \{C' \cap C | C \in \textbf{beam}, \text{depth}(C') \leq d\}$
  beam $\leftarrow$ best b elements of **candidates** according to $H_m(J, C, \omega)$
  Hvalue-best-new $\leftarrow$ H_m value of best element of **beam**

Return best element of **beam**

Figure 5: Pseudo-code for learning measures of progress

8. Experiments

We evaluated our learning techniques on the traditional benchmark domain Blocksworld and then on a subset of the STRIPS/ADL domains from two recent international planning competitions (IPC3 and IPC4). We included all of the IPC3 and IPC4 domains where FF's RPL heuristic was sufficiently inaccurate on the training data, so as to afford our heuristic learner the opportunity to learn. That is, for domains where the FF heuristic is very accurate as measured in the first 15 training problems, our heuristic learning has nothing to learn since its training signal is the difference between the observed distance in the training set and FF's heuristic. Thus, we did not include such domains.

For each domain, we used 15 problems as training data, with the solutions being generated by FF. We then learned all three types of control knowledge (heuristics, taxonomic decision lists, and measures of progress) in each domain and used that knowledge to solve the remaining problems, which were typically more challenging than the training problems. We used each form of control knowledge as described in Section 4. For the case of the policy representations (taxonomic decision lists and measures of progress), we utilized FF's RPL heuristic as the heuristic function and used a fixed policy-execution horizon of 50.

To study the utility of our proposed relaxed-plan feature space, we also conducted separate experiments that removed the relaxed plan features from consideration. As the

experiments will show, the relaxed-plan features were critical to achieving good performance in a number of domains. The time cutoff for each planning problem was set to 30 CPU minutes and a problem was considered unsolved after reaching the cutoff. For all of the experiments, we used a Linux box with 2 Gig RAM and 2.8 Ghz Intel Xeon CPU.

8.1 Table Mnemonics

Before we present our results, we first explain the mnemonics used in our data tables. We provide one table for each of our domains, each having the same structure, for example, Figure 6 gives our Blocksworld results. Each row corresponds to a distinct planning technique, some using learned control knowledge and some not. The mnemonics used for these planners are described below. These mnemonics are also used in the main text.

- **FF**: the planner FF (Hoffmann and Nebel, 2001). FF adds goal-ordering, enforced-hill climbing, and helpful action pruning on top of relaxed plan heuristic search. If all these fail, FF falls back on best first search guided by relaxed plan length.
- **RPL**: greedy best-first search using FF's RPL heuristic.
- **Best**: best performer of the corresponding domain during the competition (only available for IPC4).
- **DL**: greedy best-first search guided by RPL and *learned decision list policy* using the full relaxed-plan feature space, refer to Section 7 for the representation and learning algorithm and Section 4 for how to use the knowledge.
- **H**: greedy best-first search guided by a *learned heuristic function* using the full relaxed-plan feature space, refer to Section 6 for representation and learning.
- **MoP**: greedy best-first search guided by RPL and *learned measures-of-progress policy* using the full relaxed-plan feature space, refer to Section 7 for the representation and learning and to Section 4 for use of the knowledge to guide the search.
- **DL-noRP, H-noRP, MoP-noRP**: identical to **DL**, **H**, and **MoP** except that the control knowledge is learned from a feature space that does not include relaxed plan information. That is, the database construction described in Section 5.3 does not include any facts related to the relaxed plan into the search node databases. These experiments are conducted to check the usefulness of the relaxed plan information.

The columns of the results tables show various performance measures for the planners. In the following, we list mnemonics for the performance measures and their descriptions.

- **Solved** (n): gives the number of problems solved within 30 minutes out of n test problems.
- **Time**: the average CPU time in seconds consumed across all problems that were solved within the 30 minute cutoff.
- **Length**: average solution length of the problems solved within the 30 minute cutoff.
- **LTime**: the time used for learning (only applies to the learning systems). Unless followed by H, the number is in seconds.
- **Greedy**: number of problems solved using greedy execution of the decision list or measures-of-progress policies (only applies to systems that learn decision list rules and measures of progress). This allows us to observe the quality of the policy without the integration of heuristic search.
- **Evaluated**: the average number of states evaluated on the problems that were solved with the 30 minute cutoff.

