Learning Domain-Specific Control Knowledge from Random Walks

Alan Fern and **SungWook Yoon** and **Robert Givan**
Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907 USA
{afern, sy, givan}@purdue.edu

Abstract

We describe and evaluate a system for learning domain-specific control knowledge. In particular, given a planning domain, the goal is to output a control policy that performs well on "long random walk" problem distributions. The system is based on viewing planning domains as very large Markov decision processes and then applying a recent variant of approximate policy iteration that is bootstrapped with a new technique based on random walks. We evaluate the system on the AIPS-2000 planning domains (among others) and show that often the learned policies perform well on problems drawn from the long–random-walk distribution. In addition, we show that these policies often perform well on the original problem distributions from the domains involved. Our evaluation also uncovers limitations of our current system that point to future challenges.

Introduction

The most effective current planners utilize domain-specific control knowledge. For example, TL-Plan (Bacchus & Kabanza 2000) and SHOP (Nau *et al.* 1999) use such knowledge to dramatically prune the search space, often resulting in polynomial-time planning performance. Attaining this efficiency, however, requires effective control knowledge for each planning domain, typically provided by a human. Given a means for automatically producing good control knowledge, we can achieve domain-specific planning performance with a domain-independent system. In this work, we take a step in that direction, limiting our attention to control knowledge in the form of reactive policies that quickly select actions for any current state and goal condition, and avoiding any dependence on human-provided "small problems" for learning.

We present and evaluate a new system that takes a planning domain as input and learns a control policy that is tuned to perform well on the "long random walk" (LRW) problem distribution. This distribution randomly generates a problem (i.e. an initial-state and goal) by selecting an initial state from the given planning domain and then executing a "long" sequence of random actions, taking the goal condition to be a subset of properties from the resulting state.

We are not aware of prior work that explicitly considers learning control knowledge for the LRW distribution; however, this learning problem is interesting for a number of reasons. First, the LRW-performance goal suggests an automatic and domain-independent strategy for "bootstrapping" the learning process by using random-walk problems of gradually increasing walk length. As we discuss in the related-work section below, nearly all existing systems for learning control knowledge require human-assisted bootstrapping. Second, policies that perform well on the LRW distribution clearly capture much domain knowledge that can be leveraged in various ways. For example, in this work we show that such policies often perform well on the problem distributions from recent AIPS planning competitions. Such policies can also serve as macro actions for achieving subgoals, perhaps suggested by landmark analysis (Porteous, Sebastia, & Hoffmann 2001; Zhu & Givan 2003), though we do not explore that direction here.

Techniques for finding and improving control policies have been the predominant focus of Markov-decision-process (MDP) research. In this work, we show how to leverage our recent work in that area to solve some familiar AI planning domains. Though here we focus on solving deterministic STRIPS/ADL domains, our system is applicable to arbitrary MDPs, giving it a number of advantages over many existing control-knowledge learners. First, our system, without modification, addresses both stochastic and deterministic domains as well as domains with general reward structure (where reward is not concentrated in a "goal region"). Second, our system is not tied to a particular domain representation (e.g. STRIPS, ADL, PSTRIPS) or planner. We only require the availability of an action simulator for the planning domain. Rather than exploit the domain representation deductively, we exploit the state- and action-space structure inductively using a learning algorithm. Finally, the MDP formalism provides a natural way to leverage planning heuristics (if available) and, thus, can take advantage of recent progress in domain-independent heuristics for STRIPS/ADL domains (of course, only when the problem is specified in STRIPS/ADL, or such a specification can be learned).

Our system is based on recent work (Fern, Yoon, & Givan 2003) that introduces a variant of approximate policy iteration (API) for solving very large "relational" MDPs by *iteratively improving policies* starting from an initial (often poor) policy. In that work, API was applied to MDPs representing the blocks world and a simplified logistics world (with no planes), and produced state-of-the-art control policies. However, there remain significant challenges for incorporating API into a fully automatic system for learning policies in a wider range of domains, such as the full corpus of AIPS benchmarks. In particular, for non-trivial planning domains API requires some form of bootstrapping. Previously, for the blocks world and simplified logistics world, we were able to bootstrap API, without human assistance, by using the domain-independent FF heuristic (Hoffmann & Nebel 2001). This approach, however, is limited both by the heuristic's ability to provide useful "bootstrapping" guidance, which can vary widely across domains, as well as by the learner's ability to capture the improved policy indicated by its training set at each iteration.

Here, we continue to use the previous underlying learning method, and do not yet address its weaknesses (which do show up in some of our experiments). Instead, we describe a novel and automatic bootstrapping technique for guiding API toward policies for the LRW problem distribution, to supplement the bootstrapping guidance provided by the heuristic, if any. Intuitively, the idea is to initially use API to find policies for short-random-walk distributions, and to incrementally increase the walk length during iterative policy improvement until API uncovers a policy that works well for long walks. Our primary goal in this paper is to demonstrate that, with this idea, our resulting system is able to find good control policies in a wide range of planning benchmarks.

Our empirical results on familiar planning domains, including the AIPS-2000 benchmarks, show that our system can often learn good policies for the LRW distribution. In addition, these same policies often perform as well and sometimes better than FF on the planning-competition problem distributions, for which FF is known to be well suited. Our results also suggest that, in the domains where we do not find good policies, the primary reason is that our current policy language is unable to express a good policy. This suggests an immediate direction, orthogonal to bootstrapping, for further improving our system by adding expressiveness to the policy representation and improving the corresponding learner.

