ReTrASE: Integrating Paradigms for Approximate Probabilistic Planning

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

Past approaches for solving MDPs have several weaknesses: 1) Decision-theoretic computation over the state space can yield optimal results but scales poorly. 2) Value-function approximation typically requires human-specified basis functions and has not been shown successful on nominal ("discrete") domains such as those in the ICAPS planning competitions. 3) Replanning by applying a classical planner to a determinized domain model can generate approximate policies for very large problems but has trouble handling probabilistic subtlety [Little and Thiebaux, 2007].

This paper presents RETRASE, a novel MDP solver, which combines decision theory, function approximation and classical planning in a new way. RETRASE uses classical planning to create basis functions for value-function approximation and applies expected-utility analysis to this compact space. Our algorithm is memory-efficient and fast (due to its compact, approximate representation), returns high-quality solutions (due to the decision-theoretic framework) and does not require additional knowledge from domain engineers (since we apply classical planning to automatically construct the basis functions). Experiments demonstrate that RETRASE outperforms winners from the past three probabilistic-planning competitions on many hard problems.

1 INTRODUCTION

Markov Decision Processes (MDPs) are a popular framework for formulating probabilistic planning problems, applicable to a variety of interesting domains ranging from military-operations planning to controlling a Mars rover [Aberdeen *et al.*, 2004; Mausam *et al.*, 2005]. One of the most popular families of algorithms for solving MDPs that yields high-quality solutions is based on dynamic programming: value iteration (VI), RTDP, and related approaches [Bellman, 1957; Barto *et al.*, 1995]. Unfortunately, all of these decision-theoretic algorithms suffer from the same critical drawback — they represent the value function extensionally, *i.e.*, as a table, thus requiring memory (and time) exponential in the number of domain features. Since this extensional representation grows too rapidly, these approaches do not scale to handle real-world problems. Indeed, VI and RTDP typically exhaust memory, when applied to large problems from the ICAPS international probabilistic planning competition (IPPC).

Two broad approaches have been proposed for avoiding creation of a state/value table. One method involves *domain determinization* and uses a classical planner as a subroutine in computing a policy. Since determinization planners, e.g., FFReplan [Yoon *et al.*, 2007], tend to disregard the probabilistic nature of actions, they often have trouble with *probabilistically interesting* [Little and Thiebaux, 2007] domains, in which short plans have a low probability mass.

The other method, *dimensionality reduction*, maps the state space to a parameter space of lower dimension. Typically, the mapping is done by constructing a small set of basis functions, learning weights for them, and combining the weighted basis function values into the values of states. Researchers have successfully applied dimensionality reduction to domains after manually defining a domain-specific mapping. It is relatively easy to find such a mapping in domains with *ordinal* (*e.g.*, numeric) state variables, especially when the numeric features correlate strongly with the value of the state, *e.g.*, gridworlds, "SysAdmin" and "FreeCraft" [Guestrin *et al.*, 2003a; 2003b; Gordon, 1995]. In contrast, dimensionality reduction is difficult to use in *nominal* (*e.g.*, "discrete" or "logical") domains, such as those used in the IPPC. Besides not having metric quantities, there is often no valid distance function between states (indeed, the distance between states is usually asymmetric and violates the triangle equality). It is extremely hard for a human to devise basis functions or a reduction mapping, let alone write an automatic procedure to come up with basis functions in nominal domains.

To our knowledge, there has been little work on mating decision theory, determinization, and dimensionality reduction. This paper bridges the gap — proposing a fusion of these ideas that removes the drawbacks of each. Our algorithm RETRASE, which stands for **Re**gressing **Tr**ajectories for **A**pproximate **S**tate **E**valuation, learns a compact value function approximation successful in a range of nominal domains. It obtains a set of basis functions automatically by planning in a determinized version of the domain at hand,

learns the weights for these basis functions by the decision-theoretic means, and aggregates them to compute state values. The set of basis functions is normally much smaller than the set of reachable states, thus giving our planner a large 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 IPPC-04, 06 and 08 performers and other state-of-the-art planners, on challenging problems from these competitions. RETRASE demonstrates orders of magnitude better scalability than the best optimal planners, and frequently finds significantly better policies than the state-of-the-art approximate solvers.

