Artificial Intelligence 189 (2012) 19–47

Contents lists available at SciVerse ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Discovering hidden structure in factored MDPs

Andrey Kolobov *, Mausam, Daniel S. Weld

Dept. of Computer Science and Engineering, University of Washington, Seattle, WA 98195, United States

ARTICLE INFO

Article history:
Received 1 August 2010
Received in revised form 8 April 2012
Accepted 9 May 2012
Available online 15 May 2012

Keywords:
Markov Decision Process
MDP
Planning under uncertainty
Generalization
Abstraction
Basis function
Nogood
Heuristic
Dead end

ABSTRACT

Markov Decision Processes (MDPs) describe a wide variety of planning scenarios ranging from military operations planning to controlling a Mars rover. However, today's solution techniques scale poorly, limiting MDPs' practical applicability. In this work, we propose algorithms that automatically discover and exploit the hidden structure of factored MDPs. Doing so helps solve MDPs faster and with less memory than state-of-the-art techniques.

Our algorithms discover two complementary state abstractions — *basis functions* and *nogoods*. A basis function is a conjunction of literals; if the conjunction holds true in a state, this guarantees the existence of at least one trajectory to the goal. Conversely, a nogood is a conjunction whose presence implies the *non*-existence of any such trajectory, meaning the state is a dead end. We compute basis functions by regressing goal descriptions through a determinized version of the MDP. Nogoods are constructed with a novel machine learning algorithm that uses basis functions as training data.

Our state abstractions can be leveraged in several ways. We describe three diverse approaches — GOTH, a heuristic function for use in heuristic search algorithms such as RTDP; ReTrASE, an MDP solver that performs modified Bellman backups on basis functions instead of states; and SixthSense, a method to quickly detect dead-end states. In essence, our work integrates ideas from deterministic planning and basis function-based approximation, leading to methods that outperform existing approaches by a wide margin.

1. Introduction

Markov Decision Processes (MDPs) are a popular framework for modeling problems involving sequential decision-making under uncertainty. Examples range from military-operations planning to user-interface adaptation to the control of mobile robots [1,36]. Unfortunately, however, existing techniques for *solving* MDPs, i.e. deciding which actions to execute in various situations, scale poorly, and this dramatically limits MDPs' practical utility.

Humans perform surprisingly well at planning under uncertainty, largely because they are able to recognize and reuse abstractions, generalizing conclusions across different plans. For example, after realizing that the walls of a particular Mars crater are too steep for the rover to escape, a human planner would abandon attempts to collect any of the rock samples in the crater, while a traditional MDP solver might rediscover the navigational problem as it considered collecting each sample in turn.

This article presents new algorithms for automatically discovering and exploiting such hidden structure in MDPs. Specifically, we generate two kinds of abstraction, *basis functions* and *nogoods*, each of which describes sets of states that share a similar relationship to the planning goal. Both basis functions and nogoods are represented as logical conjunctions of state variable values, but they encode diametrically opposite information. When a basis function holds in a state, this guarantees

* Corresponding author.
E-mail addresses: akolobov@cs.washington.edu (A. Kolobov), mausam@cs.washington.edu (Mausam), weld@cs.washington.edu (D.S. Weld).

0004-3702/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.artint.2012.05.002

that a certain trajectory of action outcomes has a positive probability of reaching the goal. Our algorithms associate weights with each basis function, encoding the relative quality of the different trajectories. In contrast, when a nogood holds in a state, it signifies that the state is a dead-end; *no* trajectory can reach the goal from this state. Continuing the Mars rover example, a conjunction that described presence in the steep-walled crater would be a nogood.

Our notions of basis function and nogood are similar to the rules learned in logical theories in explanation-based learning and constraint satisfaction [27,14], but our work applies them in a probabilistic context (e.g., learns weights for basis functions) and provides new mechanisms for their discovery. Previous MDP algorithms have also used basis functions [21, 39], but to perform generalization *between different problems in a domain* rather than during the course of solving a single problem. Other researchers have used hand-generated basis functions in a manner similar to ours [22,23,20], but we present methods for their automatic generation.

1.1. Discovering nogoods and basis functions

We generate basis functions by regressing goal descriptions along an action outcome trajectory using a determinized version of the probabilistic domain theory. Thus, the trajectory is potentially executable in all states satisfying the basis function. This justifies performing Bellman backups on basis functions, rather than states — generalizing experience across similar states. Since many basis functions typically hold in a given state, the value of a state is a complex function of the applicable basis functions.

We discover nogoods using a novel machine learning algorithm that operates in two phases. First it generates candidate nogoods with a probabilistic sampling procedure using basis functions and previously discovered dead ends as training data. It then tests the candidates with a planning graph [6] to ensure that no trajectories to the goal could exist from states containing the nogood.

1.2. Exploiting nogoods and basis functions

We present three algorithms that leverage our basis function and nogood abstractions to speed MDP solution and reduce memory usage.

- GOTH uses a full classical planner to *generate a heuristic function* for an MDP solver for use as an initial estimate of state values. While classical planners have been known to provide an informative approximation of state value in probabilistic problems, they are too expensive to call from every newly visited state. GOTH amortizes this cost across multiple states by associating weights to basis functions and thus generalizing the heuristic computation. Empirical evaluation shows GOTH to be an informative heuristic that saves MDP solvers considerable time and memory.
- ReTrASE is a *self-contained MDP solver* based on the same information-sharing insight as GOTH. However, unlike GOTH, which sets the weight of each basis function only once to provide the starting guess at states' values, ReTrASE *learns* the basis functions' weights by evaluating each function's "usefulness" in a decision-theoretic way. By aggregating the weights, ReTrASE constructs a state value function approximation and, as we show empirically, produces better policies than the participants of the International Probabilistic Planning Competition (IPPC) on many domains while using little memory.
- SixthSense is a *method for quickly and reliably identifying dead ends*, i.e., states with no possible trajectory to the goal, in MDPs. In general, this problem is intractable — one can prove that determining whether a given state has a trajectory to the goal is PSPACE-complete [19]; therefore, it is unsurprising that modern MDP solvers often waste considerable resources exploring these doomed states. SixthSense acts as a submodule of an MDP solver, helping it detect and avoid dead ends. SixthSense employs machine learning, using basis functions as training data, and is guaranteed never to generate false positives. The resource savings provided by SixthSense are determined by the fraction of dead ends in the MDP's state space and reach 90% on some IPPC benchmark problems.

In the rest of the paper, we present these algorithms, discuss their theoretical properties, and evaluate them empirically. Section 2 reviews the background material and introduces relevant definitions, illustrating these with a running example. Sections 3, 4, and 5 present descriptions of and empirical results on GOTH, ReTrASE, and SixthSense respectively. Section 6 discusses potential extensions of the presented algorithms. Finally, Section 7 describes the related work and Section 8 concludes the paper.

2. Preliminaries

2.1. Example

Throughout the paper, we will be illustrating various concepts with the following scenario, called GremlinWorld. Consider a gremlin that wants to sabotage an airplane and stay alive in the process. To achieve the task, the gremlin can pick up several tools. The gremlin can either tweak the airplane with a screwdriver and a wrench, or smack it with a hammer.

```
(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))))))
)

(define (problem GremlinProb)
  (:domain GremlinWorld)
  (:init (gremlin-alive))
  (:goal (and (gremlin-alive) (plane-broken)))
)
```

Fig. 1. A PPDDL-style description of the example MDP, GremlinWorld, split into domain and problem parts.

However, smacking will, with high probability, lead to accidental detonation of the airplane's fuel, which destroys the airplane but also kills the gremlin. Fig. 1 describes this setting in Probabilistic Planning Domain Description Language (PPDDL). As we introduce relevant terminology in subsequent subsections, we will formally define the corresponding MDP.

2.2. Background

Markov Decision Processes (MDPs). In this paper, we focus on probabilistic planning scenarios modeled by discrete factored stochastic-shortest-path (SSP) MDPs with an initial state. In general, MDPs are defined as tuples of the form $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C} \rangle$, where

- $\mathcal{S}$ is a set of states.
- $\mathcal{A}$ is a set of actions.
- $\mathcal{T}$ is a transition function $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ giving the probability of moving from s_i to s_j by executing action a.
- $\mathcal{C}$ is a map $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ specifying action costs.

The MDPs we consider in this paper are a specific kind defined as a tuple $\langle \mathcal{X}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, where $\mathcal{A}, \mathcal{T}$, and $\mathcal{C}$ are as above and

- $\mathcal{X}$ is a set of *state variables* s.t. every conjunction of literals over all variables in $\mathcal{X}$ is a state of the MDP. Therefore, with a slight abuse of notation, we can set $\mathcal{S} = 2^{\mathcal{X}}$ in the general MDP definition.
- $\mathcal{G}$ is a set of (absorbing) goal states.
- s_0 is the start state.
- All action costs are positive, i.e. $\mathcal{C}$ is a map $\mathcal{S} \times \mathcal{A} \to \mathbb{R}^+$.[1]

We assume that both the state space ($2^{\mathcal{X}}$) and the action space ($\mathcal{A}$) are finite. Another assumption we make is that each action of the MDP has a *precondition*, a conjunction of literals describing the states in which the action can be executed.

[1] This requirement is actually stricter, although much easier to state, than in the original SSP MDP's definition [5]. The original statement allows costs to be completely arbitrary as long as each policy that does not reach the goal incurs an infinite cost. However, the algorithms in this paper apply to all MDPs falling under that definition as well.

```
(: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))))
```

Fig. 2. All-outcomes determinization of the GremlinWorld domain.

Our example, GremlinWorld, can be formulated as an MDP using five state variables, *gremlin-alive*, *plane-broken*, *has*(*Hammer*), *has*(*Wrench*), and *has*(*Screwdriver*), abbreviated as G, P, H, W, and S respectively. Therefore, $\mathcal{X} = \{G, P, H, W, S\}$. The problem involves five actions, $\mathcal{A} = \{$*pick-up*(*Screwdriver*), *pick-up*(*Wrench*), *pick-up*(*Hammer*), *tweak*( ), *smack*( )$\}$. Each action has a precondition; e.g., the *smack*( ) action's precondition is a single-literal conjunction (*has Hammer*), so *smack*( ) can only be used in states where the gremlin has a hammer. Actions' preconditions and effects compactly specify the transition function $\mathcal{T}$. For simplicity, we make $\mathcal{C}$ assign the cost of 1 to all actions, which conforms to the restriction on $\mathcal{C}$ imposed by the SSP MDP definition. $\mathcal{G}$ is the set of all states where the gremlin is alive and the airplane is broken. Finally, we assume that the gremlin starts alive with no tools and the airplane is originally intact, i.e. $s_0 = (G, \neg P, \neg H, \neg W, \neg S)$.

Solving an MDP means finding a good (i.e., cost-minimizing) policy $\pi : \mathcal{S} \to \mathcal{A}$ that specifies the actions the agent should take to eventually reach the goal. The optimal expected cost of reaching the goal from a state s, termed the *optimal value function* $V^*(s)$, satisfies the following conditions, called *Bellman equations*:

$$V^*(s) = 0 \quad \text{if } s \in \mathcal{G}, \text{ otherwise}$$
$$V^*(s) = \min_{a \in \mathcal{A}} \left[\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') V^*(s') \right].$$

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

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

Solution methods. The above equations suggest a dynamic programming-based way of finding an optimal policy, called *value iteration* (VI) [3]. VI iteratively updates state values using Bellman equations in a *Bellman backup* until the values converge. VI has given rise to many improvements. Trial-based methods, e.g., RTDP [2], try to reach the goal multiple times (in multiple *trials*) and update the value function over the states in the trial path, successively improving the policy during each Bellman backup. A popular variant, LRTDP, adds a termination condition to RTDP by labeling states whose values have converged as 'solved' [7]. Compared to VI, trial-based methods save space by considering fewer irrelevant states. LRTDP serves as the testbed in our experiments, but the approach we present can be used by many other search-based MDP solvers as well, e.g., LAO* [24].

