Learning Reactive Policies for Probabilistic Planning Domains

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

Abstract

We present a planning system for selecting policies in probabilistic planning domains. Our system is based on a variant of approximate policy iteration that combines inductive machine learning and simulation to perform policy improvement. Given a planning domain, the system iteratively improves the best policy found so far until no more improvement is observed or a time limit is exceeded. Though this process can be computationally intensive, the result is a reactive policy, which can then be used to quickly solve future problem instances from the planning domain. In this way, the resulting policy can be viewed as a domain-specific reactive planner for the planning domain, though it is discovered with a domain-independent technique. Thus, the initial cost of finding the policy is amortized over future problem-solving experience in the domain. Due to the system's inductive nature, there are no performance guarantees for the selected policies. However, empirically our system has shown state-of-the-art performance in a number of benchmark planning domains, both deterministic and stochastic.

Introduction

We view a planning domain (e.g. as specified via PPDDL) as a Markov Decision Process (MDP) where there is an MDP state for each possible problem instance in the domain. Viewed as such, a solution to the MDP, i.e. a policy, is a mapping from problem instances to domain actions. For goal-based domains, such a policy can be viewed as specifying what action to take given the current domain state and current goal. A good policy will select actions so as to minimize the expected cost of reaching the goal.

Typically the MDP corresponding to a PPDDL domain has far too many states to support solution via flat state-space MDP techniques. To deal with large state spaces we base our system on a form of approximate policy iteration (API), which does not rely on state-space enumeration. Most existing frameworks for API (e.g. (Bertsekas & Tsitsiklis 1996)) represent policies indirectly via value functions and use machine learning to select value function approximations. However, in many domains, particularly those with relational (first-order) structure, representing and learning value functions is much more complicated than representing and learning policies directly. Based on this observation, our system utilizes a new variant of API (Fern, Yoon, & Givan 2003), which represents policies directly as state/action mappings.

The performance of our system depends on two critical issues. First, we must provide a policy language and associated learner that allow the system to find approximations of good policies. Second, for complex domains, it is necessary to provide a mechanism to bootstrap the API process. Below we describe the choices we have made to deal with these issues in our current system.

In what follows we first provide an overview of API. Next we discuss the policy representation language and learning technique used in our system. Finally, we give an overview of our bootstrapping technique. A more detailed treatment of our algorithms can be found in (Fern, Yoon, & Givan 2003; 2004).

Approximate Policy Iteration

Figure 1 shows the core components of our system's API engine. Each iteration of API consists of two primary stages: *policy evaluation* and *policy selection*. Intuitively, policy evaluation uses simulation to produce a training set that describes an improved policy with respect to the current policy. Policy selection then uses machine learning to find an approximation of the improved policy based on the training set. Thus, if we are given a current policy and then apply these steps in sequence, the result is an (approximately) improved policy. Our system iterates these steps until no more improvement is observed.

Policy Evaluation. Policy evaluation is carried out via the simulation technique of *policy rollout* (Bertsekas & Tsitsiklis 1996). The policy-rollout component first draws a set of problem instances (which can also be viewed as MDP states) from the provided problem generator.[1] Next, for each problem instance I and each action a available in I, simulation is used to estimate the Q-value $Q(I, a, \pi)$ of the current

[1]Even when a problem generator is not provided for a planning domain, we can still use API to solve individual problem instances. Given an individual problem instance to be solved, we simply create a trivial problem generator that always returns that problem instance.