2 BACKGROUND

In this paper, we restrict ourselves to goal-oriented factored MDPs[1], defined as tuples of the form: $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, where

- $\mathcal{S}$ is a finite set of states,
- $\mathcal{A}$ is a finite set of actions,
- $\mathcal{T}$ is a transition function $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ that gives the probability of moving from s_i to s_j by executing a,
- $\mathcal{C}$ is a map $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^+$ that specifies action costs,
- s_0 is the start state,
- $\mathcal{G}$ is a set of (absorbing) goal states.

Factored MDPs that model planning problems represent states as conjunctions of logical variables, and each action has a precondition, also expressed as a variable conjunction. An action is applicable in a state if and only if the action's precondition subsumes the state. Solving an MDP means finding a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that specifies the actions the agent should take to eventually reach a goal. We are interested in computing an optimal or near-optimal policy, i.e., one that incurs the minimum expected total action cost to reach a goal, which is an absorbing state. The expected cost of reaching a 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*:

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

Given $V^*(s)$, an optimal policy may be computed as follows:

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

[1]This formulation is strictly more general than infinite-horizon MDPs with discounted rewards [Bertsekas and J.Tsitsiklis, 1996]

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

2.1 Real-Time Dynamic Programming

This algorithm, called *value iteration* (VI), has given rise to many improvements. One of them, called "Real-Time Dynamic Programming" (RTDP) [Barto *et al.*, 1995], initializes the value function using a heuristic and tries to reach the goal multiple times by using the policy derived from the current value function. During each so-called '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 and Geffner, 2003]. LRTDP also requires the heuristic to be admissible.

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 space by storing values only for states which are reachable from s_0; however, this savings only enables RTDP to solve modestly larger problems.

2.2 Determinization of Probabilistic Domains

Recently, there has been significant progress, e.g., FFReplan [Yoon *et al.*, 2007] and FFHop [Yoon *et al.*, 2008], in applying fast deterministic planners to quickly construct (suboptimal) MDP policies. One idea stemming from this work is *determinization* — converting a probabilistic planning domain, D, into classical, deterministic form, D_d, by treating uncertain outcomes of a probabilistic action as independently-controllable, deterministic actions with the same preconditions. Existence of a deterministic plan guarantees that there is a sequence of actions that, when applied from s, has a non-zero probability of reaching the goal.

3 ReTrASE

On a high level, RETRASE explores the state space in the same manner as RTDP, but, instead of performing Bellman backups on states themselves, backups are performed over *properties* of the visited states. For each property, modified RTDP learns a weight that reflects the quality of the plans enabled by that property. A state's value may then be computed by aggregating the weights of all its properties. Conceptually, there are three kinds of states at runtime: ones that have been deemed dead ends, ones for which some properties are known, and ones not yet assigned to the other two categories. When RETRASE encounters a state s of the third type, it applies a classical planner (e.g., FF [Hoffman and Nebel, 2001]) to a determinized version of the domain starting from s. If no classical plan exists, then every probabilistic policy from s has zero probability of reaching the goal, and s is marked as a dead end. If FF finds a plan, however, RETRASE regresses the goal

```
(define (domain GremlinWorld)
  (:types tool)
  (:predicates (has ?t - tool)
               (gremlin-alive)
               (plane-broken))
  (:constants Wrench - tool
              Screwdriver - tool
              Hammer - tool)

  (:action pick-up
   :parameters (?t - tool)
   :precondition (and (not (has ?t)))
   :effect (and (has ?t)))

  (:action tweak
   :parameters ()
   :precondition (and (has Screwdriver)
                      (has Wrench))
   :effect (and (plane-broken)))

  (:action smack
   :parameters ()
   :precondition (and (has Hammer))
   :effect (and (plane-broken)
            (probabilistic 0.9
             (and (not (gremlin-alive))))))
)
```

Figure 1: Example domain

conjunction through the plan to generate a logical formula which is a property holding in s. Learning in the property space supports information transfer between similar states (e.g., all states that share a given property) even before some of these states are visited. Our approach is efficient because fast classical planners can quickly derive these properties and because the number of properties is typically far smaller than the number of reachable states.