Determinization. Successes of a number of planners starting with FFReplan [42] have demonstrated the promise of *determinizing* the *domain* (the set of all actions) of the given MDP, i.e. disregarding the probabilities in the transition function, and working only with the state transition graph. Our techniques use the *all-outcomes* determinization [42] D_d of the domain D at hand. Namely, note in the example in Fig. 1 that each action a, besides precondition c, has *outcomes* $o_1, \ldots, o_n$ with respective probabilities $p_1, \ldots, p_n$. For example, the *smack*( ) action has outcomes $o_1 = P \wedge \neg G$ with $p_1 = 0.9$ and $o_2 = P$ with $p_1 = 0.1$. The all-outcomes determinization D_d, whose example for the GremlinWorld domain is shown in Fig. 2, contains, for every action a in the original domain, the set of deterministic actions $a_1, \ldots, a_n$, each with a's precondition c and effect o_i. D_d, coupled with a description of the state space, the initial state, and the goal, can be viewed as a deterministic MDP in which a plan from a given state to the goal exists if and only if a corresponding trajectory has a positive probability in the original probabilistic domain D. Importantly, the state of the art in classical planning makes solving a deterministic problem much faster than solving a probabilistic problem of a comparable size. Our abstraction framework exploits these facts to efficiently extract the structure of the given MDP by finding plans in D_d and processing them as shown in Section 2.3.

Heuristic functions. We define a *heuristic function*, hereafter termed simply as *heuristic*, as a value function that initializes the state values for an MDP algorithm. Heuristic values tend to be derived, automatically or otherwise, from the structure of the problem at hand. The properties of the heuristic determine how quickly a planning algorithm converges and whether the resulting policy is optimal. Algorithms like VI, which update the value of every state in each iteration, converge to the optimal policy faster the closer the heuristic is to V^*. In trial-based algorithms like LRTDP, heuristics help avoid visiting irrelevant states. To guarantee convergence to an optimal policy, trial-based MDP solvers typically require the heuristic to be admissible, i.e. to never overestimate V^* (importantly, admissibility is not a requirement for convergence to *a* policy). However, inadmissible heuristics tend to be more *informative* in practice, approximating V^* better on average. Informativeness often translates into a smaller number of explored states (and the associated memory savings) with reasonable sacrifices in optimality. In this paper, we adopt the number of states visited by a planner under the guidance of a heuristic as the measure of that heuristic's informativeness and show how basis functions let us derive a highly informative heuristic, GOTH, at the cost of admissibility.

A successful class of MDP heuristics is based on the all-outcomes determinization of the probabilistic domain D at hand [8]. To obtain a value for state s in D, determinization heuristics try to approximate the cost of a plan from s to a goal in D_d (finding a plan itself even in this relaxed version of an MDP is generally NP-hard). For instance, the FF heuristic [26], denoted h_{FF}, ignores the negative literals (the *delete effects*) in the outcomes of actions in D_d and attempts to find the cost of the cheapest solution to this new relaxed problem. As h_{FF} is, in our experience, the most informative general MDP heuristic, we use it as the baseline to evaluate the performance of GOTH.

Planning graph. Our work makes use of the planning graph data structure [6], a directed graph alternating between proposition and action "levels". The 0-th level contains a vertex for each literal present in an initial state s. Odd levels contain vertices for all actions, including a special no-op action, whose preconditions are present (and pairwise "nonmutex") in the previous level. Subsequent even levels contain all literals from the effects of the previous action level. Two literals in a level are *mutex* if all actions achieving them are pairwise mutex at the previous level. Two actions in a level are mutex if their effects are inconsistent, one's precondition is inconsistent with the other's effect, or one of their preconditions is mutex at the previous level. As levels increase, additional actions and literals appear (and mutexes disappear) until a fixed point is reached. Graphplan [6] uses the graph as a polynomial-time reachability test for the goal, and we use it in a procedure to discover nogoods in Section 5.

2.3. Definitions and essentials

Let an *execution trace* $e = s, a_1, s_1, \ldots, a_n, s_n$, a sequence where s is the trace's starting state, a_1 is a probabilistic action applied in s that yielded state s_1, and so on. An example of an execution trace from GremlinWorld is $e' = (G, \neg P, \neg H, \neg W, \neg S)$, *pick-up*(*Hammer*), $(G, \neg P, H, \neg W, \neg S)$, *smack*( ), $(G, P, H, \neg W, \neg S)$.

We define a *trajectory* of an execution trace e to be a sequence

$$t(e) = s, out(a_1, 1, e), \ldots, out(a_n, n, e)$$

where s is e's starting state, and $out(a_k, k, e)$ is a conjunction of literals representing the particular outcome of action a_k that was sampled at the k-th step of e's execution. E.g., $t(e') = (G, \neg P, \neg H, \neg W, \neg S),\ H,\ P$ is a trajectory of the example execution trace e'.

We say that $t(e)$ is a *goal trajectory* if the last state s_n of e is a goal state; $t(e')$ just shown is a goal trajectory. A *suffix* of $t(e)$ is a sequence

$$t_i(e) = out(a_i, i, e), \ldots, out(a_n, n, e)$$

for some $1 \leqslant i \leqslant n$.

Suppose we are given an MDP and a goal trajectory $t(e)$ of some execution trace in this MDP. Let *prec*(a) denote the precondition of a (a literal conjunction) and *lit*(c) stand for the set of literals forming conjunction c. Imagine using t to generate the following sequence of literal conjunctions:

$$b_0 = \mathcal{G},$$
$$b_i = \bigwedge \bigl[\bigl[lit(b_{i-1}) \cup lit\bigl(out(a_{n-i+1}, n-i+1, e)\bigr)\bigr] \setminus lit\bigl(prec(a_{n-i+1})\bigr)\bigr] \quad \text{for } 1 \leqslant i \leqslant n.$$

This can be done with a simple multistep procedure. We start with $b_0 = \mathcal{G}$, the MDP's goal conjunction. Afterwards, at step $i \geqslant 1$, we first remove from b_{i-1} the literals of action a_{n-i+1}'s outcome at the $(n-i+1)$-th step of e. Then, we conjoin the result to the literals of a_{n-i+1}'s precondition, obtaining conjunction b_i. We call this procedure *regression of the goal through trajectory* $t(e)$, or *regression* for short.

As an example, consider regressing trajectory $t(e')$ from GremlinWorld. In this case, $b_0 = \mathcal{G} = G \wedge P$. First we remove from b_0 literal P, the outcome of the *last* action, *smack*( ), of e'. The result is G. Then, we add to it the precondition of *smack*( ), literal H, producing $G \wedge H$. Thus, $b_1 = G \wedge H$. Similarly, we remove from b_1 the outcome of *pick-up*(*Hammer*) and add the precondition of this action, which is empty, to the result, obtaining $b_2 = G$. At this point regression terminates.

A *basis function* is defined to be a literal conjunction b produced at some step of regressing the goal through some trajectory. Whenever all literals of a basis function (or of a conjunction of literals in general) are present in state s we say that the conjunction *holds in* or *represents* s. For instance, $b_1 = G \wedge H$ from the above example holds in state $(G, \neg P, H, \neg W, S)$. An alternative view of a basis function b is a mathematical function $f_b : \mathcal{S} \to \{0, 1\}$ having the value of 1 in all states in which conjunction b holds and 0 in all others.

Basis functions are a central concept behind the algorithms in this paper, so it is important to understand the intuition behind them. Any goal trajectory is potentially a causally important sequence of actions. Regressing it gives us preconditions for the trajectory's suffixes. Basis functions are exactly these trajectory suffix preconditions. Thus, regression of the trajectories can be thought of as unearthing the relevant causal structure necessary for the planning task at hand. Moreover, our basis functions *are* that causal structure.

There are often many trajectories whose preconditions are consistent with (i.e., are a subconjunction of) a given basis function. We say that a basis function b *enables* a set of goal trajectories T if the goal can be reached from any state represented by b by following any of the trajectories in T assuming that Nature chooses the "right" outcome for each action of the trajectory.

Since each basis function is essentially a precondition (for a trajectory), it typically holds in many states of the MDP at hand. Therefore, obtaining a goal trajectory $t(e)$ from some state lets us *generalize* this qualitative reachability information to many other states via basis functions yielded by regressing the goal through this trajectory. Moreover, $t(e)$ may have interesting numeric characterizations, e.g. cost, probability of successful execution, etc. To generalize these quantitative descriptions across many states as well, we associate a *weight* with each basis function. The semantics of basis function weight depends on the algorithm, but in general it reflects the quality of the set of trajectories enabled by the basis function.

Now, consider the value of an MDP's state. As preconditions, basis functions tell us which goal trajectories are possible from that state. Basis function weights tell us how "good" these trajectories are. Since the quality of the set of goal trajectories possible in a state is a strong indicator of the state's value, knowing basis functions with their weights allows for approximating the state value function.

As we just showed, a problem's causal structure can be efficiently derived from its goal trajectories via regression. Thus, a relatively cheap source of trajectories would give us a way to readily extract the structure of the problem. Fortunately, at least two such methods exist. The first one is based on this insight that whenever a trial in an MDP solver reaches the goal we get a trajectory "for free", as a byproduct of the solver's usual computation. The caveat with using this technique as the primary strategy of getting trajectories is the time it takes an MDP solver's trials to start attaining the goal. Indeed, the majority of trials at the beginning of planning terminate in states with no path to the goal, and it is at this stage that knowing the problem's structure would be most helpful for improving the situation. Therefore, our algorithms mostly rely on a different trajectory generation approach. Note that any trajectory in an MDP is a plan in the all-outcomes determinization D_d of that MDP and vice versa. Since classical planners are very fast, we can use them to quickly find goal trajectories in D_d from several states of our choice.

By definition, basis functions represent only the states from which reaching the goal is possible. However, MDPs also contain another type of states, *dead ends*, that fall outside of the basis function framework as presented so far. Such states, in turn, can be classified into two kinds; *explicit dead ends*, in which no actions are applicable, and *implicit* ones, which do have applicable actions but no sequence of them leads to the goal with a positive probability. In GremlinWorld, there are no explicit dead ends but every state with literal $\neg G$ is an implicit dead end.

To extend information generalization to dead ends as well, we consider another kind of literal conjunctions that we call *nogoods*. Nogoods' defining property is that any state in which a nogood holds is a dead end. Notice the duality between nogoods and basis functions: both have exactly the same form but give opposite guarantees about a state. Whereas a state represented by a basis function provably cannot be a dead end, a state represented by a nogood certainly is one. Despite the representational similarity, identifying nogoods is significantly more involved than discovering basis functions. Fortunately, the duality between the two allows using the latter to derive the former and collect the corresponding benefits, as one of the algorithms we are about to present, SixthSense, demonstrates.

3. GOTH heuristic

3.1. Motivation

Our presentation of the abstraction framework begins with an example of its use in a heuristic function. As already mentioned, heuristics reduce trial-based MDP solvers' resource consumption by helping them avoid visiting many states (and memoizing corresponding state-value pairs) that are not part of the final policy. The most informative MDP heuristics, e.g., h_{FF}, are based on the all-outcomes determinization of the domain. However, although efficiently computable, such heuristics add an extra level of relaxation of the original MDP, besides determinizing it. For instance, h_{FF} is liable to highly underestimate the state's true cost because in addition to discarding the domain's probabilities it ignores actions' delete effects (i.e., negative literals, such as $\neg G$, in actions' outcomes) in the determinized version.

On the other hand, a lot of promise has been shown recently by several probabilistic planners that solve *full* (non-relaxed) determinizations, e.g., FFReplan, HMDPP [29], and others. It is natural to wonder, then, whether the improved heuristic estimates of using a full classical planner on non-relaxed determinized domains would provide enough gains to compensate for the potentially increased cost of heuristic computation.