To our knowledge this is the first thorough empirical comparison of a control-knowledge learner to a state-of-the-art planner on this corpus[1]. These results demonstrate that we are able to automatically find reactive policies that compete with established domain-independent techniques and point to a promising and much different direction for advancing domain-independent planning.

Problem Setup

Planning Domains. Our system represents a *planning domain* using an action-simulator model $D = \langle S, A, T, C, I \rangle$, where S and A are finite sets of states and actions, respectively. The third component T is a (possibly stochastic) "action simulation" algorithm, that, given state s and action a, returns a next state t. The fourth component C is an action-cost function that maps $S \times A$ to real-numbers, and I is a randomized "initial state" algorithm, that returns a state in S. Throughout this section, we assume a fixed planning domain D of the form above.

The learning component of our system assumes that the sets S and A are represented, in the usual way, by specifying a set of objects O, a set of state predicates P, and a set of action types Y. A *state fact* is a state predicate applied to the correct number of objects and a *state* in S is a set of state facts. The *actions* in A are action types applied to objects.

While our system is not tied to any particular action representation, or to deterministic actions in general, from here forward in this paper, we focus on applying our system to deterministic STRIPS/ADL planning domains with goal-based reward. In this setting, T corresponds to a deterministic interpreter for STRIPS/ADL action definitions, C will always return one, and I corresponds to a legal-state generator, e.g. from a planning competition. Goals are generated by random walks from initial states drawn from I.

Policy Selection. A *planning problem* for D is a pair $\langle s, g \rangle$, where s is a state and g is a set of state facts called the *goal condition.* A *policy* is a mapping from planning problems to actions. We say a policy is *efficient* if it can be evaluated in polynomial time in the size of its input problem. Given an initial problem $\langle s_0, g \rangle$ and an efficient policy π, we can quickly generate a sequence of states $(s_0, s_1, s_2, \ldots)$ by iteratively applying π, where $s_{i+1} = T(s_i, \pi(\langle s_i, g \rangle))$. We say that π *solves* $\langle s_0, g \rangle$ iff there is an i such that $g \subseteq s_i$. When π solves $\langle s_o, g \rangle$, we define the *solution length* to be the smallest i such that $g \subseteq s_i$.

Given a distribution $\mathcal{P}$ over planning problems, the *success ratio* $\mathrm{SR}(\pi, D, \mathcal{P})$ of π is the probability that π solves a problem drawn from $\mathcal{P}$. Treating $\mathcal{P}$ as a random variable, the *average length* $\mathrm{AL}(\pi, D, \mathcal{P})$ of π is the conditional expectation of the solution length of π on $\mathcal{P}$ given that π solves $\mathcal{P}$. For a given D and $\mathcal{P}$, we are interested in selecting an efficient policy π with "high" $\mathrm{SR}(\pi, D, \mathcal{P})$ and "low" $\mathrm{AL}(\pi, D, \mathcal{P})$. Such a policy represents an efficient domain-specific planner for D that performs well relative to $\mathcal{P}$.

Random Walk Distributions. For state s and set of state predicates G, let $s|_G$ denote the set of facts in s that are applications of a predicate in G. For planning domain D and set of *goal predicates* $G \subseteq P$, we define the *n-step random-walk problem distribution* $\mathcal{RW}_n(D, G)$ by the following stochastic algorithm: First, draw a random state s_0 from the initial state distribution I. Second, starting at s_0 take n uniformly random actions[2] to produce the state sequence $(s_0, s_1, \ldots, s_n)$. At each uniformly random action selection, we assume that an extra "no-op" action (that does not change the state) is selected with some fixed probability, for reasons explained below. Finally, return the planning problem $\langle s_0, s_n|_G \rangle$ as the output. We will sometimes abbreviate $\mathcal{RW}_n(D, G)$ by $\mathcal{RW}_n$ when D and G are clear in

[1] Testing many problem instances from each of many domains.

[2] In practice, we only select random actions from the set of applicable actions in a state s_i, provided our simulator makes it possible to identify this set.

context.

Intuitively, to perform well on this distribution a policy must be able to achieve facts involving the goal predicates that typically result after an n-step random walk from an initial state. By restricting the set of goal predicates G we can specify the types of facts that we are interested in achieving—e.g. in the blocks world we may only be interested in achieving facts involving the "on" predicate.

The random-walk distributions provide a natural way to span a range of problem difficulties. Since longer random walks tend to take us "further" from an initial state, for small n we typically expect that the planning problems generated by $\mathcal{RW}_n$ will become more difficult as n grows. However, as n becomes large, the problems generated will require far fewer than n steps to solve—i.e. there will be "more direct" paths from an initial state to the end state of a long random walk. Eventually, since S is finite, the problem difficulty will stop increasing with n.

A question raised by this idea is whether, for large n, good performance on $\mathcal{RW}_n$ ensures good performance on other problem distributions of interest in the domain. In some domains, such as the simple blocks world[3], good random-walk performance does seem to yield good performance on other distributions of interest. In other domains, such as the grid world (with keys and locked doors), intuitively, a random walk is very unlikely to uncover a problem that requires unlocking a door.

We believe that good performance on long random walks is often useful, but is only addressing one component of the difficulty of many planning benchmarks. To successfully address problems with other components of difficulty, a planner will need to deploy orthogonal technology such as landmark extraction for setting subgoals (Porteous, Sebastia, & Hoffmann 2001; Zhu & Givan 2003). For example, in the grid world, if orthogonal technology can set the subgoal of possessing a key for the first door, a long random-walk policy could provide a useful macro for getting that key.