3.1 Definitions

We define the *all-outcomes determinization* D_d of domain D to produce, for every action a with m probabilistic outcomes $o_1, \ldots, o_m$, a set of m deterministic actions $a(o_1), \ldots, a(o_m)$, each having the same precondition as a and the corresponding outcome. From this point on, we will refer to D_d as the deterministic version or simply determinization of D.

Further, we define a *trajectory* to be a sequence $t = s, a_1(o_{j_1}), \ldots, a_n(o_{j_n})$ where s is the trajectory's starting state, and each action $a_k(o_{j_k})$ represents the j_k-th outcome of the probabilistic action a_k. We say that t is a *goal trajectory* if s modified by t's action sequence is a goal state. Further, we define a state *property* to be a conjunction of literals[2]. We say that a state s *possesses* property p if p holds in s. With

[2]Our algorithm can easily extend to properties specified using general logical formulas.

```
  (:action pick-up-0
   :parameters (?t - tool)
   :precondition (and (not (has ?t)))
   :effect (and (has ?t)))

  (:action tweak-0
   :parameters ()
   :precondition (and (has Screwdriver)
                      (has Wrench))
   :effect (and (plane-broken)))

  (:action smack-0
   :parameters ()
   :precondition (and (has Hammer))
   :effect (and (plane-broken)))

  (:action smack-1
   :parameters ()
   :precondition (and (has Hammer))
   :effect (and (plane-broken)
                (not (gremlin-alive))))
```

Figure 2: All-outcomes determinization of the example domain

each property p, we associate a unique *basis function* that has value 1 in s iff s possesses p.

We say that a property p (and the corresponding basis function b_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[3]. 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.

3.2 Example

We illustrate the above concepts with an example domain from Figure 1. The domain involves a gremlin that wants to sabotage an airplane and stay alive in the process.

To achieve the task, the gremlin has several tools that it can pick up. The gremlin can either tweak the airplane with a screwdriver and a wrench or smack it with the hammer. However, smacking will with high probability (0.9) lead to accidental detonation of airplanes fuel, which destroys the airplane but also kills the gremlin. For simplicity, all actions have a unit cost. The domain has five literals, *gremlin-alive*, *plane-broken*, *has(Hammer)*, *has(Wrench)*, and *has(Screwdriver)*, which we abbreviate as G, P, H, W, and S respectively.

Figure 2 shows the actions resulting from an all-outcomes determinization of GremlinWorld. All deterministic actions remain unmodified but the probabilistic *smack* action gets split into two. Taking $s = (G, \neg P, \neg H, \neg W, \neg S)$, the sequence $t = s$, *pick-up-0(Screwdriver)*, *pick-up-0(Wrench)*,

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

tweak-0() is a goal trajectory. An example property is $p = G \wedge \neg H \wedge \neg W$. Since p specifies truth values for three of five literals it is possessed by four states, e.g., the initial one. The basis function associated with p has value 1 in all four of these states. Property p enables the singleton set $T = \{t\}$ of goal trajectories. In GremlinWorld, any state in which the gremlin is dead (i.e., which involves $\neg G$) is a dead end. In particular, state $(\neg G, P, \neg H, \neg W, \neg S)$ is an implicit dead end, since the preconditions of the *pick-up* action instantiations don't preclude execution in this state. However, the goal is unreachable from it, since the gremlin is dead and cannot be resurrected. Our example domain has no explicit dead ends.

3.3 Algorithm Intuition

Consider a trajectory $t_g = s, a_1(o_{j_1}), \ldots, a_n(o_{j_n})$ that ends in a goal state. This is an indication that the sequence of *probabilistic* actions $a_1, \ldots, a_n$ is potentially causally important, since their outcomes $o_{j_1}, \ldots, o_{j_n}$ have positive probability. To discover the causal properties $p_1, \ldots, p_n$ that allow the successful execution of $a_1, \ldots, a_n$, we simply regress sequence t from the goal conjunction. We can now claim that action sequence $a_k, \ldots, a_n$ executed starting from *any* state possessing property p_k will lead us to the goal with positive probability, though the magnitude of the probability is yet unknown. Note that t essentially chooses specific outcomes per action and thus the execution of $a_1, \ldots, a_n$ may not always reach the goal. Nevertheless, every properties that enable any positive-probability trajectory to the goal may be important for our purposes because 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.