As we show in this section, the answer is "No and Yes". We propose a new heuristic called GOTH (**G**eneralization **O**f **T**rajectories **H**euristic) [32], which *efficiently* produces heuristic state values using deterministic planning. The most straightforward implementation of this method, in which a classical planner is called every time a state is visited for the first time, does produce better heuristic estimates and reduces search but the cost of so many calls to the classical planner vastly outweighs any benefits. The crucial observation we make is that basis functions provide a way to amortize these expensive planner calls by generalizing the resulting heuristic values to give guidance on similar states. By performing this generalization in a careful manner, one may dramatically reduce the amount of classical planning needed, while still providing more informative heuristic values than heuristics with more levels of relaxation.

3.2. GOTH *description*

Given a problem P over a probabilistic domain D, an MDP solver using GOTH starts with GOTH's initialization. During initialization, GOTH determinizes D into its classic counterpart, D_d (this operation needs to be done only once). Our implementation performs the all-outcomes determinization because it is likely to give much better value estimates than the single-outcome one [42]. However, more involved flavors of determinization described in the Related Work section may yield even better estimation accuracy.

Calling a deterministic planner. Once D_d has been computed, the probabilistic planner starts exploring the state space. For every state s that requires heuristic initialization, GOTH first checks if it is an explicit dead end. This check is in place for efficiency, since GOTH should not try to use more expensive methods of analysis on such states.

For state s that is not an explicit dead end GOTH constructs a problem P_s with the original problem's goal and s as the initial state, feeding P_s along with D_d to a classical planner, denoted as *DetPlan* in the pseudocode of Algorithm 1, and setting a timeout. If s is an implicit dead end *DetPlan* either proves this or unsuccessfully searches for a plan until the timeout. In either case, it returns without a plan, at which point s is presumed to be a dead end and assigned a very high value taken to be ∞. If s is not a dead end, *DetPlan* usually returns a plan from s to the goal. The cost of this plan is taken as the heuristic value of s. Sometimes *DetPlan* may fail to find a plan before the timeout, leading the MDP solver to falsely assume s to be a dead end. In practice, we have not seen this hurt GOTH's performance.

Regression-based generalization. By using a full-fledged classical planner, GOTH produces more informative state estimates than h_{FF}, as evidenced by our experiments. However, invoking the classical planner for every newly encountered state is costly; as it stands, GOTH would be prohibitively slow. To ensure speed, we modify the procedure based on the insight about basis functions and their properties as shown in the pseudocode of Algorithm 1. Whenever GOTH computes a deterministic plan, it first regresses it, as described in Section 2. Then it memoizes the resulting basis functions with associated weights set to the costs of the regressed plan suffixes. When GOTH encounters a new state s, it minimizes over the weights of all basis functions stored so far that hold in s. In doing so, GOTH sets the heuristic value of s to be the cost of the cheapest currently known trajectory that originates at s. Thus, the weight of one basis function can become *generalized* as the heuristic value of many states. This way of computing a state's value is very fast, and GOTH employs it *before* invoking a classical planner. However, s's heuristic value may be needed even before GOTH has any basis function that holds in s. In this case, GOTH uses the classical planner as described above, computing a value for s and augmenting its basis function set. Evaluating a state first by generalization and then, if generalization fails, by classical planning greatly amortizes the cost of each classical solver invocation and drastically reduces the computation time compared to using a deterministic planner alone.

Weight updates. Different invocations of the deterministic planner occasionally yield the same basis function more than once, each time potentially with a new weight. Which of these weights should we use? The different weights are caused by a variety of factors, not the least of which are non-deterministic choices made within the classical planner.[2] Thus, the basis function weight from any given invocation may be unrepresentative of the cost of the plans for which this basis function is a precondition. For this reason, it is generally beneficial to assign a basis function *the average* of the weights computed for it by classical planner invocations so far. This is the approach we take on line 27 of Algorithm 1. Note that to compute the average we need to keep the number of times the function has been re-discovered.

[2] For instance, LPG [18], which relies on a stochastic local search strategy for action selection, may produce distinct paths to the goal even when invoked twice from the same state, with concomitant differences in basis functions and/or their weights.

Algorithm 1 GOTH heuristic.

```
1: **Input:** probabilistic domain $D$, problem $P = \langle$init. state $s_0$, goal $G\rangle$, determinization routine $Det$, classical planner $DetPlan$, timeout $T$, state $s$
2: **Output:** heuristic value of $s$
3:
4: compute global determinization $D_d = Det(D)$
5: declare global map $M$ from basis functions to weights
6:
7: **function computeGOTH**(state $s$, timeout $T$)
8: **if** no action $a$ of $D$ is applicable in $s$ **then**
9:    return a large penalty // e.g., 1000 000
10: **else if** a nogood holds in $s$ **then**
11:    return a large penalty // e.g., 1 000 000
12: **else if** some member $f'$ of $M$ holds in $s$ **then**
13:    return $\min_{\text{basis functions } f \text{ that subsume } s}\{M[f]\}$
14: **else**
15:    declare problem $P_s \leftarrow \langle$init. state $s$, goal $G\rangle$
16:    declare plan $pl \leftarrow DetPlan(D_d, P_s, T)$
17:    **if** $pl == none$ **then**
18:       return a large penalty // e.g., 1 000 000
19:    **else**
20:       declare basis function $f \leftarrow$ goal $G$
21:       declare $weight \leftarrow 0$
22:       **for all** $i = length(pl)$ through 1 **do**
23:          declare action $a \leftarrow pl[i]$
24:          $weight \leftarrow weight + Cost(s, a)$
25:          $f \leftarrow (f \cup precond(a)) - effect(a)$
26:          **if** $f$ is not in $M$ **then**
27:             insert $\langle f, weight\rangle$ into $M$
28:          **else**
29:             update $M[f]$ by incorporating $weight$ into $M[f]$'s running average
30:          **end if**
31:       **end for**
32:       **if** SchedulerSaysYes **then**
33:          learn nogoods from discovered dead ends
34:       **end if**
35:       return $weight$
36:    **end if**
37: **end if**
```

Dealing with implicit dead ends. The discussion so far has ignored an important detail. When a classical planner is called on an implicit dead end, by definition no trajectory is discovered, and hence no basis functions. Thus, this invocation is seemingly wasted from the point of view of generalization: it does not contribute to reducing the average cost of heuristic computation as described thus far.

As it turns out, we can, in fact, amortize the cost of discovery of implicit dead ends in a way similar to reducing the average time of other states' evaluation. To do so, we use the known dead ends along with stored basis functions to derive the latter's duals in our information-sharing framework, nogoods. We remind the reader that nogoods generalize dead ends in precisely the same way as basis functions do with non-dead ends and therefore help recognize many dead ends without resorting to classical planning. The precise nogood learning mechanism is called SixthSense and is described in Section 5. It needs to be invoked at several points throughout GOTH's running time as prescribed by a scheduler that is also described in that section. For now, we abstract away the operation of SixthSense on lines 32–34 of Algorithm 1. With nogoods available, positively deciding whether a state is a dead end is as simple as checking whether any of the known nogoods subsumes it (lines 8–9 of Algorithm 1). Deterministic planning is necessary to answer the question only if none do.

Speed and memory performance. To facilitate empirical analysis of GOTH, it is helpful to look at the extra speed and memory cost an MDP solver incurs while using it.

Concerning GOTH's memory utilization, we emphasize that, similar to h_{FF} and many other heuristics, GOTH *does not* store any of the states it is given for heuristic evaluation. It merely returns heuristic values of these states to the MDP solver, which can then choose to store the resulting state-value pairs or discard them. However, to compute the values, GOTH needs to memoize the basis functions and nogoods it has extracted so far. As our experiments demonstrate, the set of basis functions and nogoods discovered by GOTH throughout the MDP solver's running time is rather small and is more than compensated for by the reduction in the explored fraction of the state space due to GOTH's informativeness, when compared to h_{FF}.

Timewise, GOTH's performance is largely dictated by the speed of the employed deterministic planner(s) and the number of times it is invoked. Another component that may become significant is determining the "cheapest" basis function that holds in a state (line 11 of Algorithm 1), as it requires iterating, on average, over a constant fraction of known basis functions. Although faster solutions are possible for this pattern-matching problem, all that we are aware of (e.g., [16]) pay for the increase in speed with degraded memory performance.

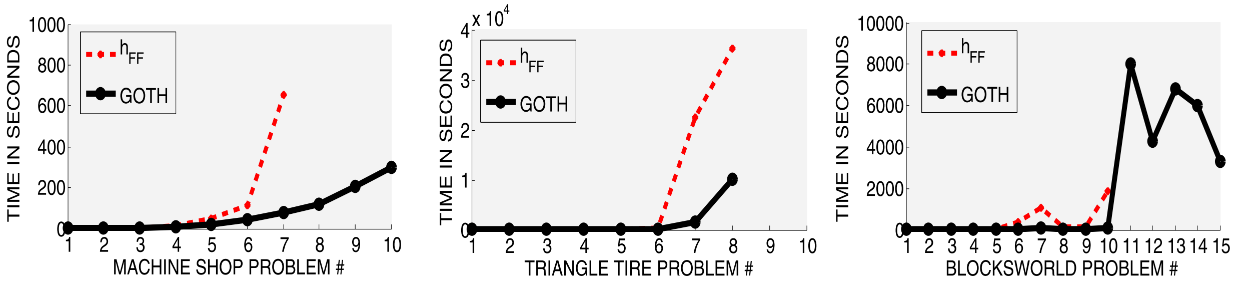


Fig. 3. GOTH outperforms h_{FF} on Machine Shop, Triangle Tireworld, and Blocksworld in speed by a large margin.

Theoretical properties. Two especially important theoretical properties of GOTH are the informativeness of its estimates and its inadmissibility. The former ensures that, compared to h_{FF}, GOTH causes MDP solvers to explore fewer states. At the same time, like h_{FF}, GOTH is inadmissible. One source of inadmissibility comes from the general lack of optimality of deterministic planners. Even if they were optimal, however, employing timeouts to terminate the classical planner occasionally causes GOTH to falsely assume states to be dead ends. Finally, the basis function generalization mechanism also contributes to inadmissibility. The set of discovered basis functions is almost never complete, and hence even the smallest basis function weight known so far may be an overestimate of the state's true value, as there may exist an even cheaper goal trajectory from this state that GOTH is unaware of. In spite of theoretical inadmissibility, in practice using GOTH usually yields very good policies whose quality is often better than of those found under the guidance of h_{FF}.

3.3. *Experimental results*

Our experiments compare the performance of a probabilistic planner using GOTH to that of the same planner under the guidance of h_{FF} across a wide range of domains. In our experience, h_{FF}, included as a part of miniGPT [8], outperforms all other well-known MDP heuristics on most IPPC domains, e.g., the min-min and atom-min heuristics supplied in the same package. Our implementation of GOTH uses a portfolio of two classical planners, FF and LPG. To evaluate a state, it launches both planners as in line 12 of Algorithm 1 in parallel and takes the heuristic value from the one that returns sooner. The timeout for each deterministic planner for finding a plan from a given state to a goal was 25 seconds. We tested GOTH and h_{FF} as part of the LRTDP planner available in the miniGPT package. Our benchmarks were six probabilistic domains, five of which come from the two most recent IPPCs with goal-oriented problems: Machine Shop [37], Triangle Tireworld (IPPC-08), Exploding Blocks World (IPPC-08 version), Blocks World (IPPC-06 version), Elevators (IPPC-06), and Drive (IPPC-06). All of the remaining domains from IPPC-06 and IPPC-08 are either easier versions of the above (e.g., Tireworld from IPPC-06) or have features not supported by our implementation of LRTDP (e.g., rewards, universal quantification, etc.) so we were not able to test on them. Additionally, we perform a brief comparison of LRTDP + GOTH against FFReplan, since it shares some insights with GOTH. In all experiments except measuring the effect of generalization, the planners had a 24-hour limit to solve each problem. All experiments for GOTH, as well as those for ReTrASE and SixthSense, described in Sections 4.3 and 5.3 respectively, were performed on a dual-core 2.8 GHz Intel Xeon processor with 2 GB of RAM.