8.2 Blocksworld Results

For Blocksworld, we have used the competition problems from track 1 of IPC2. There were 35 problems giving us 15 problems for training and 20 problems for testing. For this domain only, we included one additional predicate symbol "above" that is not part of the original domain definition. The above predicate is computed as the transitive closure of the "on" predicate and is true of tuple (x, y) if x is above y in a stack of towers. This is an important concept to be able to represent in the Blocksworld and is not expressible via the fragment of taxonomic syntax used in this work. Note that in prior work (Yoon et al., 2002; Fern et al., 2006), we included the Kleene-star operator into the taxonomic syntax, which allowed for concepts such as above to be expressed in terms of the primitive predicates. However, we have decided to not include Kleene star in this work as it did not appear necessary in most of the other domains and increases the size of the search space over taxonomic expressions and hence learning time.

Blocksworld (IPC2)						
Techniques	Solved (20)	Time	Length	LTime	Greedy	Evaluated
FF	16	0.64	38.12	-	-	15310
RPL	20	11.74	116	-	-	12932
DL	20	0.05	44	100	13	215
MoP	20	0.13	51.7	10	20	757
H	20	12.94	82.7	600	-	31940
DL-noRP	20	0.12	60.8	12	0	915
MoP-noRP	20	0.15	49.4	1	20	984
H-noRP	12	113.3	352	88	-	185808

Figure 6: Blocksworld (IPC2, Track 1) Results. For information on mnemonics, please refer sub-Section 8.1

Figure 6 shows Blocksworld results for various learning and non-learning systems. For this domain, **DL**, **H**, and **MoP** were all able to solve all of the problems. Note that greedy application of the learned decision list policy manages to solve only 13 of the 20 problems, indicating that the learned policy has a significant error rate. Despite this error rate, however, incorporating the policy into search as implemented in **DL** allows for all 20 problems to be solved. Furthermore, the incorporation of the policy into search significantly speeds up search, achieving an average search time of 0.05 seconds compared to a time of 11.74 seconds achieved by **RPL**, which uses the same RPL heuristic as **DL** but ignores the policy. The number of evaluated states partially shows why running policies in the best first search helps. The average number of evaluated states is significantly lower for **DL** compared to other techniques. Later in our discussion of the Depots domain we will give empirical evidence that one reason for this reduction in the amount of search is that the policies are able to quickly move through plateaus to find states with low heuristic values.

We see that for this domain the measures of progress are learned quite accurately, allowing for greedy search to solve all of the problems. Again, as for the decision-list policies we see that the incorporation of the measures of progress into search significantly speeds up planning time compared to **RPL**. The fact that measures of progress are learned more accurately is possibly due to the fact that the training data for decision-list policy learning is quite noisy. That is, all actions not in the training plans are treated as negative

examples, while in fact many of those actions are just as good as the selected action, since there are often many good action choices in a state. The training data for learning measures of progress does not include such noisy negative examples. Note that prior work (Yoon et al., 2002) has learned highly accurate Blocksworld policies, however, there the training data contained the set of *all* optimal actions for each state, with all other actions labeled as negative. Thus, the training data was not nearly as noisy in that work.

Considering now the performance of the heuristic learner **H**, we see that overall its solution times were larger than for **RPL** and also considers more states than **RPL**. However, the heuristic learner **H** did find significantly better solutions than **RPL**, which used only the RPL heuristic, reducing the average length from 116 to 83. Thus, by attempting to learn a more accurate heuristic, **H** is able to find higher quality solutions at the expense of more search.

Overall we see that the learners **DL** and **MoP** are more effective in this domain. This is likely because it is possible to learn very good decision list rules and measure of progress in the Blocksworld, which guide the heuristic search to good solutions very quickly. Note that FF solves fewer problems than other systems, but the average solution length of FF is the best, noting that it is difficult to compare averages between planners that solve different sets of problems. Apparently enforced hill-climbing and the goal-ordering mechanisms of FF help facilitate shorter solutions when they are able to find solutions.