Referring to GremlinWorld domain, trajectory $t_g = s$, *pick-up-0(Hammer)*, *smack-0()*, where $s = (G, \neg P, \neg H, \neg W, \neg S)$, indicates the importance of the action sequence *pick-up(Hammer)*, *smack()*. Regressing t involves "rolling back" t_g's actions in reverse order starting from the goal, yielding properties $G \wedge H$ and $G \wedge \neg H$. Note that we can start executing the sequence *pick-up(Hammer)*, *smack()* in any state that possesses $G \wedge \neg H$ and hope to reach the goal with 10% probability.

To obtain goal trajectories all we need is to find plans that reach the goal in the *deterministic* version of the domain (by using a classical planner). Every such plan corresponds to a positive-probability trajectory in the original domain.

We can now define a new probabilistic planning problem over a state space consisting of these properties. In practice, the space of properties is much smaller than the original state space, since only the relevant causal structure is retained[4], giving us large reductions in space requirements. Solving this new problem amounts to learning the *weights* for the properties. The weights will be a quantitative measure of each property's importance. There are many imaginable ways to learn them; in this paper, we explore one of such methods — a modified version of RTDP.

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

Algorithm 1 ReTrASE

Input: probabilistic domain D, problem $P = \langle$init. state s_0, goal $G\rangle$, trial length L
declare global map M from basis functions to weights
declare global set DE of dead ends
compute global determinization D_d of D
// Do modified RTDP over the basis functions
for all $i = 1 : \infty$ **do**
  declare state $s \leftarrow s_0$
  declare $numSteps \leftarrow 0$
  while $numSteps < L$ **do**
    declare action $a' \leftarrow \arg\min_a \{ExpActCost(a, s)\}$
    $ModifiedBellmanBackup(a', s)$
    $s \leftarrow$execute action a' in s
    $numSteps \leftarrow numSteps + 1$
  end while
end for

function ExpActCost(action a, state s)
declare array $S_o \leftarrow$ successors of s under a
declare array $P_o \leftarrow$ probs of successors of s under a
return $cost(a) + \sum_i P_o[i] Value(S_o[i])$

function Value(state s)
if $s \in DE$ **then**
  return a large penalty // e.g., 1000000
else if some member f' of M holds in s **then**
  return $\min_{\text{basis functions } f \text{ that hold in } s} \{M[f]\}$
else
  $GetBasisFuncsForS(s)$
  return $Value(s)$
end if

function GetBasisFuncsForS(state s)
declare problem $p' \leftarrow \langle$init. state s, goal $G\rangle$
declare plan $pl \leftarrow$ DeterministicPlanner(D_d, p')
if $pl == none$ **then**
  insert s into DE
else
  declare basis function $f \leftarrow$ goal G
  declare $cost \leftarrow 0$
  for all $i = length(pl)$ through 1 **do**
    declare action $a \leftarrow pl[i]$
    $cost \leftarrow cost + cost(a)$
    $f \leftarrow (f \cup precond(a)) - effect(a)$
    insert $\langle f, cost\rangle$ into M if f isn't in M yet
  end for
end if

function ModifiedBellmanBackup(action a, state s)
for all basis functions f in s that enable a **do**
  $M[f] \leftarrow ExpActCost(a, s)$
end for

The weights reflect the fact 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 outcome for each of its actions. The sequence of outcomes 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 in the presence of some properties than others. Now, given that each state generally has several properties, what is the connection between the state's value and the their weights? In general, the relationship is quite complex: under the optimal policy, trajectories enabled by several properties may be possible. Therefore, the exact value of a state is a summation of weights over a subset of the state's properties. However, determining this subset is at least as hard as solving the MDP exactly. Instead, we approximate the state value by the *minimum* weight of all properties that 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.