Here, we limit our focus to finding good policies for long random walks, i.e. the problem of *long random walk (LRW) policy selection.* In our experiments, we define "long" by specifying a large walk length N. Theoretically, the inclusion of the "no-op" action in the definition of $\mathcal{RW}$ ensures that the induced random-walk Markov chain[4] is aperiodic, and thus that the distribution over states reached by increasingly long random walks converges to a stationary distribution[5]. Thus $\mathcal{RW}_* = \lim_{n\to\infty} \mathcal{RW}_n$ is well-defined, and we take good performance on $\mathcal{RW}_*$ to be our goal.

Learning from Random Walks

Our system utilizes a combination of machine learning and simulation, and is based on our recent work (Fern, Yoon, & Givan 2003) on approximate policy iteration (API) for relational Markov decision processes (MDPs). Below, we first give an overview of policy iteration in our setting, describe our approximate form of policy iteration, and review the challenges of applying API to LRW policy selection. We then describe "random-walk bootstrapping" for API.

Policy Iteration. Starting with an initial policy, (exact) *policy iteration* (Howard 1960) iterates a policy improvement operator that guarantees reaching a fixed point that is a guaranteed optimal policy. Given a policy π and a planning domain D, exact policy improvement requires two steps. First, we calculate the Q-cost function $Q^{\pi}_{D}(\langle s, g\rangle, a)$, which here is the number of steps required to achieve goal g after taking action a in state s and then following π.[6] Second, we compute a new improved policy π' by greedily choosing, for problem $\langle s, g\rangle$, the action a that maximizes $Q^{\pi}_{D}(\langle s, g\rangle, a)$. The policy π' is given by $\pi'(\langle s, g\rangle) = \text{argmin}_{a\in A} Q^{\pi}_{D}(\langle s, g\rangle, a)$. Clearly, since exact policy improvement requires enumerating all planning problems $\langle s, g\rangle$, it is impractical for STRIPS/ADL domains.

Approximate Policy Improvement. API (Bertsekas & Tsitsiklis 1996) heuristically approximates policy iteration by iterating an approximate, rather than exact, policy improvement operator. This operator uses simulation to estimate sample Q^{π}_{D} values, and the generalization ability of machine learning to approximate π' given this sample; thus avoiding enumerating all planning problems. As a result, API can be applied to domains with very large state spaces, although it is not guaranteed to converge due to the possibility of unsound generalization. In practice, API often does "converge" usefully, e.g. (Tesauro 1992; Tsitsiklis & Van Roy 1996).

Given a policy π, a planning domain D, a heuristic function H from states to real numbers, and a problem distribution $\mathcal{P}$, an approximate policy improvement operator $\textsc{ImprovePolicy}(\pi, D, H, \mathcal{P})$ returns an approximately improved policy $\widehat{\pi}$. In our STRIPS/ADL setting, this means that the success ratio and average length of $\widehat{\pi}$ will (approximately) be better (or no worse) than that of π.[7] The input heuristic function H is used to provide additional guidance toward improvement when the policy cannot reach the goal at all: rather than minimizing average length, the method attempts to minimize average length plus heuristic value at the horizon (if the goal is reached before the horizon, the heuristic contribution will be zero). However, H may be the constant zero function when no heuristic is available. In our STRIPS/ADL experiments we use the FF heuristic function (Hoffmann & Nebel 2001).

[3] In the blocks world with $G = P$ and large n, $\mathcal{RW}_n$ generates various pairs of random block configurations, typically pairing states that are far apart—clearly, a policy that performs well on this distribution has captured significant information about the blocks world.

[4] We don't formalize this chain here, for space reasons, but various formalizations work well.

[5] The Markov chain may not be irreducible, so the same stationary distribution may not be reached from all initial states; however, we are only considering one initial state, described by I.

[6] To avoid infinite Q-costs we can either use discounting or a finite-horizon bound on the number of steps.

[7] Our actual MDP formulation evaluates policies based on finite-horizon cost. In practice and intuitively, this measure corresponds well with success ratio and average length, but it is difficult to determine the exact relationship among these measures without additional assumptions.

Prior to our recent work (Fern, Yoon, & Givan 2003), existing variants of API performed poorly in large relational domains such as common STRIPS/ADL benchmark domains. Our contribution was to introduce a new IMPROVEPOLICY operator, that we use here, that is suited to relational domains.[8] As shown in Figure 1, our IMPROVEPOLICY$(\pi, D, H, \mathcal{P})$ operates in two steps. First, in DRAWTRAININGSET(), we draw a set of planning problems from $\mathcal{P}$ and then for each problem $\langle s, g\rangle$ we use *policy rollout* (Tesauro & Galperin 1996) to compute the Q-cost $Q^{\pi}_{D}(\langle s, g\rangle, a)$ for all (applicable) actions a, by applying action a in s and then simulating π. If π does not reach the goal within some (human-specified, problem-specific) horizon, arriving instead in state s', we approximate the Q-cost by adding $H(s')$ to the horizon length. Using the Q-costs we can determine $\pi'(\langle s, g\rangle)$ for each problem in our set. Second, in LEARNPOLICY(), guided by the training data from the first step, we use standard machine learning techniques for cost-sensitive classification to search for a compactly represented approximation $\widehat{\pi}$ to π'. For a detailed description of this API variant for general MDPs, see (Fern, Yoon, & Givan 2003).

