Regressing Deterministic Plans for MDP Function Approximation

Andrey Kolobov   Mausam   Daniel S. Weld
{akolobov, mausam, weld}@cs.washington.edu
Dept of Computer Science and Engineering
University of Washington, Seattle
WA-98195

Abstract

Most dynamic programming methods for solving MDPs store the value function explicitly as a table of state-value pairs. However, since the size of the state space is exponential in the number of state variables, these methods often run out of memory even on medium-sized problems. In response, researchers have suggested two ways to learn a compact *approximation* of the value function, either by combining a set of basis functions or by abstraction. While these methods have proved successful in continuous domains and in a few logically-specified domains with regular structure, noone has successfully applied them to probabilistic IPC domains.

This paper fuses these two paradigms into a novel value-function approximation scheme, which enables the solution of large problems from the Competition. Our method, RETRASE, has three steps. First, we use a fast classical planner, such as LPG-d, to generate a set of diverse plans for a determinized version of the probabilistic domain. Next, we regress these plans to generate a small set of basis functions, which induce overlapping neighborhoods over the state space. Third, we use a modified RTDP procedure to learn weights for each basis function (not for the whole space of reachable states). Finally, an agent can execute in the domain by using applicable basis functions to estimate the value of destination states and choose a good policy. Preliminary experiments show that RETRASE performs well on hard problems that challenge other planners.

INTRODUCTION

Markov Decision Processes (MDPs) are a popular framework for formulating probabilistic planning problems, which are applicable to a variety of interesting domains, ranging from military-operations planning to controlling a Mars rover (Aberdeen, Thiebaux, & Zhang 2004; Mausam *et al.* 2005). One of the most popular family of algorithms for solving MDPs is based on dynamic programming: value iteration, RTDP, and related approaches (Bellman 1957; Barto, Bradtke, & Singh 1995). Unfortunately, all of these algorithms suffer from the same critical drawback — they manipulate (reachable) states explicitly, thus requiring memory (and time) exponential in the number of domain features. The explicit table of values required to represent the value function grows too quickly. As a result, these approaches do not scale to handle practical problems, which are often too large. Indeed, value iteration and RTDP may often exhaust memory within a few minutes when applied to large problems from the probabilistic planning competition.

Two general approaches have been proposed to circumvent the memory requirements of algorithms that tabulate state values: *abstraction*, associating values with sets of states equivalent in some way, and *approximation*, eschewing the exact function for a compactly-representable alternative (e.g., a linear combination of basis functions) (Boutilier, Dean, & Hanks 1999). These approaches have proved quite successful when applied to MDPs whose state variables are *ordinal quantities* (*e.g.*continuous) (Guestrin *et al.* 2003b; Gordon 1995). Nonetheless, they are sometimes difficult to apply in what we call *nominal* domains, such as the PPDDL domains used in IPC.[1]

Abstraction is an elegant and powerful technique whose applicability depends on the presence of a meaningful distance/similarity measure over the state space. Sets of states that are similar according to this measure are assumed to have the same value. In *grid worlds* such as "Racetrack" and "WetFloor" (Barto, Bradtke, & Singh 1995; Bonet & Geffner 2006), which are common testbeds for the reinforcement-learning community, Euclidean distance is applicable, since states which have similar $\langle x, y \rangle$ coordinates tend to have similar values. However, few of the established logical (nominal) testbeds admit even an approximate distance function, which should obey symmetry and the triangle equality. Any domain that contains irreversible actions doesn't afford a distance function, and irreversible actions are common. E.g., blocks cannot be recreated in "Exploding Blocksworld," a flat cannot be repaired without a spare in "Tireworld", and in many domains, irreplaceable resources may be consumed. As we explain later, an important contribution of this paper is a novel state similarity measure that, while not a metric, has a very intuitive meaning.

Approximating the value function with a linear combination of basis functions is another approach which has had some success in a few nominal domains. For example, Guestrin has used this method in the "SysAdmin" and

[1] Algebraic-decision-diagram-based approaches, such as Symbolic LAO* and SPUDD are notable exceptions (Feng & Hansen 2002; Hoey *et al.* 1999), but even these novel methods exhaust memory for moderate sized problems.

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

“FreeCraft” domains (Guestrin *et al.* 2003b; 2003a). But both of these domains have numeric features which correlate naturally with the value of the state: e.g., the number of crashed processors, the amount of gold, *etc.* To our knowledge, noone has discovered comparable decompositions for the Competition domains.

Regressing Deterministic Plans. In this paper, we propose an algorithm RETRASE, which stands for **Re**gressing **T**rajectories for **A**pproximate **S**tate **E**valuation. RETRASE learns a compact value function approximation, which appears to work well in a range of nominal domains. Our approach combines abstraction and approximation ideas, but requires neither a distance measure nor a linearly decomposable value function.