Comparison against h_{FF}. In this subsection, we use each of the domains to illustrate various aspects and modes of GOTH's behavior and compare it to the behavior of h_{FF}. As shown below, on five of the six test domains LRTDP + GOTH substantially outperforms LRTDP + h_{FF}.

We start the comparison by looking at a domain whose structure is especially inconvenient for h_{FF}, Machine Shop. Problems in this set involve two machines and a number of objects equal to the ordinal of the corresponding problem. Each object needs to go through a series of manipulations, of which each machine is able to do only a subset. The effects of some manipulations may cancel the effects of others (e.g., shaping an object destroys the paint sprayed on it). Thus, the order of actions in a plan is critical. This domain illuminates the drawbacks of h_{FF}, which ignores delete effects and does not distinguish good and bad action sequences as a result. Machine Shop has no dead ends.

Figs. 3 and 4 show the speed and memory performance of LRTDP equipped with the two heuristics on problems from MachineShop (and two other domains) that at least one these planners could solve without running out of memory. As implied by the preceding discussion of GOTH's space requirements, the memory consumption of LRTDP + GOTH is measured by the number of states, basis functions, and nogoods whose values need to be maintained (GOTH caches basis functions and LRTDP caches states). In the case of LRTDP + h_{FF} all memory used is only due to LRTDP's state caching because h_{FF} by itself does not memoize anything. On Machine Shop, the edge of LRTDP + GOTH is clearly vast, reaching several orders of magnitude. In fact, LRTDP + h_{FF} runs out of memory on the three hardest problems, whereas LRTDP + GOTH is far from that.

Concerning policy quality, we found the use of GOTH to yield optimal or near-optimal policies on Machine Shop. This contrasts with h_{FF} whose policies were on average 30% more costly than the optimal ones.

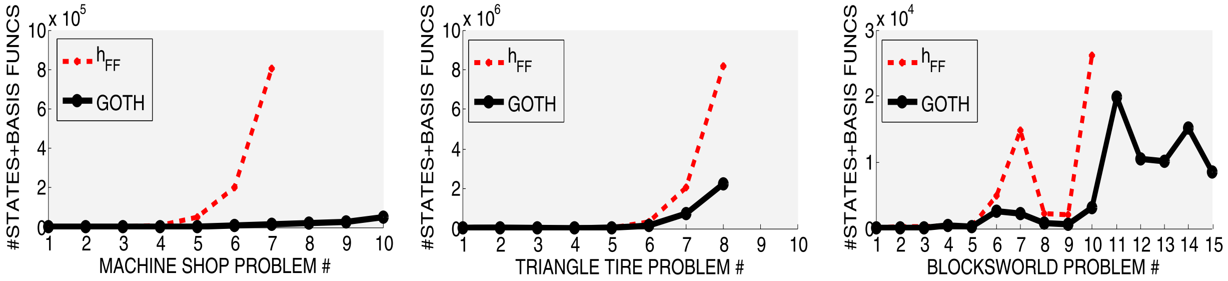


Fig. 4. GOTH's advantage over h_{FF} on Machine Shop, Triangle Tireworld, and Blocksworld in memory is large as well.

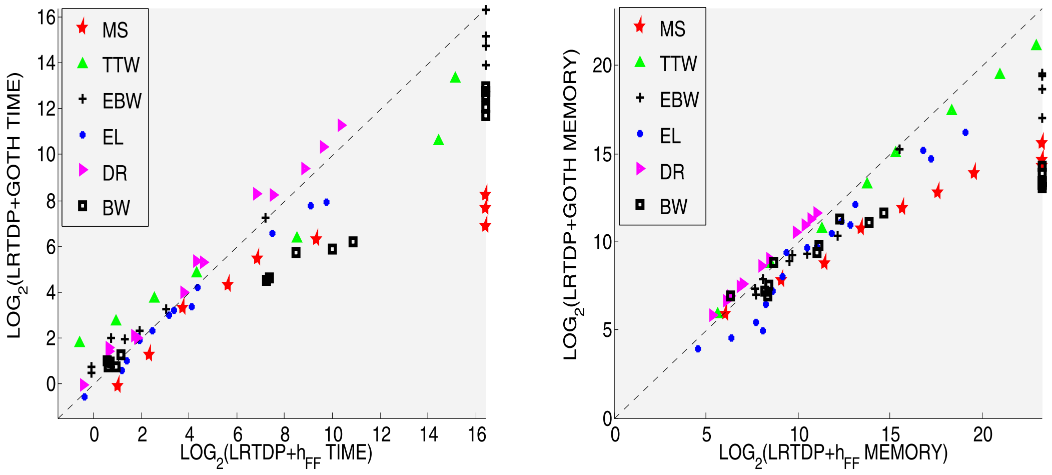


Fig. 5. The big picture: GOTH provides a significant advantage on large problems. (Note that the axes are on a Log scale.)

The Triangle Tireworld domain, unlike Machine Shop, does not have structure that is particularly adversarial for h_{FF}. However, LRTDP + GOTH noticeably outperforms LRTDP + h_{FF} on it too, as Figs. 3 and 4 indicate. Nonetheless, neither heuristic saves enough memory to let LRTDP solve past problem 8. In terms of solution quality, both planners find optimal policies on the problems they can solve.

The results on Exploding Blocks World (EBW, Fig. 5) are similar to those on Triangle Tireworld, where the LRTDP + GOTH's more economical memory consumption eventually translates to a speed advantage. Importantly, however, on several EBW problems LRTDP + GOTH is superior to LRTDP + h_{FF} in a more illustrative way: it manages to solve four problems on which LRTDP + h_{FF} runs out of space. The policy quality of the planners is similar.

The Drive domain is small, and using GOTH on it may not provide significant benefit. On Drive problems, planners spend most of the time in decision-theoretic computation but explore no more than around 2000 states. LRTDP under the guidance of GOTH and h_{FF} explores roughly the same number of states, but since so few of them are explored generalization does not play a big role and GOTH incurs the additional overhead of maintaining the basis functions without getting a significant benefit from them. Perhaps surprisingly, however, GOTH sometimes leads LRTDP to find policies with higher success rates (coverage), while never causing it to find worse policies than h_{FF}. The difference in policy quality reaches 50% on the Drive domain's largest problems. Reasons for this are a topic for future investigation.

On the remaining test domains, Elevators and Blocksworld, LRTDP + GOTH dominates LRTDP + h_{FF} in both speed and memory while providing policies of equal or better quality. Figs. 3 and 4 show the performance on Blocksworld as an example. Classical planners in our portfolio cope with determinized versions of these domains very quickly, and abstraction ensures that the obtained heuristic values are spread over many states. Similar to the situation for EBW, the effectiveness of GOTH is such that LRTDP + GOTH can solve even the five hardest problems of Blocksworld, which LRTDP + h_{FF} could not.

Fig. 5 provides the big picture of the comparison. For each problem we tried, it contains a point whose coordinates are the logarithms of the amount of time/memory that LRTDP + GOTH and LRTDP + h_{FF} took to solve that problem. Thus, points that lie below the $Y = X$ line correspond to problems on which LRTDP + GOTH did better according to the respective criterion. The axes of the time plot of Fig. 5 extend to $\log_2(86\,400)$, the logarithm of the time cutoff (86 400 s, i.e. 24 hours) that we used. Similarly, the axes of the memory plot reach $\log_2(10\,000\,000)$, the number of memoized states/basis functions at which the hash tables where they are stored become too inefficient to allow a problem to be solved within the 86 400 s

Table 1
Average ratio of the number of states memoized by LRTDP under the guidance of h_{FF} to the number under GOTH across each test domain. The bigger these numbers, the more memory GOTH saves the MDP solver compared to h_{FF}.

EBW	EL	TTW	DR	MS	BW
2.07	4.18	1.71	1.00	14.40	7.72

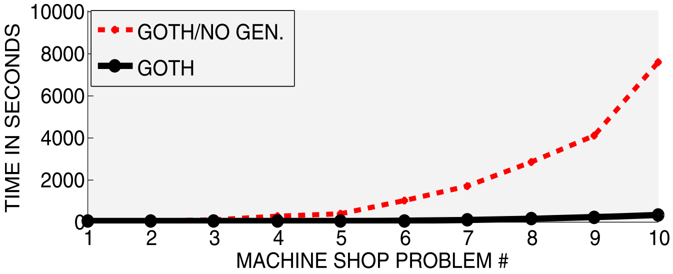


Fig. 6. GOTH is much faster with generalization than without.

time limit. Thus, the points that lie on the extreme right or top of these plots denote problems that could not be solved under the guidance of at least one of the two heuristics. Overall, the time plot shows that, while GOTH ties or is slightly beaten by h_{FF} on Drive and smaller problems of other domains, it enjoys a comfortable advantage on most large problems. In terms of memory, this advantage extends to most medium-sized and small problems as well, and sometimes translates into a qualitative difference, allowing GOTH to handle problems that h_{FF} cannot.

Why does GOTH's and h_{FF}'s comparative performance differ from domain to domain? For insight, refer to Table 1. It displays the ratio of the number of states explored by LRTDP + h_{FF} to the number explored by LRTDP + GOTH, averaged over the problems that could be solved by both planners in each domain. Thus, these numbers reflect the relative informativeness of the heuristics. Note the important difference between the data in this chart and memory usage as presented on the graphs: the information in the table disregards memory consumption due to the heuristics, thereby separating the description of heuristics' informativeness from a characterization of their efficiency. Associating the data in the table with the relative speeds of LRTDP + h_{FF} and LRTDP + GOTH on the test domains reveals a clear trend; the size of LRTDP + GOTH's speed advantage is strongly correlated with its memory advantage, and hence with its advantage in informativeness. In particular, GOTH's superiority in informativeness is not always sufficient to compensate for its computation cost. Indeed, the 1.71× average reduction (compared to h_{FF}) in the number of explored states on Triangle Tireworld is barely enough to make good the time spent on deterministic planning (even with generalization). In contrast, on domains like Blocksworld, where GOTH causes LRTDP to visit many times fewer states than h_{FF}, LRTDP + GOTH consistently solves the problems much faster.

Benefit of generalization. Our main hypothesis regarding GOTH has been that generalization is vital for making GOTH computationally feasible. To test it and measure the importance of basis functions and nogoods for GOTH's operation, we ran a version of GOTH with generalization turned off on several domains, i.e. with the classical planner being invoked from every state passed to GOTH for evaluation. (As an aside, note that this is akin to the strategy of FFReplan, with the fundamental difference that GOTH's state values are eventually overridden by the decision-theoretic training process of LRTDP. We explore the relationship between FFReplan and GOTH further in the next subsection.)

As expected, GOTH without generalization proved to be vastly slower than full GOTH. For instance, on Machine Shop LRTDP + GOTH with generalization turned off is approximately 30–40 times slower (Fig. 6) by problem 10, and the gap is growing at an alarming rate, implying that without our generalization technique the speedup over h_{FF} would not have been possible at all. On domains with implicit dead ends, e.g. Exploding Blocks World, the difference is even more dramatic, reaching over two orders of magnitude.

Furthermore, at least on the relatively small problems on which we managed to run LRTDP + GOTH without generalization, we found the quality of policies (measured by the average plan length) yielded by generalized GOTH to be typically *better* than with generalization off. This result is somewhat unexpected, since generalization is an additional layer of approximation on top of determinizing the domain. We attribute this phenomenon to our averaging weight update strategy. As pointed out earlier, the weight of a basis function (i.e., the length of a plan, in the case of non-generalized GOTH) from any single classical planner invocation may not be reflective of the basis function's quality, and non-generalized GOTH will suffer from such noise more than regular GOTH. In any event, even if GOTH without generalization yielded better policies, its slowness would make its use unjustifiable in practice.