There are two issues that are critical to the success of IMPROVEPOLICY. The first issue, which this paper addresses, is that IMPROVEPOLICY$(\pi, D, H, \mathcal{P})$ can only yield improvement if its inputs provide enough guidance. In our setting this may correspond to π occasionally reaching the goal and/or H providing non-trivial goal-distance information for problems drawn from $\mathcal{P}$. We call this "bootstrapping" because, for many domains, it appears that once a (possibly poor) policy is found that can reach the goal for a non-trivial fraction of the problems drawn from $\mathcal{P}$, API is quite effective at improving the policy. The problem is getting this first somewhat successful policy.

For example, suppose $\mathcal{P}$ is uniform on 20-block blocks world problems and that π is random and H is trivial. In this case, IMPROVEPOLICY has no hope of finding a better policy since the generated training set will essentially never provide information about how to reach or move closer to a goal. Because we are interested in solving large domains such as this, providing "guiding inputs" to IMPROVEPOLICY is critical. In (Fern, Yoon, & Givan 2003), we showed that by using FF's heuristic to "bootstrap" API, we were able to use IMPROVEPOLICY to uncover good policies for the blocks world, simplified logistics world (no planes), and stochastic variants. This type of bootstrapping, however, does not work well in many other benchmark domains, and worked poorly in the blocks world with more than 10 blocks. In this work, we contribute a new bootstrapping procedure, based on random walks, for guiding API toward good policies for LRW distributions.

The second critical issue, which is not the focus of this paper, is that IMPROVEPOLICY is fundamentally limited by the expressiveness of the policy language and the strength of the learner (together, these represent the "bias" of the learner). In particular, we must manage the conflicting tasks of selecting a policy language that is expressive enough to represent good policies, and building an effective learner for that language with good generalization beyond the training data. Since our goal is to have a fully-automated system, we give a domain-independent specification of the language and learner. As described in detail in the appendix, we specify policies as decision lists of action-selection rules built from the state predicates and action types provided by the planning domain, using a "taxonomic" knowledge representation similar to description logic. We then use standard decision-list learning to select such policies.

Indeed, our experimental results suggest that, in some domains, a primary limiting factor for our current system is an inadequately expressive combination of language and learner. In particular, for some domains (e.g., Freecell), the authors themselves cannot write a good policy in our current policy language—indicating likely difficulty for the learner to find one. This paper does not attempt to address this weakness, since it is somewhat orthogonal to the issue of bootstrapping; instead, we take a generic approach to learning, borrowed from our previous work. We plan future research focused on this issue that is expected to improve our system further. One reason to avoid redesigning the policy language here is to ensure that we do not solve each domain by providing hidden human assistance in the form of policy language customization (which resembles the feature-engineering often necessary for cost-function–based API).

Random-Walk Bootstrapping. Given a planning domain D and set of goal predicates G, our system attempts to find a good policy for $\mathcal{RW}_N$, where N is selected to be large enough to adequately approximate $\mathcal{RW}_*$, while still allowing tractable completion of the learning. Naively, given an initial policy π_0 and a heuristic H, we could try to apply API directly by computing a sequence of policies $\pi_{i+1} \leftarrow$ IMPROVEPOLICY$(\pi_i, D, H, \mathcal{RW}_N)$. As already discussed, this will not work in general, since we are interested in planning domains where $\mathcal{RW}_*$ produces extremely large and difficult problems where we cannot assume the availability of either a domain-independent π_0 or domain-independent H that are sufficient to bootstrap API.

However, for very small n (e.g. $n = 1$), $\mathcal{RW}_n$ typically generates easy problems, and it is likely that API, starting with even a random initial policy, can reliably find a good policy for $\mathcal{RW}_n$. Furthermore, we expect that if a policy π_n performs well on $\mathcal{RW}_n$, then it will also provide "reasonably good", but perhaps not perfect, guidance on problems drawn from $\mathcal{RW}_m$ when m is only "moderately larger" than n. Thus, we expect to be able to find a good policy for $\mathcal{RW}_m$ by bootstrapping API with initial policy π_n. This suggests a natural iterative bootstrapping technique to find a good policy for large n (in particular, for $n = N$).

The pseudo-code for our algorithm LRW-LEARN is given in Figure 1. Intuitively, this is an "anytime" algorithm that iterates through two stages: first, finding a "hard enough" distribution for the current policy (by increasing n); and,

[8] The primary difficulty with previous variants was that the operator IMPROVEPOLICY was based on learning approximate cost-functions and STRIPS/ADL domains typically have extremely complicated cost functions. Our operator that completely avoids cost-function learning and instead learns policies directly in a compact language. In relational domains, policies are often much simpler than their cost functions.

```
LRW-LEARN ($D, G, H, \pi_0, N$)
// planning domain $D$, goal predicates $G$,
// heuristic function $H$, initial policy $\pi_0$, max walk length $N$.
$\pi \leftarrow \pi_0$;   $n \leftarrow 1$;
loop
    if $\widehat{\text{SR}}_\pi(n) > \tau$
        // Find harder $n$-step distribution for $\pi$.
        $n \leftarrow$ least $i \in [n, N]$ s.t. $\widehat{\text{SR}}_\pi(i) < \tau - \delta$, or $N$ if none;
    $\pi \leftarrow$ IMPROVEPOLICY($\pi, D, H, \mathcal{RW}_n(D, G)$);
until satisfied with $\pi$ or progress stops
Return $\pi$;
```

```
IMPROVEPOLICY ($\pi, D, H, \mathcal{P}$)
$T \leftarrow$ DRAWTRAININGSET($\pi, D, H, \mathcal{P}$);     // Describes $\pi'$
$\widehat{\pi} \leftarrow$ LEARNPOLICY($T$);                   // Approximates $\pi'$
Return $\widehat{\pi}$;
```