3.4 Algorithm's Operation

RETRASE, whose pseudo code is presented in Algorithm 1, starts by computing the determinization D_d of the domain. We use D_d to rapidly compute the state properties. The algorithm explores the state space by running RTDP trials, memorizing all the dead ends and state properties it learns along the way. Whenever during state evaluation (line 20) RETRASEfinds a state that is neither a known dead-end nor has any property that holds in it,it uses a *deterministic* planner (line 34) to find a goal trajectory from this state in D_d. In our case, the role of a deterministic planner is performed by FF [Hoffman and Nebel, 2001]. To extract properties from a plan in D_d, we simply regress backwards through it (subroutine **GetBasisFuncsForS(.)**). Regression yields not only the basis functions but also an approximate cost of reaching the goal in D_d from any state with the given basis function via the given plan. We use this value to initialize the corresponding basis function's weight. If the deterministic planner can prove the non-existence of a plan or simply cannot find a plan within some time (we used a timeout of 30 seconds) the state in question is deemed to be a dead end (line 36).

For each state s visited by the modified RTDP, the **ModifiedBellmanBackup(.)** routine updates the weight of each basis function that enables the execution of the currently optimal action a' (line 48).

3.5 Theoretical Properties

A natural question about RETRASE is that of convergence. To answer it, we proved the following negative result:

Theorem 1 *There exist problems on which* RETRASE *does not converge.*

For the lack of space, we do not present the proof here but we outline the idea. By failing to converge we mean that, on some problems, RETRASE will indefinitely oscillate over a set of several policies with different expected costs. Moreover, the cost of the best policy in this set can be arbitrarily large compared to the optimal policy. The proof of the above result amounts to constructing such a problem.

The result is, in fact, even more general in the following sense. The set of basis functions RETRASE operates on at any point during its runtime is highly dependent on which states it has visited up to that point. There are problems that have *subsets* of basis functions that would drive RETRASE to oscillate indefinitely as described above. Should RETRASE discover strictly this subset of basis functions at some point during runtime, it will diverge. Therefore, on a given problem RETRASE may converge to a policy in one run and fail to converge at all in the next. The problem we devised for the theorem proof has the property that in any run (for any order of state space exploration) RETRASE will necessarily diverge on it. The classes of problems on which RETRASE diverges are hard to characterize generally. Predicting whether RETRASE may diverge on a particular problem requires a fairly involved analysis.

We stress, however, that the lack of theoretical guarantees is not indicative of a planner's practical success or failure. Indeed, several IPPC winners, including FFReplan, have a weak theoretical profile. The experimental results show that RETRASE too performs quite outstandingly on many of the planning community's benchmark problems.

4 EXPERIMENTAL RESULTS

Our goal in this section is to demonstrate two important properties of RETRASE – (1) scalability and (2) quality of solutions in complex, probabilistically interesting 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 hard problems than state-of-the-art approximate planners.

4.1 Implementation Details

RETRASE is implemented in C++ and uses miniGPT [Bonet and Geffner, 2005] as the base RTDP code. Our implementation is still in the prototype stage and does not yet fully support some of the PPDDL language features used to describe IPPC problems (e.g. universal quantification, disjunctive goals, rewards, etc.) We anticipate the support for them in the near future.

4.2 Experiment Setup

We report results on six problem sets — Triangle Tire World (TTW) from IPPC-06 and -08, Drive from IPPC-06, Exploding Blocks World (EBW) from IPPC-06 and -08, and Elevators from IPPC-06. In addition, we ran RETRASE on a few problems from IPPC-04. For implementation reasons we were unable to test on the remaining domains from these competitions. However, we emphasize that most of the six domains we evaluate on are probabilistically interesting and hard. Even the performance of the best IPPC participants on

most of them leaves a lot of room for improvement, which attests to their informativeness as testbeds for our planner.

To provide a basis for comparison, for the above domains we also present the results of the best IPPC participants. Namely, we give the results of the IPPC winner on that domain, of the overall winner of that IPPC, and ours. For the memory consumption experiment, we run two VI-family planners, LRTDP with the inadmissible FF heuristic (LRTDP$_{FF}$), and LRTDP$_{opt}$ – LRTDP with Atom-Min-1-Forward|Min-Min heuristic [Bonet and Geffner, 2005]. Both are among the best-known and top-performing planners of their type.