Our feature comparison results indicate that when the relaxed plan features are removed, decision-list and measures-of-progress learning still solve all of the problems, but the heuristic learner **H-noRP** only solves 12 problems. This indicates that the relaxed plan information is important to the heuristic learner in this domain. We note that previous work on learning Blocksworld decision list rules and measures did not use relaxed-plan features and also had reasonable success. Thus, for the Blocksworld it was not surprising that relaxed plan features were not critical for the policy learners.

The learning time for policies and measures are negligible for Blocksworld. As will be revealed later, the learning time for decision-list policies is quite significant in many other domains. For the Blocksworld domain, the number of predicates, the number of actions, and the arities of predicates and actions are all small, which greatly reduces the complexity of feature enumeration during learning, as described in Section 7. For many domains in the following experiments those numbers increase sharply, and accordingly so does the learning time. It is important to remember, however, that learning time is a price we pay once, and can then be amortized over all future problem solving.

8.3 IPC3 Results

IPC3 included 6 STRIPS domains: Zenotravel, Satellite, Rover, Depots, Driverlog, and FreeCell. FF's heuristic is very accurate for the first three domains, trivially solving the 15-training-problems and leaving little room for learning. Thus, here we report results on Depots, Driverlog, and FreeCell. Each domain includes 20 problems, and FF was able to generate training data for all 15 training problems. Figures 7, 8, 9 summarizes the results.

For Depots, FF performed the best, solving all of the problems. At the same time the average plan length was short. Clearly, the goal-ordering and the enforced hill climbing were key factors to this success since **RPL**, which carries out just the best-first search component of FF, only solved one problem. The learning systems **DL**, **MoP** and **H** were all able to solve all of the problems, showing that the learned knowledge was able to overcome the poor performance of **RPL**. However, the learning systems consumed more time than FF and/or resulted in longer plans. In the Driverlog domain, **MoP** was the only system able to solve all of the problems, with **H** and **DL** solving 4 and 3 problems respectively compared to the single solved problem of the non-learning systems. In FreeCell, both non-learning

Depots (IPC3)						
Techniques	Solved (5)	Time	Length	LTime	Greedy	Evaluated
FF	5	4.32	54.2	-	-	1919
RPL	1	0.28	29	-	-	1106
DL	5	3.73	63.2	8H	0	2500
MoP	5	48.5	81.2	950	0	16699
H	5	174	68.4	325	-	71795
DL-noRP	2	0.83	30	8H	0	437
MoP-noRP	4	94.5	124.7	900	0	101965
H-noRP	3	5	61.7	258	-	4036

Figure 7: Depots results

Driverlog (IPC3)						
Techniques	Solved (5)	Time	Length	LTime	Greedy	Evaluated
FF	1	37.62	149	-	-	171105
RPL	1	1623	167	-	-	165454
DL	4	75.9	177	6H	0	15691
MoP	5	546	213	600	0	126638
H	3	199	402	274	-	98801
DL-noRP	3	86	142	8H	0	33791
MoP-noRP	3	583	177	900	0	253155
H-noRP	1	37	149	252	-	22164

Figure 8: Driverlog results

systems were able to outperform the learning systems in terms of problems solved and plan lengths. The learning system **DL** is close behind these systems, solving one less problem, with about the same average length. This domain is a clear example of how learned control knowledge can sometimes hurt performance. In practice, one might utilize a validation process to determine whether to use learned control knowledge or not, and also to select among various forms of control knowledge.

Aggregating the number of problems solved across the three domains in IPC3, the learning and planning systems **DL**, **MoP** and **H** all solved more problems than FF and **RPL**. Overall the learned control knowledge generally speeds up planning. The solution lengths are sometimes longer, though it is quite likely that post processing techniques could be used to help reduce length by removing wasteful parts of the plan. This for example has been done effectively in the planning by rewriting framework (Ambite and Knoblock, 2001). In such frameworks, simply finding a sub-optimal solution quickly is a key requirement that our learning approaches help facilitate.