The key insight behind RETRASE is a novel measure of similarity between states, defined *relative to a path to the goal*. Intuitively, let P be a plan to reach the goal. We consider two states to be similar if one may use P to reach the goal from either of those states. We define properties which characterize this similarity by regressing the goal through a suffix of the plan to generate a set of logical formulas, and we use each of these formulas to specify a *basis function*. During our planning process, RETRASE uses a modified version of RTDP (Barto, Bradtke, & Singh 1995) to assign weights to each of these basis functions, instead of to individual states as is traditional. During execution, the policy is defined from Q-values, where the value of a state is calculated from all matching basis functions; we considered a variety of combination methods, concluding that `min` captures the notion of choosing the best path toward a goal.

So far our discussion has assumed the existence of paths to the goal, but of course this begs the question to some degree. The final insight underlying RETRASE is the generation of a wide variety of *diverse* paths by exploiting recent advances in *deterministic* planning. Specifically, we determinize the probabilistic domain and use existing techniques, e.g. *LPG*, to generate multiple, qualitatively different, deterministic plans to the goal (Srivastava *et al.* 2007). While no deterministic plan may be guaranteed to reach the goal, every successful probabilistic trajectory must correspond to some deterministic plan. After generating a comprehensive set of these plans, RETRASE regresses each to create the compact set of value functions. This set is typically much smaller than the set of reachable states, thus giving our planner a big reduction in memory requirements as well as in the number of parameters to be learned.

We demonstrate the practicality of our framework by comparing it to the top performing planners of the IPC-5 probabilistic track on some of the hardest problems from that competition as well as some other challenging problems. RETRASE demonstrates orders of magnitude better scalability than one of the best optimal planners, and finds significantly better policies than the state-of-the-art approximate planners.

BACKGROUND

In this paper, we restrict ourselves to indefinite-horizon MDPs. An indefinite-horizon MDP is defined to be a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, where

- $\mathcal{S}$ is a finite set of states,
- $\mathcal{A}$ is a finite set of actions,
- $\mathcal{T}$ is a transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ that describes the probability of transitioning from state S_i to S_j by executing action A,
- $\mathcal{C}$ is a map $\mathcal{C} : \mathcal{A} \to \mathbb{R}^+$ that specifies a cost for every action,
- $\mathcal{G}$ is a set of a goal states,
- s_0 is the start state.

Solving an MDP means finding a policy $\pi : \mathcal{S} \to \mathcal{A}$ that specifies for every state an action the agent should take to eventually reach the goal. We are interested in computing an optimal policy, i.e. one that incurs the minimum total action cost to reach the goal. *Indefinite horizon* refers to the fact the the total action cost is accumulated over a finite-length action sequence whose length is unknown. The expected cost of reaching the goal from a state s under policy π is described by the *value function*

$$V^{\pi}(s) = \mathcal{C}(s, \pi(s)) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s') V^{\pi}(s') \qquad (1)$$

An optimal function satisfies the following conditions, called *Bellman equations*:

$$\begin{aligned} V^*(s) &= 0 \text{ if } s \in \mathcal{G}, \text{otherwise} \qquad (2) \\ V^*(s) &= \min_{a \in \mathcal{A}} [\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s') V^*(s')] \end{aligned}$$

Given an optimal value function, an optimal policy can be computed as follows:

$$\pi^*(s) = \arg\min_{a \in \mathcal{A}} [\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, \pi(s), s') V^*(s')] \qquad (3)$$

Equations (2) and (3) suggest a dynamic programming-based way of finding an optimal policy, first described by Bellman (1957). It involves initializing state values using a heuristic and iteratively updating the values of all states using equations (2) in an operation called *Bellman backup* until the values converge (or change by only a very small amount between successive iterations). The policy is read off the value function via equation (3).

This algorithm, called *value iteration* (VI), has given rise to many improvements. One of them, RTDP (Barto, Bradtke, & Singh 1995), does value function initialization like VI and tries to reach the goal multiple times by using the policy derived from the current value function. During each trial, it updates the value function over the states in the path using Bellman backups. A popular variant, LRTDP, adds a terminating condition to RTDP by labeling those states whose values have converged as ‘solved’ (Bonet & Geffner 2003).

Note that to compute the policy, both VI and RTDP have to store values for many states, the number of which is exponential in the number of domain features. RTDP saves some

space by storing values only for the states reachable from the initial one. However, the reduction is typically limited to a constant factor for many domains.

Determinization of Probabilistic Domains Recently, there has been significant progress in exploiting the fast deterministic planners to approximate policy construction for MDPs. This approach is pioneered by FFReplan (Yoon, Fern, & Givan 2007). We will use the key insight of *determinization* from this work. Determinization refers to the conversion of the probabilistic planning domain into a deterministic domain by treating each outcome of a probabilistic action as an independently controllable deterministic action with the same preconditions and effects. Thus, the lack of a deterministic plan to the goal makes the state a dead end in the original planning problem. However, existence of a deterministic plan only guarantees a non-zero probability trajectory starting from the state.