We ran RETRASE on the test problems under the restrictions resembling those of IPPC. Namely, for each problem, RETRASE had a maximum of 40 minutes for training, as did all the planners whose results we present here. RETRASE then had 30 attempts to solve each problem. In IPPC, the winner is decided by the success rate — the percentage of 30 trials in which a particular planner managed to solve the given problem. Accordingly, on the relevant graphs we present both RETRASE's success rate and that of its competitors. The runs were performed on a 2.8 GHz Intel Xeon processor with 2GB of RAM.

While analyzing the results, it is important to be aware that our RETRASE implementation is not optimized. Consequently, RETRASE's efficiency is likely even better than indicated by the experiments.

4.3 Comparing Scalability

We begin by showcasing the memory savings of RETRASE over LRTDP$_{opt}$ and LRTDP$_{FF}$on the example of Triangle Tire World domain. Figure 3 demonstrates the savings of RETRASE to increase dramatically with problem size. In fact, neither LRTDP flavor is able to solve past problem 8 as both run out of memory, whereas RETRASE handles all ten problems without any trouble. Scalability comparisons for other domains we tested on yield generally similar results so we don't present them here for the lack of space.

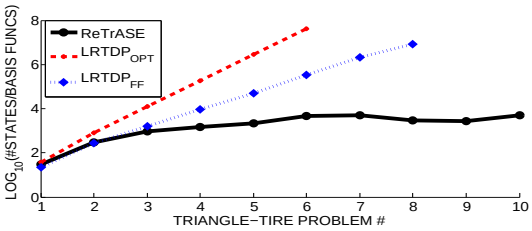

Figure 3: Memory usage on logarithmic scale: RETRASE is dramatically more efficient than both LRTDP$_{opt}$ and LRTDP$_{FF}$.

Other popular approximate algorithms (aside from LRTDP$_{FF}$) don't suffer from the scalability issues as much as LRTDP. Thus, it is more meaningful to compare RETRASE against them on the quality of solutions produced. As we show, RETRASE's scalability allows it to successfully compete on IPPC problems with any participant.

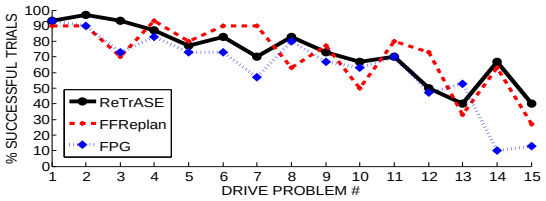

Figure 4: RETRASE is at par with the competitors on Drive.

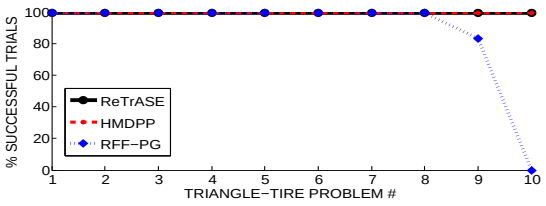

Figure 5: RETRASE achieves perfect success rate on Triangle Tire World-08.

4.4 Comparing Solution Quality: Success Rate

Continuing with the Triangle Tire World domain, we compare the success rates of RETRASE, RFF [Teichteil-Koenigsbuch *et al.*, 2008] — the overall winner of IPPC-08, and HMDPP [Keyder and Geffner, 2008] — the winner on this particular domain. We note that Triangle Tire World, perhaps, the most famous probabilistically interesting domain, was designed largely to confound solvers that rely on domain determinization [Little and Thiebaux, 2007], e.g. FFReplan; therefore, performance on it is particularly important for evaluating a new planner. Indeed, as Figure 5 shows, on this domain RETRASE ties with HMDPP by achieving the maximum possible success rate, 100%, on all ten problems and outperforms the competition winner, which cannot solve problem 10 at all and achieves only 83%-success rate on problem 9.

On the IPPC-06 Drive domain, RETRASE also fares well (Figure 4). Its average success rate is just ahead of the unofficial domain winner (FFReplan) and of the IPPC-06 winner (FPG), but the differences among all three are insignificant.

For the Exploding Blocks World domain on the IPPC-06 version (Figure 6), RETRASE dominates every other planner by a wide margin on almost every problem. Its edge is especially noticeable on the hardest problems, 11 through 15. On the most recent EBW problem set, from IPPC-08 (Figure 7), RETRASE performs very well too. Even though its advantage is not as apparent as in IPPC-06, it is nonetheless ahead of its competition in terms of the average success rate.