Interestingly, greedy action selection according to both the learned decision list policies and measures of progress is unable to solve any testing problem and very few training problems. This indicates that the learned policies and measures have significant flaws. Yet, when incorporated into our proposed search approach, they are still able to improve

Freecell (IPC3)						
Techniques	Solved (5)	Time	Length	LTime	Greedy	Evaluated
FF	5	574	108	-	-	26344
RPL	5	442	109	-	-	25678
DL	4	217	108	6H	0	12139
MoP	3	371	192	1000	0	44283
H	5	89	145	4214	-	5096
DL-noRP	4	200	111	8H	0	11132
MoP-noRP	3	368	129	900	0	18299
H-noRP	2	29.87	85.5	3892	-	1521

Figure 9: FreeCell results

planning performance. This shows that even flawed control knowledge can be effectively used in our framework, assuming it provides some amount of useful guidance.

The **DL** system took a significant amount of time to learn in all of the domains, on the order of hours. These domains have more predicates, actions and larger arities for each action than Blocksworld, leading to a bigger policy search space. Still, the benefit of the policy cannot be ignored, since once it is learned, the execution of a policy is much faster than measures of progress or heuristic functions. This is verified by the solution time. **DL** consumes the least planning time among all the learning systems. Finally, we see that for all the domains, the use of the relaxed-plan-based features improved performance as exhibited by the better performance of the systems **DL**, **MoP**, and **H** compared to **DL-noRP**, **MoP-noRP**, and **H-noRP** respectively.

Our approach to incorporating policies into search appears to help speed-up the search process by adding nodes that are far from the current node being expanded, helping to overcome local minima of the heuristic function. To help illustrate this, Figures 10 and 11 show the heuristic value trace during the search in problem 20 of Depot, where **RPL** performed poorly and **DL** solved the problem quickly. The figures plot the heuristic values of each newly expanded node. Figure 10 shows the heuristic trace for best-first search as used by **RPL**. The search stays at heuristic value 40 for more than 10000 node expansion, stuck in some local minima. Rather, Figure 11 shows that **DL**, may have been stuck in a local minima from node expansion 5 to 8, but quickly finds its way out and finds the goal after only 16 node expansions.

In contrast, recall that incorporating our learned policies into search in the Freecell domain hurt performance. To help understand this, we again plotted the heuristic trace during the search in Figures 12 and 13. As is the case with Depots problem 20, the learned policy leads to a heuristic value jump (though small), but the jump did not help and we conjecture that the jump has led the search to some local minimum for both the heuristic and policy, which would not necessarily be visited with the heuristic alone, causing poor performance here for the policy-based approach.

8.4 IPC4 Results

IPC4 includes seven STRIPS/ADL domains: Airport, Satellite, Pipesworld, Pipesworld-with-Tankage, Philosophers (Promela), Optical Telegraph (Promela), and PSR (middle-compiled). FF's heuristic is very accurate for the first two domains, where for all of the solved problems the solution length and FF's heuristic are almost identical, leaving little

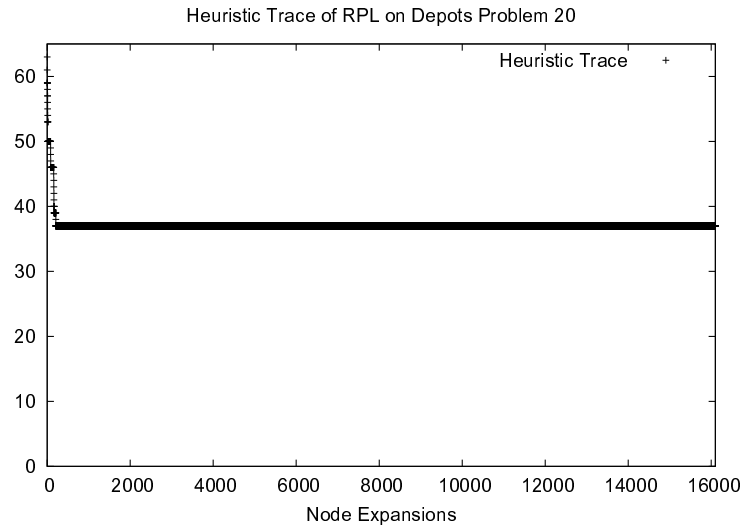

Figure 10: Heuristic trace of RPL on Depots problem 20: The RPL search found some local minimum heuristic around 38 at about 200th node expansion and remain in that region for over 15000 search nodes.