Diverse Deterministic Plans Our work crucially builds on yet another previous work. We use a package called *LPG*-d (Srivastava *et al.* 2007) that returns a set of diverse deterministic plans for a domain. Based on the LPG planner (Gerevini, Saetti, & Serina 2003), it finds a desired number of plans for the given problem while making sure that these plans differ from each other in the number and sequence of actions they involve. The desired diversity of the plan set is specified by the d-parameter, its values ranging from 0 to 1.

ReTrASE DETAILS

On a high level, RETRASE works as follows. First, it computes certain "good" properties of states. Possessing a given such property p guarantees the existence of certain trajectories from the state to the goal. Thus, possessing property p naturally defines a set of p-*similar* states. The properties differ in the expected cost of the trajectories to the goal that they enable, as well as in the total probability of these trajectories. RETRASE takes this difference into account by learning a weight for each computed property. With property weights learned, the value of a state is, to a first approximation, the minimum of its properties' weights. A key advantage of the algorithm is that in practice the number of properties RETRASE needs in order to compute a good approximation to the value function is far smaller than the number of states VI/RTDP need to store values for.

Definitions. We define a state *property* to be a conjunction of literals. We say that a state s *possesses* property p if p holds in s. With each property p, we associate a *basis function* that has value 1 in s iff s possesses p. We call a set of states p-*similar* if all states in the set possess property p.

We say that property p *enables* a set of trajectories T to the goal if the goal can be reached from any state possessing p by following any of the trajectories in T[2]. We call a state that doesn't possess any properties from the set we are interested in a *mysterious* state. A *dead-end* is a state with no trajectory to the goal. An *implicit dead-end* is a state that has no trajectory to the goal but has at least one applicable action. An *explicit dead-end* is a state with no applicable actions.

[2]assuming that the desired outcome is obtained for each action on the trajectory.

Algorithm intuition. Consider a trajectory of actions and outcomes, $A = a_1, o_1, a_2, o_2, \ldots, a_n, o_n$. Moreover, suppose that a goal is reached when A is executed. This is an indication that A is causally an important sequence of actions. To discover the causal properties $p_1, \ldots, p_n$ that allow A's successful execution, we simply regress from the goal state. We can now claim that action sequence $a_j, \ldots, a_n$ executed starting from *any* state possessing property p_j will lead us to the goal with positive probability. Note that A chooses specific outcomes per action and thus the actual real-world execution may not always reach the goal. Nevertheless, all properties that enable any positive-probability trajectory to the goal are important for our purposes, as they act as a basis for further planning. In essence, this step can be thought of as unearthing the relevant causal structure necessary for the planning task at hand. If we start with diverse candidate trajectories that take us to goal we can hope to compute several causal properties of the domain useful for our task.

We can now define a new probabilistic planning problem over a state space comprising of these properties. Hopefully, the space of properties will be much smaller than the original state space, since only the relevant causal structure will be retained[3]. There are many imaginable ways to learn the *weights* for the basis functions. In this paper, we explore one of them — a modified version of LRTDP.

Importantly, each property defines a set of states with similar expected behavior, based on the action sequence it enables. A set of properties will define a set of sets of p-similar states (not all disjoint).

Note, however, that the properties differ in the total expected cost of trajectories they enable as well as in the total probability of these trajectories. This happens partly because each trajectory considers only one effect for each of its actions. The sequence of effects the given trajectory considers may be quite unlikely. In fact, getting some action outcomes that the trajectory doesn't consider may prevent the agent from ever getting to the goal. Thus, it may be much "easier" to reach the goal from some p-similar sets than others. Property weights reflect these differences. Now, given that each state is generally in several p-similar sets, what is the connection between the state's value and the weights of the p-similar sets of which the state is a member? Intuitively, the state's value shouldn't be higher than the lowest w_p of any of its p-similar sets. Since several properties can enable the same plan, it is hard to determine the true value of a state even while knowing the property weights. However, we can *approximate* the state value by the smallest weight of any property the state possesses. This amounts to saying that the "better" a state's "best"' property is, the "better" is the state itself.

Thus, deriving useful state properties and their weights gives us an approximation to the optimal value function.

[3]We may approximate this further by putting a bound on the number of properties we are willing to handle in this step.

Algorithm's operation. ReTrASE, whose pseudo code is presented in Algorithm 1, starts by computing the determinization D' of the domain. We need D' to rapidly compute many trajectories from the start state to the goal in the original domain D. This is exactly what the algorithm does next. It uses *LPG-d* to find a desired number of plans in D'. The plans differ in the number of actions they use as well as in their sequence. Finding all ways of reaching the goal from the initial state in D' would give us all properties we are interested in. However, this approach is impractical, because in some domains the number of plans may be exponential in the number of states. The best we can hope for is a large number of fairly diverse plans.