The Elevators and Triangle Tire World-06 domains are easier than the ones presented above. Surprisingly, on many of Elevators problems RETRASE did not converge within the allocated 40 minutes and was outperformed by several planners. We suspect this is due to a bad luck RETRASE has with basis functions in this domain. However, on TTW-06 RETRASE was the winner on every problem.

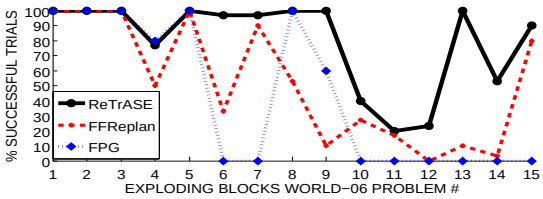

Figure 6: RETRASE dominates on Exploding Blocks World-06.

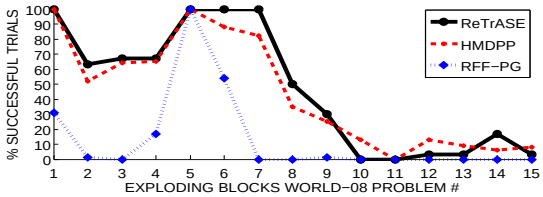

Figure 7: RETRASE outmatches all competitors on Exploding Blocks World-08, although by a narrow margin.

4.5 Comparing Solution Quality: Expected Cost

On problems where RETRASE achieves the maximum success rate it is interesting to ask how close the expected trajectory cost that its policy yields is to the optimal. The only way we could find out the expected cost of an optimal policy for a problem is by running an optimal planner on it. Unfortunately, the optimal planner we used, LRTDP$_{opt}$, scales enough to solve only relatively small problems. On such problems we found RETRASE to produce trajectories of expected cost within 5% of the optimal.

4.6 Comparison with FFHop

FFReplan has been a very powerful planner and a winner of at least one IPPC. However, recent benchmarks defeat it by exploiting its near-complete disregard for probabilities when computing a policy. Researchers have proposed a powerful improvement to FFReplan, FFHop [Yoon *et al.*, 2008], and demonstrated its capabilities on problems from IPPC-04. Unfortunately, due to the current lack of support for some of PPDDL language features we were not able to run RETRASE on most IPPC-04 domains. Table 1 compares the success rates of the two planners on the IPPC-04 problems we did test. Even though RETRASE performs better on these problems, the small size of the experimental base makes the comparison of RETRASE and FF-Hop inconclusive.

Problem name	FFHop	RETRASE
exploding-block	93.33%	100%
g-tire-problem	60%	70%

Table 1: Success rates on some IPPC-04 problems.

While we do not test on all IPPC-domains our current experimental evaluation clearly demonstrate RETRASE's scalability improvements over the VI-family planners and its at-par or better performance on many competition problems compared to state-of-the-art systems.

5 RELATED WORK

Besides basis function approximation (discussed in Section 1) other flavors of dimensionality reduction include algebraic and binary decision diagram (ADD/BDD), and principle component analysis (PCA) based methods. SPUDD, Symbolic LAO*, and Symbolic RTDP are optimal algorithms that exploit ADDs and BDDs for a compact representation and efficient backups in an MDP [Hoey *et al.*, 1999; Feng and Hansen, 2002]. In practice these optimal algorithms do not scale to large problems. APRICODD, an approximation scheme developed over SPUDD [St-Aubin *et al.*, 2000], showed promise, but it is not clear whether it is competitive with today's top methods since it hasn't been applied to the competition domains.

Some researchers have applied non-linear techniques like exponential-PCA and NCA for dimensionality reduction [Roy and Gordon, 2003; Keller *et al.*, 2006]. These methods assume the original state space to be continuous and hence are not applicable to typical planning benchmarks.

Most basis function based techniques are not applied in nominal domains, however, a notable exception is FPG [Buffet and Aberdeen, 2006]. It performs policy search and represents the policy compactly with a neural network. Our experiments demonstrate that RETRASE outperforms FPG consistently on several domains.