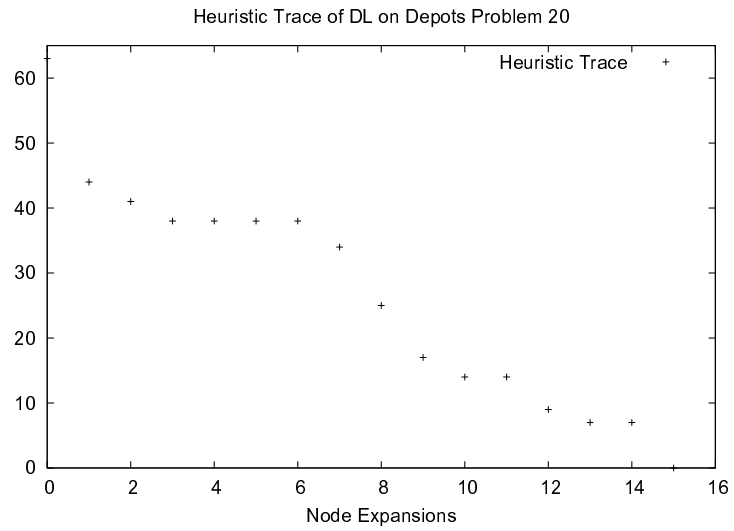

Figure 11: Heuristic trace of DL on Depots problem 20: The DL search hit a local minimum from node expansion 5 to 8, but quickly found a way out of it and reached the goal in just 16 node expansions.

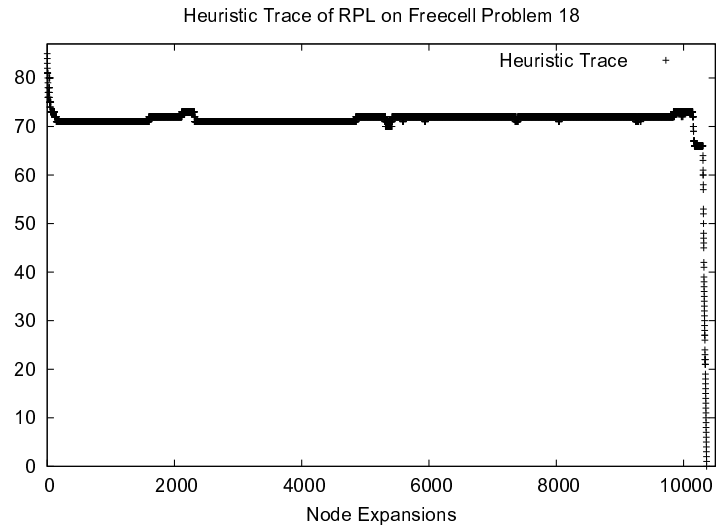

Figure 12: Heuristic trace of RPL on FreeCell problem 18: The RPL search hit a local minimum of around 70 for a long sequence of node expansions, but in the end, the search found a way out after 10000 node expansions.

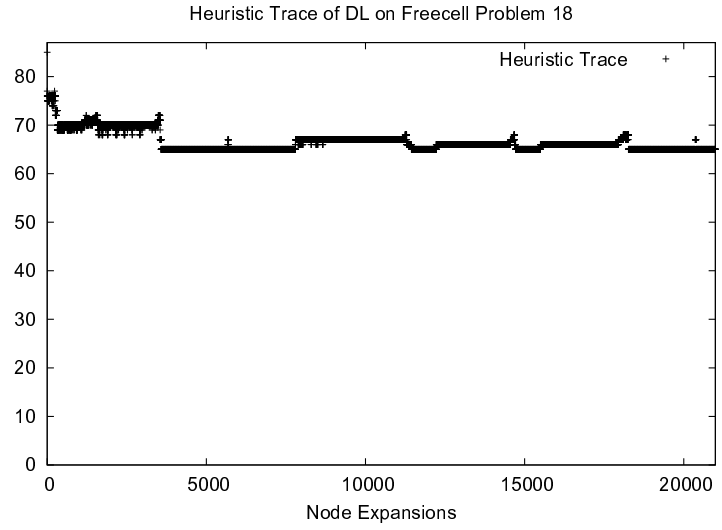

Figure 13: Heuristic trace of DL on FreeCell problem 18: Compared to Figure 12, DL search found lower heuristic states faster but did not find a way out of it. The states may not necessarily been found by RPL search.