To extract properties from a plan in D', we simply regress through it (subroutine $Regress(p_{D'})$ in the pseudo code). Regression yields not only the basis functions but also the cost of reaching the goal in D' from any state with the given basis function via the given plan. We use these values to compute a powerful (though inadmissible) heuristic, described in the next subsection. The heuristic is computed by method $GetBasisFuncsAndHeuristic(P_{D'})$ in Algorithm 1.

After obtaining the basis functions and heuristic values for their weights, ReTrASE runs a modified version of RTDP to learn their actual values. For each state s_j the modified RTDP visits, $ModifiedBellmanBackup(.)$ routine updates the smallest weight of any basis function in s_j, i.e. the weight of the basis function that currently determines s_j's value. Ideally, we would like to update the weight of the basis function(s) that enabled the action that brought the agent from s_j to the next state in the current RTDP trial, s_i. However, there may be no such basis function (e.g., in case s_j is an implicit dead-end), or this basis function may be expensive to determine. Nonetheless, the operation in $ModifiedBellmanBackup(.)$ serves as a reasonable approximation.

Heuristic. For every basis function F and a plan, the total cost of actions of the plan we've regressed to obtain F is an estimate of the "distance" from F to the goal, i.e. an estimate of F's weight. If we knew all deterministic plans whose regression yields F, we could construct a very good admissible heuristic H by taking the minimum of distances from F to the goal over all these plans. In practice, the minimum of distances from F to the goal over all plans *that we computed with LPG-d* gives a natural approximation H' to H. H' is inadmissible, since deterministic plans we haven't considered may see F closer to the goal than our estimate. However, it is easy to see that the more plans we choose to consider, the closer H' values get to those of H.

We use H' in our algorithm by initializing basis function weights with values computed in $GetBasisFuncsAndHeuristic(P_{D'})$.

Dead-ends and mysterious states. So far, we have implicitly assumed all states to possess at least one of the discovered properties. Mysterious states are states without any of the properties we are interested in, so this assumption does not hold for them. Mysterious states can be states that lie on

---

Algorithm 1 ReTrASE

```
1: Input: probabilistic domain D, problem P, #RTDP trials N_R, trial length L, #deterministic plans N_{D'}, plan difference δ
2: declare M, a map from basis functions to weights
3: compute D', the determinization of D
4: compute a set of deterministic plans P_{D'} ← LPG-d(D', P, N_{D'}, δ)
5: // Populate M with basis function - heur. weight pairs
6: GetBasisFuncsAndHeuristic(P_{D'})
7: // Do modified RTDP over the basis functions
8: for all i = 1 : N_R do
9:     declare current state s ← s_0
10:    declare numSteps ← 0
11:    while numSteps < L do
12:        choose action a with minimum expected value
13:        ModifiedBellmanBackup(a, s)
14:        execute and randomly sample next state
15:        numSteps ← numSteps + 1
16:    end while
17: end for
18:
19: GetBasisFuncsAndHeuristic(P_{D'})
20: for all deterministic plans p_{D'} ∈ P_{D'} do
21:    declare S_FC, set of basis function-cost pairs ⟨F_p, cost(F_p)⟩
22:    Regress(p_{D'})
23:    for all ⟨F_i, cost(F_i)⟩ ∈ S_FC do
24:        if M[F_i] == NULL then
25:            M[F_i] ← cost(F_i)
26:        else
27:            M[F_i] ← min[M[F_i], cost(F_i)]
28:        end if
29:    end for
30: end for
31:
32: Regress(p_{D'})
33: declare F ← Goal
34: declare cost ← 0
35: for all i = length(p_{D'}) : 1 do
36:    action ← p_{D'}[i]
37:    cost ← cost + cost(action)
38:    F ← (F ∪ precond(action)) − effect(action)
39:    S_FC ← S_FC ∪ {F, cost}
40: end for
41:
42: ModifiedBellmanBackup(a, s)
43: find basis function F with smallest weight M[F] that holds in s
44: M[F] ← Cost(a) + ExpectedValue(a, s)
```

some trajectory to the goal that we haven't discovered. They can also be implicit or explicit dead-ends – neither type can be on any trajectory to the goal by definition. In either case, recognizing them and learning their values may be crucial for learning a good policy approximation. To be able to do that, whenever during the RTDP trials we encounter a mysterious state (which may also be a dead-end), we create a special basis function equal to the state at hand. Clearly, this basis function only holds in this state; therefore, creating too many such basis functions may potentially increase ReTrASE 's memory requirements to those of ordinary RTDP. Fortunately, our experiments have shown that in practice, we have to create relatively few of these basis functions. We discuss this issue more in the Future Work section.

We also tried another method of dealing with mysterious states, by assuming *all* of them to be dead ends. Theoretically, this makes the algorithm very sensitive to the basis functions used in a given run, as they carve out the valid part of the state space. Therefore, the assumption should be accurate if we use nearly all plans to the goal; as the fraction of plans that we use for discovering basis functions diminishes, the performance of the algorithm should drop. Experimentally, we found the behavior to be as described.