RETRASE is a further development of the system described in [Kolobov *et al.*, 2008]. The critical difference, however, is that in this previous work the algorithm tried to find *several* deterministic plans from each state. Looking for just one, as RETRASE does now, turns out to greatly increase scalability, with almost no sacrifices in solution quality. Our algorithm is also related in spirit to the probabilistic planners that use determinized domains for probabilistic planning. The most popular of these is FFReplan, [Yoon *et al.*, 2007], the competition winner in IPC-04. Similar planners include Tempastic [Younes and Simmons, 2004], precautionary planning [Foss *et al.*, 2007], and FFHop [Yoon *et al.*, 2008].

The idea of using determinization followed by regression has parallels to some research on relational MDPs, which uses first-order regression on optimal plans in small problem instances to construct a policy for large problems in a given domain [Gretton and Thiebaux, 2004; Sanner and Boutilier, 2006]. However, our function aggregation and weight learning methods are completely different from theirs.

6 CONCLUSION

Many popular decision-theoretic algorithms for solving MDPs over nominal domains (e.g. those used in IPPC) tabulate the value function and consequently suffer from high memory requirements. To resolve the issue, a number of approaches have been proposed, including replanning in a determinized domain and dimensionality reduction. Both have shown promise, but in nominal domains determinization has trouble with probabilistic subtleties, while dimensionality re-

duction relies on humans to specify the necessary basis functions. This paper makes the following contributions:

- We define a set of domain-specific state properties, each enabling a trajectory from a state to the goal. We compute 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 RETRASE scales drastically better than optimal planners and outmatches state-of-the-art planners on hard problems from several IPPC competitions.

In the future, RETRASE would benefit from the added support of more PPDDL language features like rewards, existential-, and universal quantification. Our initial experimental results suggest that RETRASE may perform very well at the next IPPC.

Acknowledgments

We would like to thank Eytan Adar, Jessica Chang, Peng Dai, Raphael Hoffmann, Subbarao Kambhampati, Cynthia Matuszek, and the anonymous reviewers for their insightful comments. We are also grateful to Blai Bonet for making the code of miniGPT package freely available. This work was supported by Office of Naval Research grant N00014-06-1-0147 and the WRF / TJ Cable Professorship.

References

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

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

[3] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[4] D. Bertsekas and J.Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, 1996.

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

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

[7] Olivier Buffet and Douglas Aberdeen. The factored policy gradient planner (ipc-06 version). In *Fifth International Planning Competition at ICAPS'06*, 2006.

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

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

[10] Geoff Gordon. Stable function approximation in dynamic programming. In *ICML*, pages 261–268, 1995.

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

[12] C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *IJCAI'03*, Acapulco, Mexico, 2003.

[13] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored MDPs. *Journal of Artificial Intelligence Research*, 19:399–468, 2003.

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

[15] J. Hoffman and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[16] Philipp Keller, Shie Mannor, and Doine Precup. Automatic basis function construction for approximate dynamic programming and reinforcement learning. In *ICML'06*, pages 449–456, 2006.

[17] Emil Keyder and Hector Geffner. The HMDPP planner for planning with probabilities. In *Sixth International Planning Competition at ICAPS'08*, 2008.

[18] Andrey Kolobov, Mausam, and Daniel Weld. Regressing deterministic plans for mdp function approximation. In *ICAPS'08 Workshop on a Reality Check for Planning and Scheduling Under Uncertainty*, 2008.

[19] Iain Little and Sylvie Thiebaux. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*, 2007.

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

[21] Nicholas Roy and Geoffrey Gordon. Exponential family PCA for belief compression in POMDPs. In *NIPS'02*, pages 1043–1049. MIT Press, 2003.

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

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

[24] Florent Teichteil-Koenigsbuch, Guillaume Infantes, and Ugur Kuter. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. In *Sixth International Planning Competition at ICAPS'08*, 2008.

[25] Sungwook Yoon, Alan Fern, and Robert Givan. FF-Replan: A baseline for probabilistic planning. In *ICAPS'07*, pages 352–359, 2007.

[26] Sungwook Yoon, Alan Fern, Subbarao Kambhampati, and Robert Givan. Probabilistic planning via determinization in hindsight. In *AAAI'08*, 2008.

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