Figure 1: Pseudo-code for LRW-LEARN. $\widehat{\text{SR}}_\pi(n)$ estimates the success ratio of π in planning domain D on problems drawn from $\mathcal{RW}_n(D, G)$ by drawing a set of problems and returning the fraction solved by π. Constants τ and δ, and functions DRAWTRAININGSET and LEARNPOLICY are described in the text.

then, finding a good policy for the hard distribution using API. The algorithm maintains a current policy π and current walk length n (initially, $n = 1$). As long as the success ratio of π on RW_n is below the *success threshold* τ, which is a constant close to one, we simply iterate steps of approximate policy improvement. Once we achieve a success ratio of τ with some policy π, the if-statement increases n until the success ratio of π on $\mathcal{RW}_n$ falls below $\tau - \delta$. That is, when π performs well enough on the current n-step distribution we move on to a distribution that is "slightly" harder. The constant δ determines how much harder and is set small enough so that π can likely be used to bootstrap policy improvement on the harder distribution. (The simpler method of just increasing n by 1 whenever success ratio τ is achieved will also find good policies whenever this method does. This can take much longer, as it may run API repeatedly on a training sets for which we already have a good policy.)

Once n becomes equal to the maximum walk length N, we will have $n = N$ for all future iterations. It is important to note that even after we find a policy with a good success ratio on $\mathcal{RW}_N$ it may still be possible to improve on the average length of the policy. Thus, we continue to iterate policy improvement on this distribution until we are satisfied with both the success ratio and average length of the current policy.

Experiments

We perform experiments in seven familiar STRIPS/ADL planning domains: Blocks World, Freecell, Logistics, Schedule, Elevator, Gripper, and Briefcase. These domains represent the union of the STRIPS/ADL domains from the AIPS-2000 competition and those used to evaluate TL-Plan in (Bacchus & Kabanza 2000), along with the Gripper domain.

Blocks World

iter. #	n	$\mathcal{RW}_n$ SR	$\mathcal{RW}_n$ AL	$\mathcal{RW}_*$ SR	$\mathcal{RW}_*$ AL
1	4	0.92	2.0	0	0
2	14	0.94	5.6	0.10	41.4
3	54	0.56	15.0	0.17	42.8
4	54	0.78	15.0	0.32	40.2
5	54	0.88	33.7	0.65	47.0
6	54	0.98	25.1	0.90	43.9
7	334	0.84	45.6	0.87	50.1
8	334	0.99	37.8	1	43.3
FF				0.96	49.0

Freecell

iter. #	n	$\mathcal{RW}_n$ SR	$\mathcal{RW}_n$ AL	$\mathcal{RW}_*$ SR	$\mathcal{RW}_*$ AL
1	5	0.97	1.4	0.08	3.6
2	8	0.97	2.7	0.26	6.3
3	30	0.65	7.0	0.78	7.0
4	30	0.72	7.1	0.85	7.0
5	30	0.90	6.7	0.85	6.3
6	30	0.81	6.7	0.89	6.6
7	30	0.78	6.8	0.87	6.8
8	30	0.90	6.9	0.89	6.6
9	30	0.93	7.7	0.93	7.9
FF				1	5.4

Elevator

iter. #	n	$\mathcal{RW}_n$ SR	$\mathcal{RW}_n$ AL	$\mathcal{RW}_*$ SR	$\mathcal{RW}_*$ AL
1	20	1	4.0	1	26
FF				1	23

Gripper

iter. #	n	$\mathcal{RW}_n$ SR	$\mathcal{RW}_n$ AL	$\mathcal{RW}_*$ SR	$\mathcal{RW}_*$ AL
1	10	1	3.8	1	13
FF				1	13

Logistics

iter. #	n	$\mathcal{RW}_n$ SR	$\mathcal{RW}_n$ AL	$\mathcal{RW}_*$ SR	$\mathcal{RW}_*$ AL
1	5	0.86	3.1	0.25	11.3
2	45	0.86	6.5	0.28	7.2
3	45	0.81	6.9	0.31	8.4
4	45	0.86	6.8	0.28	8.9
5	45	0.76	6.1	0.28	7.8
6	45	0.76	5.9	0.32	8.4
7	45	0.86	6.2	0.39	9.1
8	45	0.76	6.9	0.31	11.0
9	45	0.70	6.1	0.19	7.8
10	45	0.81	6.1	0.25	7.6
...	...	...	...	...	...
43	45	0.74	6.4	0.25	9.0
44	45	0.90	6.9	0.39	9.3
45	45	0.92	6.6	0.38	9.4
46	70	**	**	**	**
FF				1	13

Schedule

iter. #	n	$\mathcal{RW}_n$ SR	$\mathcal{RW}_n$ AL	$\mathcal{RW}_*$ SR	$\mathcal{RW}_*$ AL
1	1	0.93	1	0.1	14.3
2	5	0.89	1.59	0.1	14.4
3	5	1	1.68	1	13.0
FF				1	13