Pipesworld (IPC4)						
Techniques	Solved (35)	Time	Length	LTime	Greedy	Evaluated
FF	21	71.2	48.2	-	-	63077
RPL	15	71.3	48.2	-	-	26830
Best	35	4.94	74.6	-	-	-
DL	28	129	76.7	18H	0	37778
MoP	25	155	79.2	1000	0	48460
H	24	17.6	98.4	709	-	22448
DL-noRP	26	10.6	67.7	12H	0	8453
MoP-noRP	23	137	87.6	870	0	67733
H-noRP	15	276	319	685	-	134366

Figure 14: Pipesworld Results

Pipesworld-with-Tankage (IPC4)						
Techniques	Solved (35)	Time	Length	LTime	Greedy	Evaluated
FF	4	532	62	-	-	120217
RPL	4	333	62	-	-	100952
Best	28	221	165	-	-	-
DL	16	124	86	36H	0	49455
MoP	14	422	138	1200	0	101833
H	12	281	39	2091	-	137996
DL-noRP	10	553	52	33H	0	159113
MoP-noRP	13	287	116	1038	0	77926
H-noRP	7	441	719	1970	-	153108

Figure 15: Pipesworld Tankage Results

room for learning. Thus, we only give results for the latter five domains. Each domain includes either 48 or 50 problems, giving a total of 33 or 35 testing problems, using the 15 lower numbered problems as training examples. Figures 14, 15, 16, 17, 18 present the results.

For the IPC4 results, we show the performance of the competition's best performer in each domain, labeled as **Best**. Note that the best planner varies from domain to domain. The CPU times for the best performer were taken from the official competition results, and thus are not exactly comparable to the CPU times of the other systems which were run on our local machine. In order to provide a rough comparison between the times reported for **Best** from the IPC4 results, and the times on our own machine, we ran Yahsp (Vidal, 2004) on our machine for a number of benchmark problems. In most cases, the times were quite similar. For example, on problems 40 and 45 from Pipesworld with Tankage the IPC4 results reported CPU times of 112.08 and 60.45, while we recorded times of 127.35 and 68.76 on our machine

Overall, as for the IPC4 domains, our learning systems **DL** and **H** solved more problems than FF or **RPL**, showing that the learning and planning approaches are useful. **DL** and **H** solved 127 problems among 171 testing problems from all of the domains while FF solved

PSR (IPC4)						
Techniques	Solved (35)	Time	Length	LTime	Greedy	Evaluated
FF	17	692	108	-	-	13706
RPL	22	710	116	-	-	12829
Best	18	134	111	-	-	-
DL	17	736	102	1800	0	15448
MoP	-	-	-	-	-	-
H	25	568	109	2848	-	3189
DL-noRP	11	795	94	1530	0	26022
MoP-noRP	-	-	-	-	-	-
H-noRP	23	565	296	2685	-	4807

Figure 16: PSR Results

Philosophers (IPC4)						
Techniques	Solved (33)	Time	Length	LTime	Greedy	Evaluated
FF	0	-	-	-	-	-
RPL	0	-	-	-	-	-
Best	14	0.2	258	-	-	-
DL	33	2.59	363	3H	33	727
MoP	0	-	-	-	0	-
H	33	58.2	363	340	-	30325
DL-noRP	0	-	-	-	-	-
MoP-noRP	0	-	-	-	-	-
H-noRP	0	-	-	-	-	-

Figure 17: Philosophers Results

42 and **RPL** solved 41. Quite surprisingly, **DL** and **H** were even able to outperform the collective results of the best performers, which solved a total of 105 problems. The **MoP** system did not perform as well as the other learners. In PSR, **MoP** was unable to learn any monotonic properties, and so was not even run. In Philosophers and Optimal Telegraph, **MoP** did find monotonic properties, but those properties hurt performance. Compared to **DL** and **H**, **DL-noRP** and **H-noRP** significantly underperformed. **DL-noRP** solved 37 and **H-noRP** solved 45 showing the usefulness of the relaxed plan feature space.