EXPERIMENTAL RESULTS

Our goal in this section is to demonstrate two important properties of ReTrASE – (1) scalability on large domains, and (2) quality of solutions in complex domains. We start by showing that ReTrASE easily scales to problems on which the state-of-the-art optimal planners run out of memory. Then, we illustrate ReTrASE's ability to compute better policies for very hard problems than state-of-the-art approximate planners.

We report results on two domains — Triangle-Tire, and Drive. Triangle-Tire is a modified version of the TireWorld domains from the IPC5 due to (Little & Thiebaux 2007). The task of the domain is to reach a destination while navigating through a network of roads, in our case, of the form of a triangle. The car may get a flat tire with some probability, but if it has a spare tire then it can still reach the goal. We test on 6 problems, reported by Little and Thiebaux, with increasing complexity. The size of the problem state space and the difficulty of finding the optimal solution increase exponentially in the ordinal of the problem.

Our second domain is the Drive domain[4] from IPC-5. Here, the task is to drive along a network of crossroads and traffic crossings. We experiment on two of the hardest problems (p13 and p15) used in the competition for this domain.

We run several planners on these domains, including FPG (Buffet & Aberdeen 2006) – the competition winner of IPC5, FFReplan (Yoon, Fern, & Givan 2007) – the competition winner of IPC4, LRTDP run with the inadmissible FF heuristic (LRTDP$_{FF}$), and LRTDP$_{opt}$ – LRTDP with Atom-Min-1-Forward|Min-Min heuristic (Bonet & Geffner 2005). The last of these is one of the best known *optimal* algorithms, whereas the rest are the state-of-the-art approximate solvers.

Our algorithm, ReTrASE, takes two parameters – the number of desired diverse deterministic plans, and the total number of trials for RTDP. For now, we focus on providing the proof of concept for our system, and hence we have set these parameters by manual inspection. We plan to estimate these parameters automatically by analyzing the structure of the problem and a better convergence test. We revisit this issue in the next section.

Comparing Scalability. We begin by demonstrating the memory savings and the running time improvements of ReTrASE over LRTDP$_{opt}$. Figure 1 and 2 shows the savings of ReTrASE to increase dramatically with problem size. For example, for the hardest triangle-tire problem we outperform the optimal solver in the memory usage by an order of magnitude. Moreover, even though ReTrASE has an unoptimized implementation, it still outperforms the optimal algorithm in the total planning time. The performance of ReTrASE is also significantly better than LRTDP$_{FF}$.

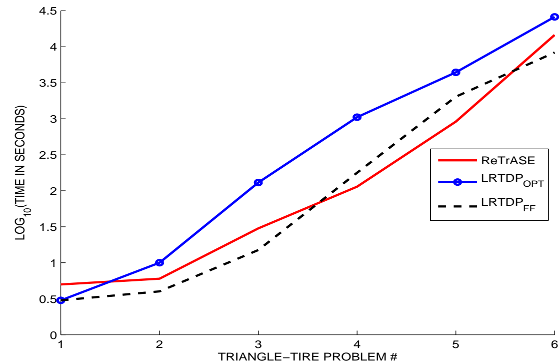

Figure 1: Speed of ReTrASE and other VI-based planners. Even our unoptimized implementation is significantly faster than the optimal algorithm and quite comparable to LRTDP$_{FF}$.

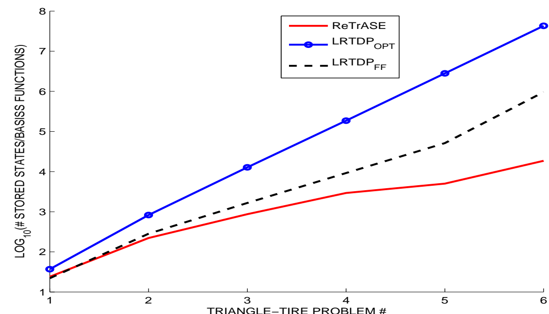

Figure 2: Memory usage of ReTrASE and other VI-based planners. ReTrASE is dramatically better than LRTDP$_{opt}$ and somewhat better than LRTDP$_{FF}$.

Of course, other approximate algorithms don't suffer from the scalability issues so much. For instance, FPG learns policy directly and represents it with an appropriately trained neural net. FFReplan is essentially a deterministic planner that tries to act according to a deterministic plan and replans when something goes wrong. Thus, it is more meaningful to compare ReTrASE against these on the

[4] We discovered a bug in the original PPDDL file, and hence, report on a slightly modified version of the domain.

quality of solutions produced.

Comparing Solution Quality. Figure 3 describes the coverage (percentage of simulation runs reaching the goal) comparisons for the Triangle-Tire domain, whereas Table 1 reports the same for the Drive domain. We observe that RETRASE outperforms both FFReplan and FPG by significant margins. The performance of FPG and FFReplan drops sharply on hard problems, whereas RETRASE is able to learn near-optimal policies for all of them.