One may wonder whether generalization can also benefit h_{FF} the way it helped GOTH. While we have not conducted experiments to verify this, we believe the answer is no. Unlike full deterministic plan construction, finding a relaxed plan sought by h_{FF} is much easier and faster. Considering that the generalization mechanism involves iterating over many of the available basis functions to evaluate a state, any savings that may result from avoiding h_{FF}'s relaxed plan computation will be negated by this iteration.

Computational profile. An interesting aspect of GOTH's modus operandi is the fraction of the computational resources an MDP solver uses that is due to GOTH. E.g., across the Machine Shop domain, LRTDP + GOTH spends 75–90% of the time in heuristic computation, whereas LRTDP + h_{FF} only 8–17%. Thus, GOTH is computationally much heavier but causes LRTDP to spend drastically less time exploring the state space.

Comparison against FFReplan. One can find similarities between the techniques used by GOTH and FFReplan. Indeed, both employ deterministic planners, FFReplan — for action selection directly, while GOTH — for state evaluation. One key difference again lies in the fact that GOTH is not a complete planner, and lets a dedicated MDP solver correct its judgment. As a consequence, even though GOTH per se ignores probabilistic information in the domain, probabilities are (or can be) nonetheless taken into account during the solver's search for a policy. FFReplan, on the other hand, ignores them entirely. Due to this discrepancy, performance of FFReplan and a planner guided by GOTH is typically vastly distinct. For instance, FFReplan is faster than most decision-theoretic planners. On the other hand, FFReplan has difficulty dealing with probabilistic subtleties. It is known to come up with very low success rate policies on probabilistically interesting problems, e.g., on almost all problems of Triangle Tireworld'06 [35]. LRTDP + GOTH can handle such domains much better. E.g., as stated above, it produces optimal, 100% success-rate policies on the first eight out of ten problems of the even harder version of Triangle Tireworld that appeared at IPPC'08.

3.4. Summary

GOTH is a heuristic function that provides an MDP solver with informative state value estimates using costs of plans in the deterministic version of the given MDP. Computing such plans is expensive. To amortize the time spent on their computation, GOTH employs basis functions, which generalize the cost of one plan to many states. As the experiments show, this strategy and the informativeness of state value estimates make GOTH into a more effective heuristic than the state of the art, h_{FF}.

4. ReTrASE

4.1. Motivation

In GOTH, the role of information transfer via basis functions and nogoods was primarily to reuse computation in the form of classical planner invocations and thus save time. In this section, we present an MDP solver called ReTrASE, **Re**gressing **Tr**ajectories for **A**pproximate **S**tate **E**valuation, initially described in [31] that employs basis functions in a similar way but this time chiefly for the purpose of drastically reducing the memory footprint.

Many dynamic programming-based MDP algorithms, e.g. VI and (L)RTDP, suffer from the same critical drawback — they represent the state value function extensionally, i.e., as a table, thus requiring memory (and time) exponential in the number of MDP variables. Since this extensional representation grows very 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 IPPC.

Two broad approaches have been proposed for avoiding creation of a state/value table. One method consists in computing the policy online with the help of a domain determinization, such as the all-outcomes one. In online settings, the policy needs to be decided on-demand, only for the current state at each time step. This makes maintaining a state-value table unnecessary (although potentially useful). Running a classical planner on a domain determinization helps choose an action in the current state without resorting to this table. Determinization-based planners, e.g., FFHop [43], are often either slow due to invoking a classical planner many times or, as in the case of FFReplan, disregard the probabilistic nature of actions and have trouble with *probabilistically interesting* [35] domains, in which short plans have a low probability mass.

The other method, *dimensionality reduction*, maps the MDP 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 by manually defining a domain-specific basis function set in which basis functions captured some human intuition about the domain at hand. 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" [22,23,20]. 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 in nominal domains. The focus of section is an automatic procedure for doing so.

To our knowledge, there has been little work on mating decision theory, determinization, and dimensionality reduction. With the ReTrASE algorithm, we are bridging the gap, proposing a fusion of these ideas that removes the drawbacks of each. ReTrASE learns a compact value function approximation successful in a range of nominal domains. Like GOTH, it

Algorithm 2 ReTrASE.

```
Input: probabilistic domain $D$, problem $P = \langle$init. state $s_0$, goal $G\rangle$, trial length $L$, determinization routine $Det$, classical planner $DetPlan$, timeout $T$
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., 1 000 000
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 ModifiedBellmanBackup(action $a$, state $s$)
for all basis functions $f$ in $s$ that enable $a$ do
    $M[f] \leftarrow ExpActCost(a, s)$
end for
```

does so by first obtaining a set of basis functions automatically by planning in a determinized version of the domain at hand. However, being a full probabilistic planner, unlike GOTH, it also *learns* the weights for these basis functions by the decision-theoretic means and aggregates them to compute state values as other dimensionality-reduction methods do. Thus, as opposed to GOTH, ReTrASE tries to incorporate the probabilistic information lost at the determinization stage back into the solution. 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, while the implicit reuse of classical plans thanks to basis functions makes it fast.

We demonstrate the practicality of ReTrASE 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.

4.2. ReTrASE *description*

The main intuition underlying ReTrASE is that extracting basis functions in an MDP is akin to mapping the MDP to a lower-dimensional parameter space. In practice, this space is much smaller than the original state space, since only the relevant causal structure is retained,[3] giving us large reduction in space requirements. Solving this new problem amounts to learning weights, a quantitative measure of each basis function's quality. There are many imaginable ways to learn them; in this paper, we explore one such method — a modified version of RTDP.

The weights reflect the fact that basis functions differ in the total expected cost of goal trajectories they enable as well as in the total probability of these trajectories. At this point, we stress that ReTrASE makes two approximations on its way to computing an MDP's value function, and the first of them is related to the semantics of basis function weights and importance. Any given basis function enables only some subset T of the goal trajectories in a given state, and is oblivious to all other trajectories in that state. The other trajectories may or may not be preferable to the ones in T (e.g., because the former may lead the agent to the goal with 100% probability). Therefore, the importance of the trajectories (and hence of corresponding basis functions!) depends on the state. Our intuitive notion of weights ignores this subtlety, since the weight

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

Algorithm 3 Generating basis functions.

```
1: Input: probabilistic domain $D$, problem $P = \langle$init. state $s_0$, goal $G\rangle$, determinization routine $Det$, classical planner $DetPlan$, timeout $T$
2: declare global map $M$ from basis functions to weights
3: declare global set $DE$ of dead ends
4: compute global determinization $D_d$ of $D$
5:
6: function GetBasisFuncsForS(state $s$)
7: declare problem $P_s \leftarrow \langle$init. state $s$, goal $G\rangle$
8: declare plan $pl \leftarrow DetPlan(D_d, P_s, T)$
9: if $pl == none$ then
10:     insert $s$ into $DE$
11: else
12:     declare basis function $f \leftarrow$ goal $G$
13:     declare $cost \leftarrow 0$
14:     for all $i = length(pl)$ through 1 do
15:         declare action $a \leftarrow pl[i]$
16:         $cost \leftarrow cost + cost(a)$
17:         $f \leftarrow (f \cup precond(a)) - effect(a)$
18:         insert $\langle f, cost\rangle$ into $M$ if $f$ is not in $M$ yet
19:     end for
20: end if
```

of a basis function does not vary with states in which this basis function holds. Thus, a weight is in effect the reflection of an "average" importance of a basis function across the states it represents.

The above details notwithstanding, the differences among basis function weights exist 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 does not 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 basis functions than others.

Now, given that each state is generally represented by several basis functions, what is the connection between the state's value and their weights? In general, the relationship is quite complex: under the optimal policy, trajectories enabled by several basis functions may be possible, causing some trajectories to factor into weights of several basis functions simultaneously. However, determining the subset of basis functions enabling these trajectories is at least as hard as solving the MDP exactly. Instead, we approximate the state value by the *minimum* weight of all basis function that represent the state. This amounts to saying that the "better" a state's "best" basis function is, the "better" is the state itself, and is the second approximation ReTrASE makes.

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

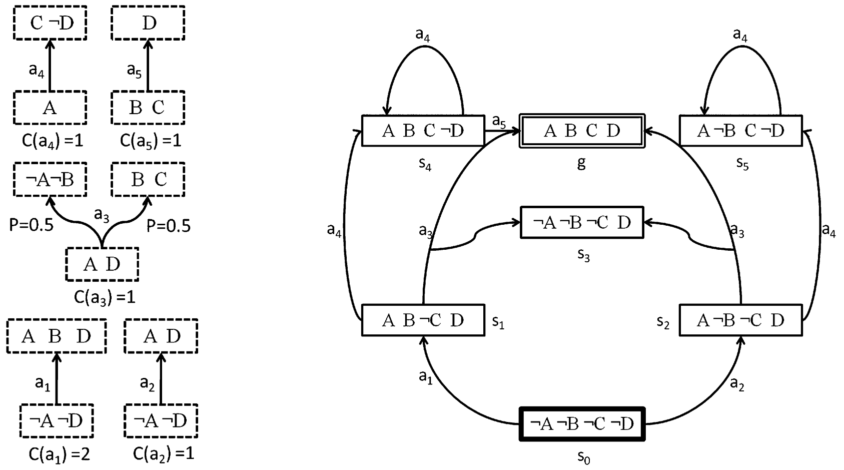
Algorithm's operation. For a step-by-step example of operation of ReTrASE, whose pseudo code is presented in Algorithm 2, please refer to the proof of Theorem 1. ReTrASE starts by computing the determinization D_d of the domain. As in GOTH, we use D_d to rapidly compute the basis functions. The algorithm explores the state space by running modified RTDP trials, memoizing all the dead ends and basis functions it learns along the way. Whenever during state evaluation (line 21) ReTrASE finds a state that is neither a known dead-end nor has any basis function that holds in it, ReTrASE uses the regression procedure **GetBasisFuncsForS(.)** presented in Algorithm 3 to generate a basis function for it. 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. As in GOTH, if the deterministic planner can prove the non-existence of a plan or simply cannot find a plan before some timeout, the state in question is deemed to be a dead end (line 10 of Algorithm 3).

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 33). The new weight of each such basis function becomes the expected cost of action a'. The intuitive reason for updating the basis functions enabling a' is that a' can be executed in any state where basis functions hold; hence, the quality of a' should be reflected in these basis functions' weights. Analogously, other a' cannot be executed wherever basis functions that do not enable it hold, so the expected cost of those actions is irrelevant to determining their weights.

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

Theorem 1. *There are problems on which* ReTrASE *may not converge.*

Proof. By failing to converge we mean that, on some problems, depending on the order in which basis functions are discovered ReTrASE may indefinitely oscillate over a set of several policies with different expected costs. One such MDP M is presented in Fig. 7, which shows M's transition graph and action set. Solving M amounts to finding a policy of minimum expected cost that takes the agent from state s_0 to state g and uses actions $a_1 - a_5$. The optimal solution to M is a linear plan $s_0 - a_1 - s_1 - a_4 - s_4 - a_5 - g$.



(a) Actions (b) Transition graph

Fig. 7. An example MDP on which ReTrASE fails to converge.

To see that ReTrASE fails to converge on M, we simulate ReTrASE's operation on this MDP. Recall that ReTrASE executes a series of trials, all originating at s_0. □

Trial 1. To choose an action in s_0, ReTrASE needs to evaluate states s_1 and s_2. It does not yet have any basis functions to do that, so it uses the procedure in Algorithm 3 to generate them, together with initial estimates for their weights.