Finally, note that for most of these domains, greedy execution of the learned policies does not solve any problems. Again, however, our approach to incorporating the policies into search is still able to exploit them for significant benefits.

9. Discussion and Future Work

This study provided two primary contributions. First, we introduced a novel feature space for representing control knowledge based on extracting features from relaxed plans. Second, we showed how to learn and use control knowledge over this feature space for forward state-

Optical Telegraph (IPC4)						
Techniques	Solved (33)	Time	Length	LTime	Greedy	Evaluated
FF	0	-	-	-	-	-
RPL	0	-	-	-	-	-
Best	10	721	387	-	-	-
DL	33	501	594	1H	33	930
MoP	0	-	-	-	-	-
H	33		594	826	-	9777
DL-noRP	0	-	-	-	-	-
MoP-noRP	0	-	-	-	-	-
H-noRP	0	-	-	-	-	-

Figure 18: Optical Telegraph Results

space heuristic search planning, a planning framework for which little work has been done in the direction of learning. We have shown that the combined approach is competitive with state-of-the-art planners across a wide range of benchmark problems. To the best of our knowledge, no prior learning-to-plan system has competed this well across such a wide set of benchmarks.

One natural extension to the relaxed-plan feature space introduced in this paper is to consider properties based on the temporal structure of relaxed plans. This could be accomplished by extending our current feature language to include temporal modalities. Regarding the learning of heuristics, our learning approach reduces the problem to one of standard function approximation. There are a number of ways in which we might further improve the quality of the learned heuristic. One approach would be to use ensemble-learning techniques such as bagging (Breiman, 1996), where we learn and combine multiple heuristic functions. Another more interesting extension would be to develop a learning technique that explicitly considers the search behavior of the heuristic, focusing on parts of the state space that need improvement the most. An initial step in this direction has been considered by Xu et al. (2007).

A key problem in applying learned control knowledge in planning is to robustly deal with imperfect knowledge resulting from the statistical nature of the learning process. Here we have shown one approach to help overcome imperfect policies by incorporating them into a search process. However, there are many other planning settings and forms of control knowledge for which we are interested in developing robust mechanisms for applying control knowledge. For example, stochastic planning domains and planning with richer cost functions are of primary interest. Learning control knowledge for regression-based planners is also of interest—it is not clear how the forms of knowledge we learn here could be used in a regression based setting. As another example, we are interested in robust methods of incorporating control knowledge into SAT-based planners. Huang et al. (2000) have considered an approach to learning and incorporating constraints into a SAT-based planner. However, the approach has not been widely evaluated and it appears relatively easy for imperfect knowledge to make the planner incomplete by ruling out possible solutions.

Another research direction is to consider extending the relaxed-plan feature space to stochastic planning domains. Here one might determinize the stochastic domain in one or more ways and compute the corresponding relaxed plans. The resulting features could then be used to learn policies or value functions, using an approach such as approximate

policy iteration (Fern et al., 2006). It would also be interesting to consider extending our approach for incorporating imperfect policies into search in the context of real-time dynamic programming (Barto et al., 1995).

In summary, we have demonstrated that it is possible to utilize machine learning to improve the performance of forward state-space search planners across a range of planning domains. However, the results are still far from the performance of human-written control knowledge in most domains, for example, TL-Plan (Bacchus and Kabanza, 2000) and SHOP (Nau et al., 1999). Also the results have still not shown large performance gains over state-of-the-art non-learning systems. Demonstrating this level of performance should be a key goal of future work in learning-based planning systems.

10. Acknowledgement

We thank Subbarao Kambhampati for valuable discussion and support. This work was supported in part by NSF grant IIS-0546867, DARPA contract FA8750-05-2-0249 and the DARPA Integrated Learning Program (through a sub-contract from Lockheed Martin).

References

Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Using genetic programming to learn and improve control knowledge. *Artificial Intelligence Journal*, 141(1-2):29–56, 2002.