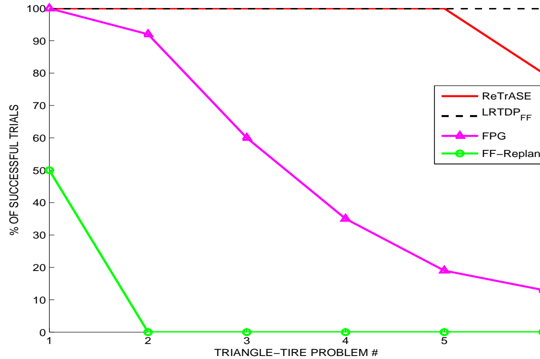

Figure 3: Success rates on Triangle-Tire domain. RETRASE outperforms both FFReplan and FPG by wide margins. It performs almost as well as LRTDP$_{FF}$.

Problem	RETRASE	LRTDP$_{opt}$	LRTDP$_{FF}$	FPG
p13	37%	37%	37%	–
p15	67%	67%	67%	30%

Table 1: Success rates on Drive domain. RETRASE produces optimal solutions for both domains. FPG doesn't converge on one problem, and produces lower quality solution on the other.

Due to its learning strategy, FFReplan is not well-suited for the Tireworld domain. FFReplan doesn't plan for contingencies; as a result, it doesn't make an effort to go through locations that have a spare tire. When a tire bursts, FFReplan finds itself in a dead-end. This is indeed what happens in practice. Consider problem p14 of Tireworld, one of the hardest problems from this domain. IPC-5 report shows that FFReplan's policy is only successful in about 40% of runs. FPG, incidentally, has a 70%-success. However, it is possible to learn a better policy. RETRASE does exactly that, succeeding in 100% of trials.

Moreover, RETRASE is at-par with LRTDP$_{FF}$ on coverage. However, as illustrated in Figure 4, RETRASE results with better expected costs to reach the goal compared to LRTDP$_{FF}$. So, overall RETRASE is enormously better than the optimal algorithm on scalability, and yields a much higher quality solution than the other state-of-the-art systems.

We particularly experimented on the variants of the TiresWorld domain, since it is considered very difficult for the approximate techniques. RETRASE's excellent performance on the hard problems in this domain is highly encouraging. We plan to further test our algorithm on the other difficult problems of the planning competition.

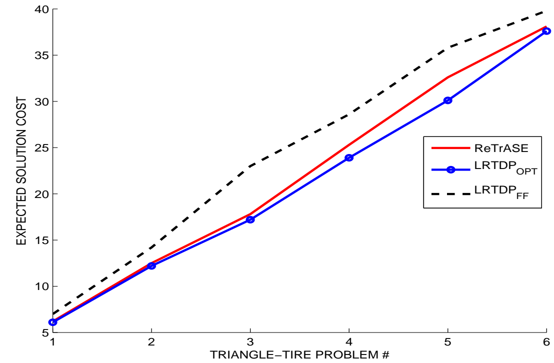

Figure 4: Plan quality. While all algorithms have 100% coverage for almost all problems RETRASE returns near-optimal solutions whereas LRTDP$_{FF}$ produces worse solutions.

Control Experiment: Mysterious States. For the Drive domain, one that has several dead-end states, we ran a control experiment to test two approaches to handle the mysterious states. Recall that mysterious states are ones that do not possess any property from our set of properties. In other words no deterministic plans found in the first step can be executed from these states. This can happen due to two reasons — the states are dead ends, or the deterministic plans found were not diverse enough. Explicit dead ends are easy to detect, but implicit ones are not.

In the first of the approaches, we treated all mysterious states as dead ends. In the second, we created a new basis function for each mysterious state encountered (except when it was an explicit dead end). This way, we ended up learning values for each of the mysterious states individually. We observed that the first solution is not a robust approach, since it branded many legitimate states as dead ends, and hence, the coverage of the resulting policy was low. The second technique, on the other hand, worked well. However, it ended up exploring a large number of states, since it had to explicitly store each dead end, which can be large in number. For the results above we have used the second technique. However, we discuss other approaches to handle mysterious states in the next section.

DISCUSSION

As evidenced by the experimental results, our algorithm performs on the par or better with the state-of-the-art systems, while requiring considerably less memory. However, there are a number of areas for improvement, which we discuss below.

Mysterious states and dead ends One limitation of our current implementation is its lack of skill in dealing with mysterious states. Our experiments with the Drive domain illustrate that mysterious states may have paths to goal not found earlier, and so we cannot presuppose that all mysterious states will be implicit dead ends.

Our current solution is robust to quality of diverse plans, since it explicitly stores all mysterious states, including im-

plicit dead ends found so far. However, it is potentially expensive, since it might end up enumerating all dead ends, which can be huge in number. In the future we plan to explore several ways to deal with mysterious states effectively. For instance, one approach can be to use causal regression to infer properties that will prove that a state is a dead end. Thus we will be able to compactly characterize the mysterious states which will reduce much of our computation and memory requirements.