Briefcase

iter. #	n	$\mathcal{RW}_n$ SR	$\mathcal{RW}_n$ AL	$\mathcal{RW}_*$ SR	$\mathcal{RW}_*$ AL
1	5	0.91	1.4	0	0
2	15	0.89	4.2	0.2	38
3	15	1	3.0	1	30
FF				1	28

Figure 2: Results for each iteration of LRW-LEARN in seven planning domains. For each iteration, we show the walk length n used for learning, along with the success ratio (SR) and average length (AL) of the learned policy on both $\mathcal{RW}_n$ and $\mathcal{RW}_*$. The final policy shown in each domain performs above $\tau = 0.9$ SR on walks of length $N = 10,000$, and further iteration does not improve the performance. The exception is Logistics, where the large number of iterations required exhausted the CPU time available at the time of this submission. For each benchmark we also show the SR and AL of FF on problems drawn from $\mathcal{RW}_*$.

LRW Experiments. Our first set of experiments evaluates the ability of LRW-LEARN to find good policies for $\mathcal{RW}_*$. We provided LRW-LEARN with the FF heuristic function H and an initial policy π_0 that corresponded to a one-step–look-ahead greedy search based on the FF heuristic. The maximum-walk-length parameter N was set to be 10,000 for all experiments, with τ equal to 0.9 and δ equal to 0.1. In each iteration, DRAWTRAININGSET generates a training set constructed from 100 problems. Recall that in each iteration of LRW-LEARN we compute an (approximately) improved policy and may also increase the walk length n to find a harder problem distribution. We continued iterating LRW-Learn until we observed no further improvement. The training time per iteration is approximately five hours. Though the initial training period is signficant, once a policy is learned it can be used to solve new problems very quickly, terminating in seconds with a solutions when one is found, even for very large problems.

Figure 2 provides data for each iteration of LRW-Learn in each of the seven domains[9]. The first column, for each domain, indicates the iteration number (e.g. the Blocks World was run for 8 iterations). The second column records the walk length n used for learning in the corresponding iteration. The third and fourth columns record the SR and AL of the policy learned at the corresponding iteration as measured on 100 problems drawn from $\mathcal{RW}_n$ for the corresponding value of n (i.e. the distribution used for learning). When this SR exceeds τ, the next iteration seeks an increased walk length n. The fifth and sixth columns record the SR and AL of the same policy, but measured on 100 problems drawn from the LRW target distribution $\mathcal{RW}_*$, which in these experiments is approximated by $\mathcal{RW}_N$ for $N = 10,000$.

So, for example, we see that in the Blocks World there are a total of 8 iterations, where we learn at first for one iteration with $n = 4$, one more iteration with $n = 14$, four iterations with $n = 54$, and then two iterations with $n = 334$. At this point we see that the resulting policy performs well on $\mathcal{RW}_*$. Further iterations with $n = N$, not shown, showed no improvement over the policy found after iteration eight. In other domains, we also observed no improvement after iterating with $n = N$, and thus do not show those iterations. We note that all domains except Logistics (see below) achieve policies with good performance on $\mathcal{RW}_N$ by learning on much shorter $\mathcal{RW}_n$ distributions, indicating that we have indeed selected a large enough value of N to capture $\mathcal{RW}_*$, as desired.

Comments on Figure 2 Results. For several domains, our learner bootstraps very quickly from short random-walk problems, finding a policy that works well even for much longer random-walk problems. These include Schedule, Briefcase, Gripper, and Elevator. Typically, large problems in these domains have many somewhat independent subproblems with short solutions, so that short random walks can generate instances of all the different typical subproblems. In each of these domains, our best LRW policy is found in a small number of iterations and performs comparably to FF on $\mathcal{RW}_*$. We note that FF is considered a very good domain-independent planner for these domains, so we consider this a successful result.

For two domains, Logistics[10] and Freecell, our planner is unable to find a policy with success ratio one on $\mathcal{RW}_*$. We believe that this is a direct result of the limited knowledge representation we allowed for policies for the following reasons. First, we ourselves cannot write good policies for these domains within our current policy language. Second, the success ratio (not shown) for the sampling-based rollout policy[11] π' is substantially higher than that for the resulting learned policy $\widehat{\pi}$ that becomes the policy of the next iteration. This indicates that LearnPolicy is learning a much weaker policy than the sampling-based policy generating its training data, indicating a weakness in either the policy language or the learning algorithm (or possibly too small a training set). For example, in the logistics domain, at iteration eight, the training data for learning the iteration-nine policy is generated by a sampling rollout policy that achieves success ratio 0.97 on 100 training problems drawn from the same $\mathcal{RW}_{45}$ distribution, but the learned iteration-nine policy only achieves success ratio 0.70, as shown in the figure at iteration nine. Because of these limitations, it is not surprising that FF outperforms our learned policy on $\mathcal{RW}_*$.

		π_*		FF	
Domain	Size	SR	AL	SR	AL
Blocks	(20 blocks)	1	54	0.81	60
	(50 blocks)	1	151	0.28	158
Freecell	(8 cards)	0.36	15	1	10
	(52 cards)	0	—	0.47	112
Logistics	(6 packages)	0.87	6	1	6
	(30 packages)	0	—	1	158
Elevator	(30 people)	1	112	1	98
Schedule	(50 parts)	1	174	1	212
Briefcase	(10 objects)	1	30	1	29
	(50 objects)	1	162	0	—
Gripper	(40-60 balls)	1	150	1	150

Figure 3: Results on "standard" problem distributions for seven benchmarks. Success ratio (SR) and average length (AL) are provided for both FF and our policy learned for the LRW problem distribution. For a given domain, the same learned LRW policy is used for each problem size shown.