Suppose the procedure first looks for a basis function for s_1 and finds plan $s_1 - a_4 - s_4 - a_5 - g$. Regressing it yields the following basis function-weight pairs: $W(A \wedge B \wedge C \wedge D) = 0$, $W(A \wedge B \wedge C) = 1$, $W(A \wedge B) = 2$. $A \wedge B$ is the only basis functions that holds in s_1 so far. Therefore, the current estimate for the value of s_1, $V(s_1)$, is 2. Accordingly, the current estimate for the value of action a_1 in s_0, Q-value(s_0, a_1), becomes $C(a_1) + V(s_1) = 4$.

Next, suppose that for state s_2, *GetBasisFuncsForS* finds plan $s_2 - a_3 - g$. Regressing it yields one basis function-weight pair in addition to the already discovered ones, $W(A \wedge D) = 1$. Function $A \wedge D$ is the only one that holds in s_2, so we get $V(s_2) = 1$ and Q-value$(s_0, a_2) = 2$.

Now ReTrASE can choose an action in s_0. Since at the moment Q-value$(s_0, a_1) >$ Q-value(s_0, a_2), it picks a_2 and executes it, transitioning to s_2.

In s_2, ReTrASE again needs to evaluate two actions, a_3 and a_4. Notice that a_4 leads to s_5, which is a dead end. *GetBasisFuncsForS* discovers this fact by failing to produce any basis functions. Thus, $V(s_5)$ is a very large dead-end penalty, e.g. 1 000 000, yielding Q-value$(s_2, a_4) = 1\,000\,001$. However, a_3 may also lead to dead end s_3 with $P = 0.5$, so Q-value$(s_2, a_3) = 500\,001$. Nonetheless, a_3 is more preferable, so this is the action that ReTrASE picks in s_2.

At this time, ReTrASE performs a modified Bellman backup in s_2. The only known basis function that holds in s_2 and enables the chosen action a_3 is $A \wedge D$. Therefore, ReTrASE sets $W(A \wedge D) =$ Q-value$(s_2, a_3) = 500\,001$.

Executing a_3 in s_2 completes the trial with a transition either to goal g or to dead end s_3.

Trial 2. This time, ReTrASE can select an action in s_0 without resorting to regression. Currently, $V(s_1) = 2$, since $A \wedge B$ with $W(A \wedge B) = 2$ is the minimum-weight basis function in s_1. However, $V(s_2) = 500\,001$ due to the backup performed during trial 1. Therefore, Q-value$(s_0, a_1) = 4$ but Q-value$(s_0, a_2) = 500\,002$, making a_1 look more attractive. So, ReTrASE chooses a_1, causing a transition to s_1.

In s_1, the choice is between a_3 and a_4. The values of both are easily calculated with known basis functions, Q-value$(s_1, a_3) = 500\,001$ and Q-value$(s_1, a_4) = 2$.

The natural choice is a_4, and ReTrASE performs the corresponding backup. The basis functions enabling a_4 in s_1 are $A \wedge B$ and $A \wedge D$. Their weights become Q-value$(s_1, a_4) = 2$ after the update.

The rest of the trial does not change any weights and is irrelevant to the proof.

Trial n. Crucially, basis function $A \wedge D$, whose weight changed in the previous trials, holds both in state s_1 and in state s_2. Due to the update in s_2 during trial 1, $W(A \wedge D)$ became large and made s_1 look beneficial. On the other hand, thanks to the update in s_1 during trial 2, $W(A \wedge D)$ became small and made s_2 look beneficial. It is easy to see that this cycle will continue in subsequent trials. As a result, ReTrASE will keep on switching between two policies, one of which is suboptimal. □

Overall, the classes of problems on which ReTrASE may diverge are hard to characterize generally. Predicting whether ReTrASE may diverge on a particular problem is an area for future work. We maintain, however, that a lack of theoretical

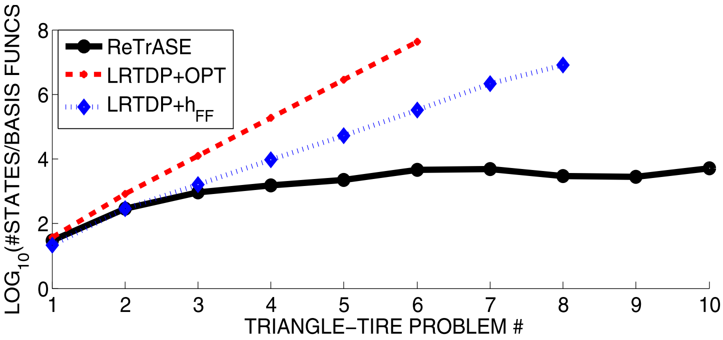


Fig. 8. Memory usage on logarithmic scale: ReTrASE is dramatically more efficient than both LRTDP + OPT and LRTDP + h_{FF}.

guarantees is not indicative of a planner's practical performance. Indeed, several IPPC winners, including FFReplan, have a weak theoretical profile. The experimental results show that ReTrASE too performs very well on many of the planning community's benchmark problems.

4.3. Experimental results

Our goal in this subsection 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 and non-determinization-based approximate 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.

Implementation details. ReTrASE is implemented in C++ and uses miniGPT [8] 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.).

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. Since our implementation does not yet support such PPDDL features as universal quantification, 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 inadmissible h_{FF} (LRTDP + h_{FF}), and LRTDP + OPT — LRTDP with Atom-Min-1-Forward|Min-Min heuristic [8]. 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.

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.

Comparing scalability. We begin by showcasing the memory savings of ReTrASE over LRTDP + OPT and LRTDP + h_{FF} on the Triangle Tire World domain. Fig. 8 demonstrates the savings of ReTrASE to increase dramatically with problem size. In fact, neither LRTDP variant is able to solve past problem 8 as both run out of memory, whereas ReTrASE copes with all ten problems. Scalability comparisons for other domains we tested on yield generally similar results.

Other popular approximate algorithms (aside from LRTDP + h_{FF}) do not 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.

Comparing solution quality: Success rate. Continuing with the Triangle Tire World domain, we compare the success rates of ReTrASE, RFF [41] — the overall winner of IPPC-08, and HMDPP [29] — the winner on this particular domain. Note that Triangle Tire World, perhaps, the most famous probabilistically interesting domain, was designed largely to confound

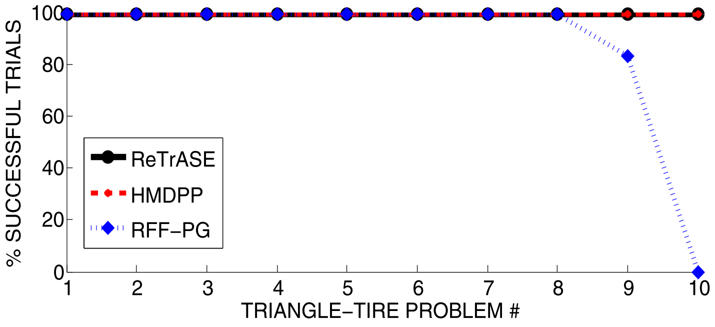


Fig. 9. ReTrASE achieves perfect success rate on Triangle Tire World-08.

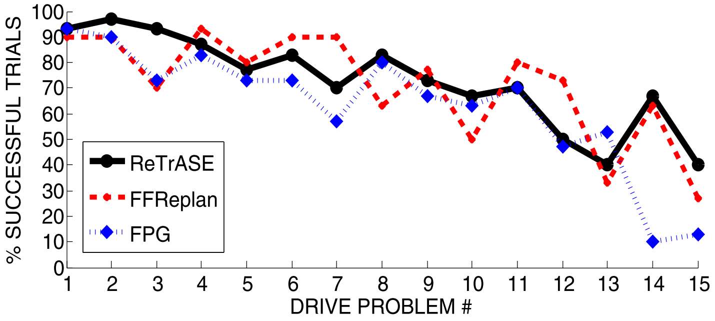


Fig. 10. ReTrASE is at par with the competitors on Drive.

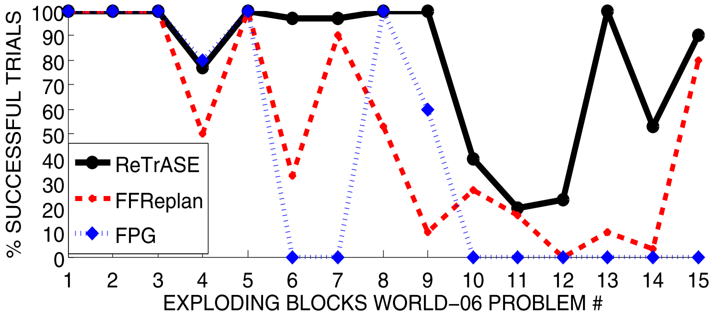


Fig. 11. ReTrASE dominates on Exploding Blocks World-06.

solvers that rely on domain determinization [35], e.g., FFReplan; therefore, performance on it is particularly important for evaluating a new planner. Indeed, as Fig. 9 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 (Fig. 10). 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 (Fig. 11), 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 (Fig. 12), 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 the Elevators problems ReTrASE did not converge within the allocated 40 minutes and was outperformed by several planners. We suspect this is due to bad luck ReTrASE has with basis functions in this domain. However, on TTW-06 ReTrASE was the winner on every problem.

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

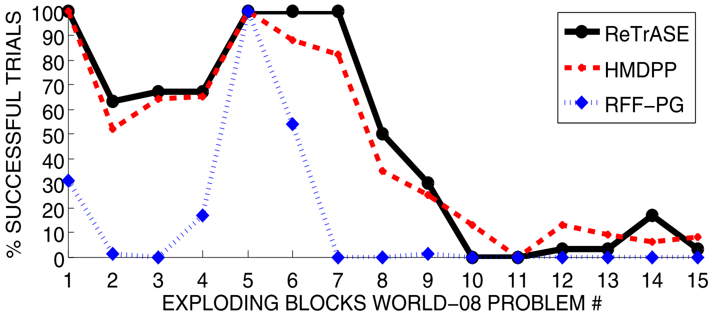


Fig. 12. ReTrASE outmatches all competitors on Exploding Blocks World-08, although by a narrow margin.

Table 2
Success rates on some IPPC-04 problems.

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

planner we used, LRTDP + OPT, scales enough to solve only relatively small problems (at most a few million states). On such problems we found ReTrASE to produce trajectories of expected cost within 5% of the optimal.

Comparison to 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 [43], and demonstrated its capabilities on problems from IPPC-04. Unfortunately, due to the current lack of support for some PPDDL language features we were not able to run ReTrASE on most IPPC-04 domains. Table 2 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 FFHop inconclusive.

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.

4.4. Summary

ReTrASE is an MDP solver based on a combination of state abstraction and dimensionality reduction. It automatically extracts basis functions, which provide a compact representation of the given MDP while retaining its causal structure. Simultaneously with discovering basis functions, it learns weights for the already discovered ones using modified Bellman backups. These weights let ReTrASE evaluate states without memoizing state values explicitly. Such an approach allows ReTrASE to solve larger problems than the best performers of several recent IPPCs.

5. SixthSense

5.1. Motivation

Although basis functions efficiently generalize information about states from which reaching the goal is possible, they have nothing to say about dead ends. As a result, algorithms that use only basis functions for information transfer cannot avoid either caching dead ends or rediscovering them every time they run into them. In fact, the issue of quickly and reliably recognizing dead ends plagues virtually all modern MDP solvers. For instance, in IPPC-2008 [9], the domains with a complex dead-end structure, e.g., Exploding Blocks World, have proven to be the most challenging. Surprisingly, however, there has been little research on methods for effective discovery and avoidance of dead ends in MDPs. Of the two types of dead ends, implicit ones confound planners the most, since they do have executable actions. However, explicit dead ends can be a resource drain as well, since verifying that none of the available actions are applicable in a state can be costly if the number of actions is large.