Jose Luis Ambite and Craig Knoblock. Planning by rewriting. *Journal of Artificial Intelligence Research*, 15:207–261, 2001.

Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence Journal*, 16:123–191, 2000.

Andy Barto, Steven Bradtke, and Satinder Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72:81–138, 1995.

J. Benton, Minh Do, and Subbarao Kambhampati. Oversubscription planning with metric goals. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*, 2006.

Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1636–1642, 1995.

Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33, 2001.

Adi Botea, Markus Enzenberger, Martin Muller, and Jonathan Schaeffer. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

Daniel Bryce and Subbarao Kambhampati. Planning graph heuristics for belief space search. *Journal of Artificial Intelligence Research*, (26):35–99, 2006.

Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

Andrew I. Coles and Amanda J. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, February 2007. ISSN 11076-9757.

Minh Do and Subbarao Kambhampati. Sapa: A scalable multi-objective heuristic metric temporal planner. *Journal of Artificial Intelligence Research*, (20):155–194, 2003.

Saso Dzeroski, Luc De Raedt, and Kurt Driessens. Relational reinforcement learning. *Machine Learning Journal*, 43:7–52, 2001.

Tara A. Estlin and Rymond J. Mooney. Multi-strategy learning of search control for partial-order planning. In *Proceedings of 13th National Conference on Artificial Intelligence*, 1996.

Alan Fern, Sungwook Yoon, and Robert Givan. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *Journal of Artificial Intelligence Research*, 25: 85–118, 2006.

Richard Fikes, Peter Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence Journal*, 3(1–3):251–288, 1972.

Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

Johannes Furnkranz and Peter A. Flach. An analysis of rule evaluation metrics. In *Proceedings of Twentieth International Conference on Machine Learning*, 2003.

Alfonso Gerevini and Lenhart K. Schubert. Discovering state constraints in DISCOPLAN: Some new results. In *Proceedings of National Conference on Artificial Intelligence*, pages 761–767. AAAI Press / The MIT Press, 2000.

William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1995, August 20-25 1995.

Malte Helmert, Patrik Haslum, and Jrg Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling*, 9 2007.

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

Yi-Cheng Huang, Bart Selman, and Henry Kautz. Learning declarative control rules for constraint-based planning. In *Proceedings of Seventeenth International Conference on Machine Learning*, pages 415–422, 2000.

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

Mario Martin and Hector Geffner. Learning generalized policies in planning domains using concept languages. In *Proceedings of Seventh International Conference on Principles of Knowledge Representation and Reasoning*, 2000.

David McAllester and Robert Givan. Taxonomic syntax for first-order inference. *Journal of the ACM*, 40:246–283, 1993.

Steve Minton, Jaime Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence Journal*, 40:63–118, 1989.

Steven Minton. Quantitative results concerning the utility of explanation-based learning. In *Proceedings of National Conference on Artificial Intelligence*, 1988.

Steven Minton, editor. *Machine Learning Methods for Planning*. Morgan Kaufmann Publishers, 1993.

Dana Nau, Yue Cao, Amnon Lotem, and Héctor Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 968–973, 1999.

XuanLong Nguyen, Subbarao Kambhampati, and Romeo Sanchez Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. *Artificial Intelligence*, 135(1-2):73–123, 2002.

Aarati Parmar. A logical measure of progress for planning. In *Proceedings of Eighteenth National Conference on Artificial Intelligence*, pages 498–505. AAAI Press, July 2002.

Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

Vincent Vidal. A lookahead strategy for heuristic search planning. In *International Conference on Automated Planning and Scheduling*, 2004.

Yuehua Xu, Alan Fern, and Sungwook Yoon. Discriminative learning of beam-search heuristics for planning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, 2007.

Sungwook Yoon, Alan Fern, and Robert Givan. Inductive policy selection for first-order MDPs. In *Proceedings of Eighteenth Conference in Uncertainty in Artificial Intelligence*, 2002.

Sungwook Yoon, Alan Fern, and Robert Givan. Learning measures of progress for planning domains. In *Proceedings of Twentieth National Conference on Artificial Intelligence*, 2005.

Terry Zimmerman and Subbarao Kambhampati. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine*, 24(2)(2):73–96, 2003.