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

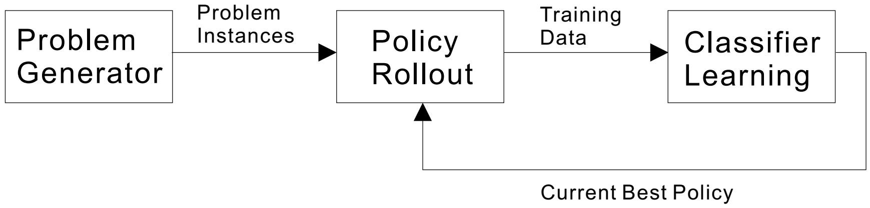

Figure 1: Block diagram of approximation policy iteration. We assume each planning domain provides a problem generator and our goal is to produce a policy that performs well on problem instances drawn from the generator. Given the current best policy, the policy-rollout component creates a training set that describes an improved policy as evaluated on problems drawn from the generator. The classifier learner than analyzes this training set and outputs an approximately improved policy.

policy π, which is simply the expected value of taking action a in I and then following π until a terminating state is reached or a horizon limit is exceeded.

It is straightforward to compute a new improved policy π' from the Q-values of policy π. In particular, it is a basic property of MDPs that π' defined as $\pi'(I) = \text{argmax}_a Q(I, a, \pi)$ is guaranteed to improve upon π, if improvement is possible. Thus, for each of our sample problem instances, the estimates of $Q(I, a, \pi)$ can be used to calculate $\pi'(I)$, that is an "improved action" for problem instance I. Intuitively, the pairs $\langle I, \pi'(I)\rangle$ can be viewed as training examples for learning an approximation of π'. To support such learning, the output of the policy-rollout component is a training set, with one training instance $\langle I, Q(I, a_1, \pi), \ldots, Q(I, a_n, \pi)\rangle$ for each instance I drawn from the problem generator. Please refer to (Fern, Yoon, & Givan 2003) for more details.

Policy Selection. Policy selection is carried out by the classifier-learning component of our system. Note that a policy can be viewed as a classifier that maps problem instances (i.e. states) to actions. The training set obtained from policy rollout is used to learn an (approximately) improved policy. Given a language for compactly representing policies, the job of the classifier learner is to select a policy within that language that chooses actions with high Q-value for problem instances in the training set. With a proper language bias, such policies also tend to select good actions in problem instances outside of the training set. In the next section we give an overview of the policy description language and the corresponding learner used in our system.

Compute Time. In our current system, the compuatation time of API is mostly consumed by generating training sets via policy rollout. This is particularly the case for domains where problem instances contain many ground actions, as multiple trajectories must be simulated for each ground action in each problem instance encountered. Presently the rollout component is implemented in Scheme, hence one way to significantly improve runtime is to provide a C implementation. We are also working to exploit the independence of the rollout trajectories with a parallel implementation. If completed, this speedup may be in effect for our competition entry.

Representing and Learning Policies

For API to succeed, we must provide an adequate language for representing good policies in a domain, and an associated learner that can find good policies, in that language, based on the guidance provided by the rollout training sets.

One of our primary interests is in applying our system to relationally structured planning domains, such as the blocks world, where problem instances are described by specifying a domain of objects (e.g. a set of blocks) and relations among the objects. Thus, it is critical that we provide a policy language that leverages the relational structure in order to generalize across problem instances with different sets of objects. For example, our language needs to represent policies that can be applied to any problem instance of the blocks worlds, regardless of the number and identity of blocks. In order to represent such "generalized policies" we draw upon ideas from the knowledge-representation community, using a language based on taxonomic syntax.

Policy Representation. Our policy representation is an ordered list of rules. The head of each rule is a variablized action type such as **pickup**(?a). The body of each rule specifies a conjunction of constraints on the "object variables" in the head, which indicate when an action should be applied. Given a problem instance, we say that a rule suggests an action if: 1) the action is the result of replacing the object variables in the head with objects from the problem instance, and 2) those objects satisfy the appropriate constraints in the body. The action selected by an ordered list of rules (i.e. a policy) is equal to the action chosen by the earliest rule that selects an action.