Broadly speaking, existing planners use one of two approaches for identifying dead ends. When faced with a yet-unvisited state, many planners (e.g., LRTDP) apply a heuristic value function (such as h_{FF}), which hopefully assigns a high cost to dead-end states. This method is fast to invoke but often fails to catch many implicit dead ends due to the problem relaxation inevitably used by the heuristics. Failure to detect them causes the planner to waste much time in exploring the states reachable from implicit dead ends, these states being dead ends themselves. Other MDP solvers use state value estimation approaches that recognize dead ends reliably but are very expensive; for example, RFF, HMDPP, and ReTrASE employ full deterministic planners. When a problem contains many dead ends, these MDP solvers may spend a lot of their time launching classical planners from dead ends. Indeed, most probabilistic planners would run faster if recognizing dead ends was not so computationally expensive.

In this section, we complete our abstraction framework by presenting a novel mechanism, SixthSense, to do exactly this — quickly and reliably identify dead-end states in MDPs. Underlying SixthSense, pioneered in [33], is a key insight: large sets of dead-end states can usually be characterized by a compact logical conjunction, a *nogood*, which "explains" why no solution exists. For example, a Mars rover that flipped upside down will be unable to achieve its goal, regardless of its location, the orientation of its wheels, etc. Knowing this explanation lets a planner quickly recognize millions of states as dead ends. Crucially, dead ends in most MDPs can be described with a small number of nogoods.

SixthSense learns nogoods by generating candidates with a bottom-up greedy search (resembling that used in rule induction [12]) and tests them to avoid false positives with a planning graph-based procedure. A vital input to this learning algorithm are basis functions, derived as shown in the previous sections. SixthSense is provably sound — every nogood output represents a set of true dead ends. We empirically demonstrate that SixthSense speeds up two different types of MDP solvers on several IPPC domains with implicit dead ends and show the performance improvements SixthSense gives to GOTH and ReTrASE. Overall, SixthSense tends to identify most of the dead ends that the solvers encounter, reducing memory consumption by as much as 90%. Because SixthSense runs quickly, it also gives a 30–50% speedup on large problems. With these savings, it enables planners to solve problems they could not previously handle.

5.2. SixthSense *description*

An MDP may have an exponential number of dead end states, but often there are just a few "explanations" for why a state has no goal trajectory. A Mars rover flipped upside down is in a dead-end state, irrespective of the values of the other variables. In the Drive domain of IPPC-06, all states with the (*not* (*alive*)) literal are dead ends. Knowing these explanations obviates the dead-end analysis of each state individually and the need to store the explained dead ends in order to identify them later.

Our method, SixthSense, strives to induce explanations as above in the factored MDP setting and use them to help the planner recognize dead ends quickly and reliably. Formally, its objective is to find nogoods, conjunctions of literals with the property that all states in which such a conjunction holds are dead ends. After at least one nogood is discovered, whenever the planner encounters a new state, SixthSense notifies the planner if the state is represented by a known nogood and hence is a dead end.

To discover nogoods, we devise a machine learning generate-and-test algorithm that is an integral part of SixthSense. The "generate" step proposes a *candidate* conjunction, using some of the dead ends the planner has found so far as training data. For the testing stage, we develop a novel planning graph-based algorithm that tries to prove that the candidate is indeed a nogood. Nogood discovery happens in several attempts called *generalization rounds*. First we outline the generate-and-test procedure for a single round in more detail and then describe the scheduler that decides when a generalization round is to be invoked. Algorithm 4 contains the learning algorithm's pseudocode.

Generation of candidate nogoods. There are many ways to generate a candidate but if, as we conjecture, the number of explanations/nogoods in a given problem is indeed very small, naive hypotheses, e.g., conjunctions of literals picked uniformly at random, are very unlikely to pass the test stage. Instead, our procedure makes an "educated guess" by employing basis functions according to one crucial observation. Recall that, by definition, basis functions are preconditions for goal trajectories. Therefore, no state represented by them can be a dead end. On the other hand, any state represented by a nogood, by the nogoods' definition, *must* be a dead end. These facts combine into the following observation: *a state may be generalized by a basis function or by a nogood but not both.*

Of more practical importance to us is the corollary that any conjunction that has no conflicting pairs of literals (a literal and its negation) and contains the negation of at least one literal in every basis function (i.e., *defeats* every basis function) is a nogood. This fact provides a guiding principle — form a candidate by going through each basis function in the problem and, if the candidate does not defeat it, picking the negation of one of the basis function's literals. By the end of the run, the candidate provably defeats all basis functions in the problem. The idea has a big drawback though: finding *all* basis functions in the problem is prohibitively expensive. Fortunately, it turns out that making sure the candidate defeats only a few randomly selected basis functions (100–200 for the largest problems we encountered) is enough in practice for the candidate to be a nogood with reasonably high probability (although not for certain, motivating the need for verification). Therefore, before invoking the learning algorithm for the first time, our implementation acquires 100 basis functions by running the classical planner FF. Candidate generation is described on lines 5–11.

So far, we have not specified how exactly defeating literals should be chosen. Here as well we can do better than naive uniform sampling. Intuitively, the frequency of a literal's occurrence in the dead ends that the MDP solver has encountered so far correlates with the likelihood of the literal's presence in nogoods. The algorithm's *sampleDefeatingLiteral* subroutine samples a literal defeating basis function b with a probability proportionate to the literal's frequency in the dead ends represented by the constructed portion of the nogood candidate. The method's strengths are twofold: not only does it take

Algorithm 4 SixthSense.

```
Input: training set of known non-generalized dead ends $setDEs$, set of basis functions $setBFs$, set of nogoods $setNG$, goal $g$, set of all domain literals $setL$

function learnNogood($setDEs$, $setBFs$, $setNGs$, $g$)
// construct a candidate
declare candidate conjunction $c = \{\}$
for all $b \in setBFs$ do
    if $c$ does not defeat $b$ then
        declare literal L = sampleDefeatingLiteral($setDEs$, $b$, $c$)
        $c = c \cup \{L\}$
    end if
end for
// check candidate with planning graph, and prune it
if checkWithPlanningGraph($setL$, $c$, $g$) then
    for all literals $L \in c$ do
        if checkWithPlanningGraph($setL$, $c \setminus \{L\}$, $g$) == $success$ then
            $c = c \setminus \{L\}$
        end if
    end for
else
    return $failure$
end if
// if we got here then the candidate is a valid nogood
empty $setDEs$
add $c$ to $setNG$
return $success$

function checkWithPlanningGraph($setL$, $c$, $g$)
for all literals $G$ in $(g \setminus c)$ do
    declare conjunction $c' = c \cup ((setL \setminus (\neg c)) \setminus \{G\})$
    if PlanningGraph(c') == $success$ then
        return $failure$
    end if
end for
return $success$

function sampleDefeatingLiteral($setDEs$, $b$, $c$)
declare counters $C_{\neg L}$ for all $L \in b \setminus c$
for all $d \in setDEs$ do
    if $c$ generalizes $d$ then
        for all $L \in b$ s.t. $\neg L \in d$ do
            $C_{\neg L}++$
        end for
    end if
end for
return a literal $L'$ sampled according to $P(L') \sim C_{L'}$
```

into account information from the solver's experience but also lets literals' co-occurrence patterns direct creation of the candidate.

Nogood verification. If in the above candidate generation procedure we used the set of *all* basis functions that exist for a given MDP, verifying the resulting candidate would not be necessary. The set of states represented by at least one basis function from this exhaustive set would itself be the exhaustive set of non-dead-end states. Therefore, any generated candidate would only represent dead-end states, and thus would be a true nogood.

However, in general we do not have all possible basis functions at our disposal. Consequently, we need to verify that the candidate created by the algorithm from the available basis functions is indeed a nogood. Let us denote the problem of establishing whether a given conjunction is a nogood as *NOGOOD-DECISION*.

Theorem 2. *NOGOOD-DECISION is PSPACE-complete.*

Proof. First, we show that *NOGOOD-DECISION* ∈ *PSPACE*. To verify that a conjunction is a nogood, we can verify that each state this conjunction represents is a dead end. For each state, such verification is equivalent to establishing plan existence in the all-outcomes determinization of the MDP. This problem is *PSPACE*-complete [11], i.e., is in *PSPACE*. Thus, nogood verification can be broken down into a set of problems in *PSPACE* and is in *PSPACE* itself.

To complete the proof, we point out that the already mentioned problem of establishing deterministic plan existence is an instance of *NOGOOD-DECISION*, providing a trivial reduction to *NOGOOD-DECISION* from a *PSPACE*-complete problem. □

In the light of Theorem 2, we may realistically expect an efficient algorithm for *NOGOOD-DECISION* to be either sound or complete, but not both. A sound algorithm would never conclude that a candidate is a nogood when it is not. A complete one would pronounce a candidate to be a nogood whenever the candidate is in fact a nogood. A key contribution of this paper is a sound algorithm for identifying nogoods. It is based on the observation that all the per-state checks in the naive scheme above can be replaced by only a few whose running time is polynomial in the problem size. Although sound, the operation is incomplete, i.e. may reject some candidates that are in fact nogoods. Nonetheless, this check is effective at identifying nogoods in practice.

To verify a candidate c efficiently, we group all *non-goal* states represented by c into several *superstates of* c. We define a superstate of a candidate c to be a set consisting of c's literals, of the negation of one of the goal literals that are not present in c, and of all literals over all other variables in the domain. As an example, suppose the complete set of literals in our problem is $\{A, \neg A, B, \neg B, C, \neg C, D, \neg D, E, \neg E\}$, the goal is $A \wedge \neg B \wedge E$, and the candidate is $A \wedge C$. Then the superstates our algorithm constructs for this candidate are $\{A, \mathbf{B}, C, D, \neg D, E, \neg E\}$ and $\{A, B, \neg B, C, D, \neg D, \neg \mathbf{E}\}$ (the negation of a goal literal in each superstate is highlighted **in bold**).

The intuition behind this definition of superstates of c is as follows. Every non-goal state s represented by c is "contained" in one of superstates of c in the sense that there is a superstate of c containing all of s's literals. Moreover, if a superstate has no trajectory to the goal, no such trajectory exists for any state contained in the superstate, i.e. these states are all dead ends. Combining these two observations, if *no* goal trajectory exists from *any* superstate of c then *all* the states represented by the candidate are dead ends. By definition, such a candidate is a nogood.

Accordingly, to find out whether the candidate is a nogood, our procedure runs the planning graph algorithm on each of the candidate's superstates using determinized actions. Each instance returns *success* if and only if it can reach the goal literals and resolve all mutexes between them. The initial set of mutexes it feeds to the planning graph are just the mutexes between each literal and its negation.

Theorem 3. *The candidate conjunction is a nogood if each of the planning graph expansions on the superstates either* (a) *fails to achieve all of the goal literals or* (b) *fails to resolve mutexes among any two of the goal literals.*

Proof. Since the planning graph is sound, *failure* on *all* superstate expansions indicates the candidate is a true nogood (lines 27–34). □

Our procedure is incomplete for two reasons. First, since each superstate has more literals than any single state it contains, it may have a goal trajectory that is impossible to execute from any state. Second, the planning graph algorithm is incomplete by itself; it may declare plan existence when no plan actually exists.

At the cost of incompleteness, our algorithm is only polynomial in the problem size. To see this, note that each planning graph expansion from a superstate is polynomial in the number of domain literals, and the number of superstates is polynomial in the number of goal literals.

If the verification test is passed, we try to prune away unnecessary literals (lines 13–18) that may have been included into the candidate during sampling. This analog of Occam's razor strives to reduce the candidate to a *minimal* nogood and often gives us a much more general conjunction than the original one at little extra verification cost. At the conclusion of the pruning stage, compression empties the set of dead ends that served as the training data so that the MDP solver can fill it with new ones. The motivation for this step will become clear once we discuss scheduling of compression invocations.