In the remaining domain, the Blocks World, the bootstrapping provided by increasingly long random walks appears particularly useful. The policies learned at each of the walk lengths 4, 14, 54, and 334 are increasingly effective on target LRW distribution $\mathcal{RW}_*$. For walks of length 54 and 334, it takes multiple iterations to master the provided level of difficulty beyond the previous walk length. Finally, upon mastering walk length 334, the resulting policy appears to perform well for any walk length. The learned policy is modestly superior to FF on $\mathcal{RW}_*$ in success ratio and average length.

Evaluation on the Original Problem Distributions. Figure 3 shows results for our best learned LRW policy (denoted π_*)[12] from each domain, in comparison to FF, on the

[9] Learning was conducted using the following domain sizes: 20 Blocks World, 8 card Freecell, 6 package Logistics, 10 person Elevator, 8 part Schedule, and 10 object Briefcase, and 10 ball Gripper.

[10] In Logistics, the planner generates a long sequence of policies with similar, oscillating success ratio that are elided from the table with an ellipsis for space reasons.

[11] The policy described by the training data generated by DrawTrainingSet, but only approximated by LearnPolicy, see Figure 1.

[12] The policy, from each domain, with the highest performance on $\mathcal{RW}_*$, as shown in Figure 2.

original intended problem distributions for those domains. Here we have attempted to select the largest problem sizes previously used in evaluation of domain-specific planners (either in AIPS-2000 or in (Bacchus & Kabanza 2000)), as well as show a smaller problem size for those cases where one of the planners we show performed poorly on the large size. In each case, we use the problem generators provided with the domains, and evaluate on 100 problems of each size used (or 20 problems of similar sizes in the case of Gripper).

Overall, these results indicate that our learned, reactive policies (learned in a domain-independent fashion) are competitive with the domain-independent planner FF. On two domains, Logistics and Freecell, FF substantially outperforms our learned policies on success ratio, for reasons discussed above, especially on large domain sizes. On two other domains, Blocks World and Briefcase, our learned policies substantially outperform FF on success ratio, especially on large domain sizes. On the other domains, the two approaches perform quite similarly on success ratio, with our approach superior in average length on Schedule but FF superior in average length on Elevator.

Related Work

For a collection and survey of work on "learning for planning" see (Minton 1993; Zimmerman & Kambhampati 2003). Two primary approaches are to learn domain-specific control rules for guiding search-based planners e.g. (Minton *et al.* 1989; Veloso *et al.* 1995; Estlin & Mooney 1996; Huang, Selman, & Kautz 2000; Ambite, Knoblock, & Minton 2000; Aler, Borrajo, & Isasi 2002), and, more closely related, to learn domain-specific reactive control policies (Khardon 1999; Martin & Geffner 2000; Yoon, Fern, & Givan 2002).

The ultimate goal of such systems is to allow for planning in large, difficult problems that are beyond the reach of domain-independent planning technology. Clearly, learning to achieve this goal requires some form of bootstrapping and almost all previous systems have relied on the human for this purpose. By far, the most common human-bootstrapping approach is "learning from small problems". Here, the human provides a small problem distribution to the learner, by limiting the number of objects (e.g. using 2-5 blocks in the blocks world), and control knowledge is learned for the small problems. For this approach to work, the human must ensure that the small distribution is such that good control knowledge for the small problems is also good for the large target distribution. In contrast, our approach can be applied without human assistance directly to large planning domains. However, as already pointed out, our goal of performing well on the LRW distribution may not always correspond well with a particular target problem distribution.

Our bootstrapping approach is similar in spirit to the bootstrapping framework of "learning from exercises"(Natarajan 1989; Reddy & Tadepalli 1997). Here, the learner is provided with planning problems, or "exercises", in order of increasing difficulty. After learning on easier problems, the learner is able to use its new knowledge, or "skills", in order to bootstrap learning on the harder problems. This work, however, has previously relied on a human to provide the exercises, which typically requires insight into the planning domain and the underlying form of control knowledge and planner. Our work can be viewed as an automatic instantiation of "learning from exercises", specifically designed for learning LRW policies.

Our random-walk bootstrapping is most similar to the approach used in MICRO-HILLARY (Finkelstein & Markovitch 1998), a macro-learning system for problem solving. In that work, instead of generating problems via random walks starting at an initial state, random walks were generated "backwards" from goal states. This approach assumes that actions are invertible or that we are given a set of "backward actions". When such assumptions hold, the backward random-walk approach may be preferable when we are provided with a goal distribution that does not match well with the goals generated by forward random walks. Of course, in other cases forward random walks may be preferable. MICRO-HILLARY was empirically tested in the $N \times N$ sliding-puzzle domain; however, as discussed in that work, there remain challenges for applying the system to more complex domains with parameterized actions and recursive structure, such as familiar STRIPS/ADL domains. To the best of our knowledge, the idea of learning from random walks has not been previously explored in the context of STRIPS/ADL planning domains.