Improving Property Construction The quality of our state value approximation is highly dependent on the quality of our basis function set. An ideal set of properties will – (1) be causally diverse, and (2) include all high probability, low cost trajectories. At the moment, LPG-d is not looking for either of the two, it is just looking for diversifying the set of actions that participate in the plans. We plan to test our implementation with the version of LPG-d reported in (Srivastava *et al.* 2007) that specifically tries to diversify the causal structure of the returned plans. Moreover, we hope to modify the algorithm that will include, with each determinized action, a pseudo cost (reflecting the probability of outcome, and cost of the original action), so that the returned plans have higher probabilities of success and low total costs.

Updating basis function weights As previously mentioned, our modified Bellman backup isn't always updating the correct basis function. We need a more principled way of selecting the basis function to update during this operation. Ideally, we will store the mapping from a property to an action (duplicating the property if it appears in multiple plans) and update the basis function based on the greedy action at the current backup.

Convergence Test Running RETRASE currently involves specifying the desired number of RTDP trials. In the current experiments we stop the process by manual inspection when the basis function weights seem to have converged. An aspect that complicates an automatic convergence detection is that the basis function weights and state values don't change monotonically. In fact, in some cases these may oscillate a lot even when the fraction of trials that reach the goal stays nearly constant over many trials, indicating no benefit in additional learning. We are considering two solutions to convergence detection problem. One is to use the benefit of additional learning for convergence. Let $f(t)$ be the fraction of trials that reached the goal for a short window along the tth trial, the learning process should stop when $f'(t) \approx 0$. Our experiments have showed the graph of $f(t)$ to be not very smooth. To simplify matters, we could pick the stopping condition $|\frac{f(t)-f(t-c)}{c}| < \delta$, where c is an integer smoothing parameter. An alternative solution is to employ a learning rate α_t similar to reinforcement learning approaches like Q-learning. Using an appropriately decaying α, s.t. $\alpha_t \to 0$ as $t \to \infty$ we can guarantee convergence. More experiments are needed to test the quality of policies produced by this technique.

RELATED WORK

Several researchers have considered state abstraction for MDP planning. A popular approach is based on algebraic decision diagrams (ADDs). ADDs compactly represent a value function for a factored nominal domain. All steps of value iteration are compactly performed for the whole state space at once using operations on ADDs. Optimal algorithms such as SPUDD and Symbolic LAO* (Hoey *et al.* 1999; Feng & Hansen 2002) have been popular, however, the compactness achieved depends severely on the variable order in an ADD. Moreover, the algorithms scale well only to medium sized domains, after which the memory typically exhausts. APRICODD is an approximation technique using ADDs (St-Aubin, Hoey, & Boutilier 2000). While APRICODD has been shown successful on many problems, it is not clear whether it is competitive with today's top methods since it hasn't been applied to the competition domains.

Another technique to solve large MDPs involves a basis function decomposition for a domain, and uses a combination of basis function weights (*e.g.*, linear combination) to represent the value function. This kind of function approximation has been immensely popular (Gordon 1995; Koller & Parr 2000; Guestrin *et al.* 2003b). These techniques have two major drawbacks. First, finding a set of good basis functions is fairly tricky. Most often the researchers or domain experts encode them manually. Second, and more importantly, these algorithms are typically applied on domains that have ordinal state variables. Such variables offer a very different structure (easy to find distance metrics in the domains) that is well exploited by these methods. Unfortunately, nominal domains behave very differently, and our initial attempts to directly apply these techniques to nominal domains were not that successful.

A notable exception to above is FPG (Buffet & Aberdeen 2006), a rare example of a function approximation approach successful in nominal domains. It performs policy search and represents the policy compactly with a neural network. For each action, FPG learns the probability with which it should be executed in each state. Using gradient ascent on the probabilities and doing simulation to estimate the rewards, FPG converges on a local optimum. In the domains we tried RETRASE performed as well or better than FPG: it reached the goal a larger percentage of trials than FPG.

There is also a large body of work on learning policies for relational MDPs. (Gretton & Thiebaux 2004) do first-order regression on optimal plans in small problem instances to construct a policy for large problems in a given domain. The policy is learned from the obtained first-order formulas by inductive logic programming. (Sanner & Boutilier 2006) use first-order regression to obtain a set of basis functions. They aggregate them into a linear weighted combination by solving a linear program to compute their weights. (Wu & Givan 2007) analyze the Bellman residual of a value function approximation to iteratively compute a set of basis functions.

Our work uses the determinization of the domain similarly to (Gretton & Thiebaux 2004) and (Sanner & Boutilier 2006) — regressing deterministic plans to compute basis functions. However, our function aggregation and weight-learning methods are completely different from theirs. Our approach is also related in spirit to the probabilistic planners that use determinized domains for solving probabilistic

planning problems. The most popular of these is FFReplan, (Yoon, Fern, & Givan 2007), the competition winner in IPC-4. Other planners include Tempastic (Younes & Simmons 2004), precautionary planning (Foss, Onder, & Smith 2007), and hindsight determinization (Yoon *et al.* 2008).