The object constraints in a rule's body are represented via taxonomic syntax expressions, which are constructed from the predicate symbols of the planning domain and object variables in the rule's head. As an example policy, consider a blocks-world domain where the goal is always to clear off block A. We can represent an optimal policy in our taxonomic representation as follows.

$$\begin{aligned} \textbf{pickup}(\text{?a}) &: (\text{?a} \in \text{on}^* \text{ A}) \wedge (\text{?a} \in \text{clear}) \\ \textbf{putdown}(\text{?a}) &: \text{?a} \in \text{holding} \end{aligned}$$

The first rule indicates that we should "pick up a clear block which is above block A". The second rule says that we should "put down any block that is being held".

For a detailed description of the syntax and semantics of our policy language please refer to the appendix of (Fern, Yoon, & Givan 2004).

Learning. Recall that each training instance is of the form $\langle I, Q(I, a_1, \pi), \ldots, Q(I, a_n, \pi)\rangle$, where I is a problem instance and the $Q(I, a_i, \pi)$ are the associated Q-values. The goal of the learner is to select a list of rules such that the actions chosen by the corresponding policy results in high Q-value over the training data. Ideally the learned policy should always select an action corresponding to the largest Q-value.

We use a simple greedy covering strategy for learning lists of taxonomic rules. We add one rule to the list at a time until the resulting policy covers all of the training data (i.e. the policy selects an action for every problem instance in the training data). Each rule is learned by greedily adding object constraints to the body according to a heuristic measure that attempts to balance the coverage and quality of a rule. For more information on the learner, please refer to (Yoon, Fern, & Givan 2002) and (Fern, Yoon, & Givan 2003).

Bootstrapping from Random Walks

API must be initialized with a base policy from which iterative policy improvement begins. Since our objective is to have a domain-independent system, we use the random policy as the default base policy in our system. However, for many planning domains it is unlikely that a random policy will achieve any non-trivial reward in problem instances drawn from the provided problem generator. For example, in a blocks world with even a relatively small number of blocks, it is unlikely that a random policy will achieve the goal configuration. As a result, in such domains, API will tend to fail when initialized with a random base policy. The primary reason for the failure is that the Q-values for each action under the random policy will tend to be equal. Thus, the rollout training set, which is based on the Q-values, will not provide the learner with useful guidance as to what actions are desirable.

Our current approach to this problem is to utilize a new bootstrapping technique (Fern, Yoon, & Givan 2004). Rather than initially drive API with the original problem generator (which generates difficult problems), we instead automatically construct an new problem generator that generates easier problems. We then increase the problem difficulty in accordance with the quality of the current best policy found by API. Below we describe this process for goal-based domains. Our current system does not provide a bootstrapping mechanism for non-goal-based domains.

We generate problem instances of varying difficulty by performing random walks in the planning domain. To construct a single problem instance from a planning domain, we first draw a problem instance from the original problem generator. In a goal-based setting, such a problem instances will specify an (initial) domain state s and a goal. Next, starting at s, we take a sequence of n random actions (i.e. an n-step random walk) and observe the resulting state g. We construct a new problem instance with initial state s and goal g. When n is small, such problem instances are relatively easy to solve and we can learn a policy to solve all such problem instances using API starting with a random base policy.

Once we learn a policy for "random-walk problems" with small n, we increase the value of n until the current policy performs poorly and then continue to apply API using the more difficult problem distribution. This process of iteratively increasing n and then applying API continues until we either achieve a policy that performs well on the original problem distribution or no more improvement is observed. For more details and empirical results please see (Fern, Yoon, & Givan 2004).

Acknowledgments

This work was supported in part by NSF grants 9977981-IIS and 0093100-IIS.

References

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

Fern, A.; Yoon, S.; and Givan, R. 2003. Approximate policy iteration with a policy language bias. In *NIPS*.

Fern, A.; Yoon, S.; and Givan, R. 2004. Learning domain-specific control knowledge from random walks. In *ICAPS*.

Yoon, S.; Fern, A.; and Givan, R. 2002. Inductive policy selection for first-order MDPs. In *UAI*.