Conclusion

Our evaluation demonstrates that, with random-walk bootstrapping, our system is often able to select good control knowledge (i.e., a good policy) for familiar planning benchmarks. The results point to an immediate direction for improvement—most significantly, extensions to the policy language and corresponding learner are needed. Our immediate goal is to show that with these extensions we can succeed across an even wider range of planning benchmarks—in particular, benchmarks where search-based domain-independent planners fail. Policy-language extensions that we are considering include various extensions to the knowledge representation used to represent sets of objects in the domain (in particular, for route-finding in maps/grids), as well as non-reactive policies that incorporate search into decision-making.

Acknowledgements

We would like to thank Lin Zhu for originally suggesting the idea of using random walks for bootstrapping. This work was supported in part by NSF grants 9977981-IIS and 0093100-IIS.

Appendix: Policy Language

For single argument actions, useful rules often take the form "apply action type a to any object in set C"; e.g., "unload any object that is at its destination". (Martin & Geffner 2000) introduced decision lists of such rules as a language bias for learning policies. Here we use a similar rule form, but generalized to handle multiple arguments.

Each action-section rule has the form: $A : L_1, L_2, \ldots L_m$, where A is an m-argument action type, and the L_i are literals. Literals have the form $x_i \in C_i(x_1, \ldots, x_m)$, where each x_i is an action-argument variable and the C_i are set expressions expressed in an enriched taxonomic syntax (McAllester & Givan 1993), defined by

$$\begin{aligned} C &::= C_0 \mid \textbf{a-thing} \mid \neg C \mid (R\ C) \mid C \cap C \mid (\min R) \\ R &::= R_0 \mid R^{-1} \mid R \cap R \mid R^*. \end{aligned}$$

Here, C_0 is any one argument predicate, or one of the x_i variables, and R_0 any binary predicate from the predicates in P. One argument predicates denote the set of objects that they are true of, $(R\ C)$ denotes the image of the objects in class C under the binary predicate R, $(\min R)$ denotes the class of minimal elements under the binary predicate R, and for the (natural) semantics of the other constructs shown, please refer to (Yoon, Fern, & Givan 2002). A new predicate symbol is included for each predicate in G, to represent the desired goal state; e.g., **gclear**(x) represent that x is clear in the goal. Given a planning problem $\langle s, g\rangle$ and a concept C expressed in this syntax, it is straightforward to compute the set of domain objects that are represented by C in $\langle s, g\rangle$, in order to execute the policy. Predicates of three or more arguments are represented with multiple introduced auxiliary binary predicates.

For a particular planning problem we say that a rule *allows* action $A(o_1, \ldots o_m)$, where the o_i are objects, iff each literal is true when the variables are instantiated with the objects. That is, $o_i \in C_i(o_1, \ldots, o_m)$ is true for each i. Thus, a rule places mutual constraints on the tuples of objects that an action type can be applied to. Given a list of such rules we say that an action is allowed by the list if it is allowed by some rule in the list, and no previous rule allows any actions. Given a planning problem and a decision-list policy, the policy selects the lexicographically least allowed action.

References

Aler, R.; Borrajo, D.; and Isasi, P. 2002. Using genetic programming to learn and improve control knowledge. *Artificial Intelligence* 141(1-2):29–56.

Ambite, J. L.; Knoblock, C. A.; and Minton, S. 2000. Learning plan rewriting rules. In *Artificial Intelligence Planning Systems*, 3–12.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116:123–191.

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

Estlin, T. A., and Mooney, R. J. 1996. Multi-strategy learning of search control for partial-order planning. In *13th National Conference on Artificial Intelligence*, 843–848.

Fern, A.; Yoon, S.; and Givan, R. 2003. Approximate policy iteration with a policy language bias. In *16th Conference on Advances in Neural Information Processing*.

Finkelstein, L., and Markovitch, S. 1998. A selective macro-learning algorithm and its application to the NxN sliding-tile puzzle. *Journal of Artificial Intelligence Research* 8:223–263.

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

Howard, R. 1960. *Dynamic Programming and Markov Decision Processes*. MIT Press.

Huang, Y.-C.; Selman, B.; and Kautz, H. 2000. Learning declarative control rules for constraint-based planning. In *17th International Conferebce on Machine Learning*, 415–422. Morgan Kaufmann, San Francisco, CA.

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

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

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

Minton, S.; Carbonell, J.; Knoblock, C. A.; Kuokka, D. R.; Etzioni, O.; and Gil, Y. 1989. Explanation-based learning: A problem solving perspective. *Artificial Intelligence* 40:63–118.

Minton, S., ed. 1993. *Machine Learning Methods for Planning*. Morgan Kaufmann Publishers.

Natarajan, B. K. 1989. On learning from exercises. In *Annual Workshop on Computational Learning Theory*.

Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. Shop: Simple hierarchical ordered planner. In *International Joint Conference on Artificial Intelligence*, 968–973.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In *6th European Conference on Planning*, 37–48.

Reddy, C., and Tadepalli, P. 1997. Learning goal-decomposition rules using exercises. In *International Conference on Machine Learning*, 278–286.

Tesauro, G., and Galperin, G. R. 1996. On-line policy improvement using monte-carlo search. In *9th Conference on Advances in Neural Information Processing*.

Tesauro, G. 1992. Practical issues in temporal difference learning. *Machine Learning* 8:257–277.

Tsitsiklis, J. N., and Van Roy, B. 1996. Feature-based methods for large scale dynamic programming. *Machine Learning* 22:59–94.

Veloso, M.; Carbonell, J.; Perez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence* 7(1).

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

Zhu, L., and Givan, R. 2003. Landmark Extraction via Planning Graph Propagation. In *ICAPS Doctoral Consortium*.

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