CONCLUSION

The most popular algorithms for solving MDPs over nominal domains (e.g. those specified in PPDDL) have to tabulate the value function and consequently suffer from high memory requirements. To resolve the issue, a number of approaches have been proposed, including approximating the value function by state abstraction or as a combination of basis functions. Both methods have been applied rather successfully in domains with ordinal-valued variables. In nominal domains, however, state abstraction suffers from a lack of a state-similarity metric, caused by the presence of irreversible actions. This paper makes the following contributions:

- We present a domain-independent state-similarity measure, under which states are alike if they share a goal path.
- We define a set of domain-specific state properties, each enabling a trajectory from a state to the goal. These properties let us induce the above similarity measure over a nominal state space. We propose computing these properties in a domain-independent way by regressing through a set of plans in the determinized version of the domain.
- We show how combining the weights of the state properties helps approximate the state value. We modify the RTDP algorithm to learn the property weights.
- We empirically demonstrate that our planner learns better policies than state-of-the-art planners like FPG and FFReplan on the hardest problems from IPC-5 competition.

Despite the promise of our method, much remains to be done. Diversifying the deterministic plans that we use to extract state properties by different metrics could give us a more informative set of properties. Ability to better deal with states that have none of the properties we've discovered may help us further shrink the set of basis functions. Detecting convergence is crucial for increasing usability and reducing the running time of our algorithm.

Acknowledgements We'd like to thank the reviewers for insightful comments. This work is supported by ONR grant N00014-06-1-0147 and the WRF/TJ Cable Professorship.

References

Aberdeen, D.; Thiebaux, S.; and Zhang, L. 2004. Decision-theoretic military operations planning. In *ICAPS'04*.

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

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS'03*, 12–21.

Bonet, B., and Geffner, H. 2005. mGPT: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research* 24:933–944.

Bonet, B., and Geffner, H. 2006. Learning in depth-first search: A unified approach to heuristic search in deterministic non-deterministic settings, and its applications to MDPs. In *ICAPS*, 142–151.

Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research* 11:1–94.

Buffet, O., and Aberdeen, D. 2006. The factored policy gradient planner (ipc-06 version). In *Fifth International Planning Competition at ICAPS'06*.

Feng, Z., and Hansen, E. 2002. Symbolic heuristic search for factored Markov decision processes. In *AAAI'02*.

Foss, J.; Onder, N.; and Smith, D. 2007. Preventing unrecoverable failures through precautionary planning. In *ICAPS'07 Workshop on Moving Planning and Scheduling Systems into the Real World*.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research* 20:239–290.

Gordon, G. 1995. Stable function approximation in dynamic programming. In *ICML*, 261–268.

Gretton, C., and Thiebaux, S. 2004. Exploiting first-order regression in inductive policy selection. In *UAI'04*.

Guestrin, C.; Koller, D.; Gearhart, C.; and Kanodia, N. 2003a. Generalizing plans to new environments in relational MDPs. In *IJCAI'03*.

Guestrin, C.; Koller, D.; Parr, R.; and Venkataraman, S. 2003b. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research* 19:399–468.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. Spudd: Stochastic planning using decision diagrams. In *UAI'99*, 279–288.

Koller, D., and Parr, R. 2000. Policy iteration for factored MDPs. In *UAI'00*, 326–334.

Little, I., and Thiebaux, S. 2007. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*.

Mausam; Benazara, E.; Brafman, R.; Meuleau, N.; and Hansen, E. 2005. Planning with continuous resources in stochastic domains. In *IJCAI'05*, 1244.

Sanner, S., and Boutilier, C. 2006. Practical linear value-approximation techniques for first-order mdps. In *UAI'06*.

Srivastava, B.; Kambhampati, S.; Nguyen, T.; Do, M.; Gerevini, A.; and Serina, I. 2007. Domain independent approaches for finding diverse plans. In *IJCAI'07*.

St-Aubin, R.; Hoey, J.; and Boutilier, C. 2000. APRICODD: Approximate policy construction using decision diagrams. In *NIPS'00*.

Wu, J., and Givan, B. 2007. Discovering relational domain features for probabilistic planning. In *ICAPS'07*.

Yoon, S.; Fern, A.; Kambhampati, S.; and Givan, R. 2008. Probabilistic planning via determinization in hindsight. In *AAAI'08*.

Yoon, S.; Fern, A.; and Givan, R. 2007. FF-Replan: A baseline for probabilistic planning. In *ICAPS'07*, 352–359.

Younes, H. L. S., and Simmons, R. G. 2004. Policy generation for continuous-time stochastic domains with concurrency. In *ICAPS'04*, 325.