Scheduling. Since we do not know *a priori* the number of nogoods in the problem, we need to perform several generalization rounds. Optimally deciding when to do that is hard, if not impossible, but we have designed an adaptive scheduling mechanism that works well in practice. It tries to estimate the size of the training set likely sufficient for learning an extra nogood, and invokes learning when that much data has been accumulated. When generalization rounds start failing, the scheduler calls them exponentially less frequently. Thus, very little computation time is wasted after all nogoods that could reasonably be discovered have been discovered. (There are certain kinds of nogoods whose discovery by SixthSense, although possible, is highly improbable. We elaborate on this point in the Discussion section.)

Our algorithm is inspired by the following tradeoff. The sooner a successful round happens, the earlier SixthSense can start using the resulting nogood, saving time and memory. On the other hand, trying too soon, with hardly any training data available, is improbable to succeed. The exact balance is difficult to locate even approximately, but our empirical trials indicate three helpful trends: (1) The learning algorithm is capable of operating successfully with surprisingly little training data, as few as 10 dead ends. The number of basis functions does not play a big role provided there is more than about 100 of them. (2) If a round fails with statistics collected from a given number of dead ends, their number usually needs to be increased drastically. However, because learning is probabilistic, such a failure could also be accidental, so it is justifiable to return to the "bad" training data size occasionally. (3) A typical successful generalization round saves the planner enough time and memory to compensate for many failed ones. These three regularities suggest the following algorithm.

- Initially, the scheduler waits for a small batch of basis functions, *setBFs* in Algorithm 4, and a small number of dead ends, *setDEs*, to be accumulated before invoking the first generalization round. For reasons above, in our implementation used the initial settings of $|setBFs| = 100$ and $|setDEs| = 10$ for all problems.

- After the first round and including it, whenever a round succeeds the scheduler waits for a number of dead ends *unrecognized by known nogoods* equal to half of the previous batch size to arrive before invoking the next round. Decreasing the batch size is usually worth the risk according to observations (2) and (3) and because the round before succeeded. If a round fails, the scheduler waits for the accumulation of twice the previous number of unrecognized dead ends before trying generalization again.

Perhaps unexpectedly, in many cases we have seen very large training sets decrease the probability of learning a nogood. This phenomenon can be explained by training sets of large sizes sometimes containing subcollections of dead ends caused by different nogoods. Consequently, the literal occurrence statistics induced by such a mix make it hard to generate reasonable candidates. This finding has led us to restrict the training batch size (*setDEs* in Algorithm 4) to 10 000. If, due to exponential backoff, the scheduler is forced to wait for the arrival of $n > 10\,000$ dead ends, it skips the first $(n - 10\,000)$ and retains only the latest 10 000 for training. For the same locality considerations, each training set is emptied at the end of each round (line 23).

Theoretical properties. Before presenting the experimental results, we analyze SixthSense's properties. The most important one is that the procedure of identifying dead ends as states in which at least one nogood holds is sound. It follows directly from the nogood's definition.

Importantly, SixthSense puts no bounds on the nogood length, being theoretically capable of discovering any nogood. One may ask: are there any nontrivial bounds on the amount of training data for SixthSense to generate a nogood of a given length with at least a given probability? As the following argument indicates, even if such bounds exist they are likely to be of no use in practice. For SixthSense to generate any given nogood, the training data must contain many dead ends caused by this nogood. However, depending on the structure of the problem, most such dead ends may be unreachable from the initial state. If the planning algorithm that uses SixthSense never explores those parts of the state space (e.g., LRTDP), no amount of *practically collectable* training data will help SixthSense discover some of the nogoods with high probability.

At the same time, we can prove another important property of SixthSense:

Theorem 4. *Once a nogood has been discovered and memoized by* SixthSense, SixthSense *will never discover it again.*

Proof. This fact is a consequence of using only dead ends that are not recognized by known nogoods to construct the training sets, as described in the *Scheduling* subsection, and erasing the training data after each generalization attempt. According to Algorithm 4, each nogood candidate is built up iteratively by sampling literals from a distribution induced by training dead ends that are represented by the constructed portion of the candidate. Also, we know that no training dead end is represented by any known nogood. Therefore, the probability of sampling a known nogood (lines 5–11) is 0. □

Regarding SixthSense's speed, the number of frequently encountered nogoods in any given problem is rather small, which makes identifying dead ends by iterating over the nogoods a very quick procedure. Moreover, a generalization round is polynomial in the training data size, and the training data size is linear in the size of the problem (due to the length of the dead ends and basis functions). We point out, however, that obtaining the training data theoretically takes exponential time. Nevertheless, since training dead ends are identified as a part of the usual planning procedure in most MDP solvers, the only extra work to be done for SixthSense is obtaining a few basis functions. Their required number is so small that in nearly every probabilistic problem, they can be quickly obtained by invoking a speedy deterministic planner from several states. This explains why in practice SixthSense is very fast.

Last but not least, we believe that SixthSense can be incorporated into nearly any existing trial-based factored MDP solver, since, as explained above, the training data SixthSense requires is either available in these solvers and can be cheaply extracted, or can be obtained independently of the solver's operation by invoking a deterministic planner.

5.3. Experimental results

Our goal in the experiments was to explore the benefits SixthSense brings to different types of planners, as well as to gauge the effectiveness of nogoods and the amount of computational resources taken to generate them. We used three IPPC domains as benchmarks: Exploding Blocks World-08 (EBW-08), Exploding Blocks World-06 (EBW-06), and Drive-06. IPPC-06 and -08 contained several more domains with dead ends, but their structure is similar to that of the domains we chose. In all experiments, we restricted each MDP solver to use no more than 2 GB of memory.

Structure of dead ends in IPPC domains. Among the IPPC benchmarks, we found domains with only two types of implicit dead ends. In the Drive domain, which exemplifies the first of them, the agent's goal is to stay alive and reach a destination by driving through a road network with traffic lights. The agent may die trying but, because of the domain formulation, this does not necessarily prevent the car from driving and reaching the destination. Thus, all of the implicit dead ends in the domain are generalized by the singleton conjunction (*not alive*). A few other IPPC domains, e.g., Schedule, resemble Drive in having one or several exclusively single-literal nogoods representing all the dead ends. Such nogoods are typically easy for SixthSense to derive.

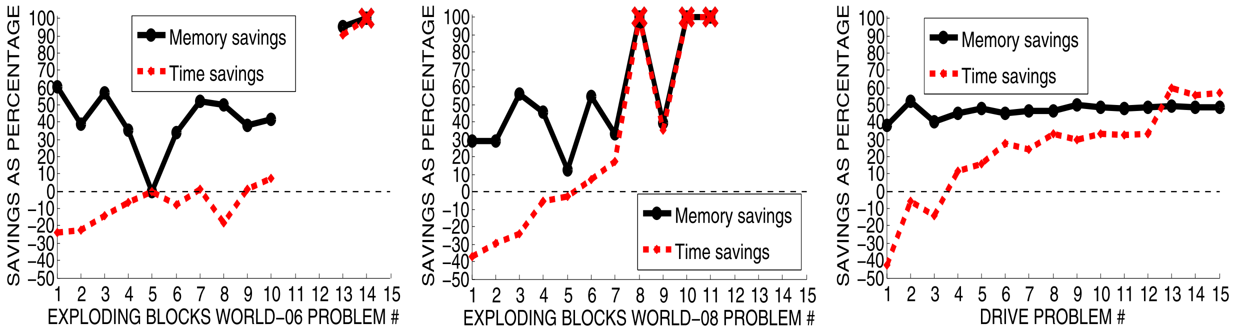


Fig. 13. Time and memory savings due to nogoods for LRTDP + h_{FF} (representing the "Fast but Insensitive" type of planners) on 3 domains, as a percentage of resources needed to solve these problems without SixthSense (higher curves indicate bigger savings; points below zero require more resources with SixthSense). The reduction on large problems can reach over 90% and even enable more problems to be solved (their data points are marked with a ×).

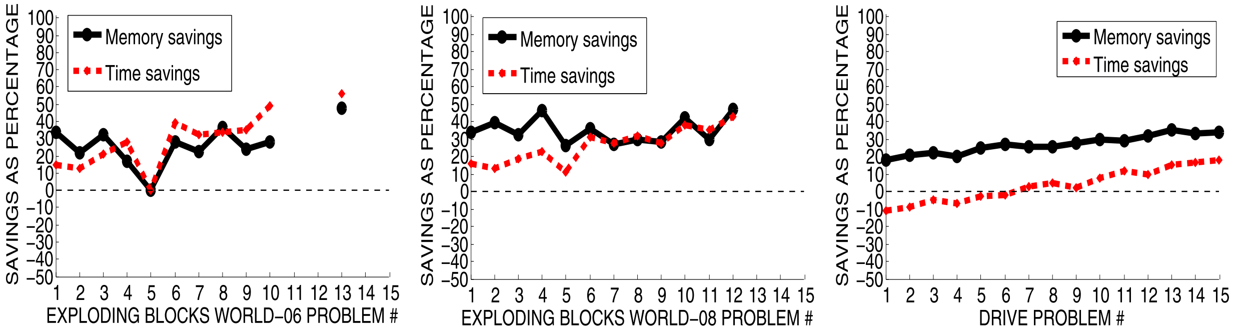


Fig. 14. Resource savings from SixthSense for LRTDP + GOTH/NO 6S (representing the "Sensitive but Slow" type of planners).

EBW-06 and -08's dead ends are much more complex, and their structure is unique among the IPPC domains. In the EBW domain, the objective is to rearrange a number of blocks from one configuration to another, and each block might explode in the process. For each goal literal, EBW has two multiple-literal nogoods explaining when this literal cannot be achieved. For example, if block *b*4 needs to be on block *b*8 in the goal configuration then any state in which *b*4 or *b*8 explodes before being picked up by the manipulator is a dead end, represented either by nogood (*not* (*no-destroyed b*4)) ∧ (*not* (*holding b*4)) ∧ (*not* (*on b*4 *b*8)) or by (*not* (*no-destroyed b*8)) ∧ (*not* (*on b*4 *b*8)). We call such nogoods *immediate* and point out that EBW also has other types of nogoods, described in the Discussion section. The variety and structural complexity of EBW nogoods makes them challenging to learn.

Planners. As pointed out earlier, MDP solvers can be divided into two groups according to the way they handle dead ends. Some of them identify dead ends using fast but unreliable means like heuristics, which miss a lot of dead ends, causing the planner to waste time and memory exploring useless parts of the state space. We will call such planners "fast but insensitive" with respect to dead ends. Most others use more accurate but also more expensive dead-end identification means. We term these planners "sensitive but slow" in their treatment of dead ends. The monikers for both types apply only to the way these solvers handle dead ends and not to their overall performance. With this in mind, we demonstrate the effects SixthSense has on each type.

Benefits to fast but insensitive. This group of planners is represented in our experiments by LRTDP with the h_{FF} heuristic. We will call this combination LRTDP + h_{FF}, and LRTDP + h_{FF} equipped with SixthSense — LRTDP + h_{FF} + 6S for short. Implementationwise, SixthSense is incorporated into h_{FF}. When evaluating a newly encountered state, h_{FF} first consults the available nogoods produced by SixthSense. Only when the state fails to match any nogood does h_{FF} resort to its traditional means of estimating the state value. Without SixthSense, h_{FF} misses many dead ends, since it ignores actions' delete effects.

Fig. 13 shows the time and memory savings due to SixthSense across three domains as the percentage of the resources LRTDP + h_{FF} took to solve the corresponding problems (the higher the curves are, the bigger the savings). No data points for some problems indicate that neither LRTDP + h_{FF} nor LRTDP + h_{FF} + 6S could solve them with only 2 GB of RAM. There are a few large problems that could only be solved by LRTDP + h_{FF} + 6S. Their data points are marked with a × and savings for them are set at 100% (e.g., on problem 14 of EBW-06) as a matter of visualization, because we do not know how much resources LRTDP + h_{FF} would need to solve them. Additionally, we point out that as a general trend, problems grow

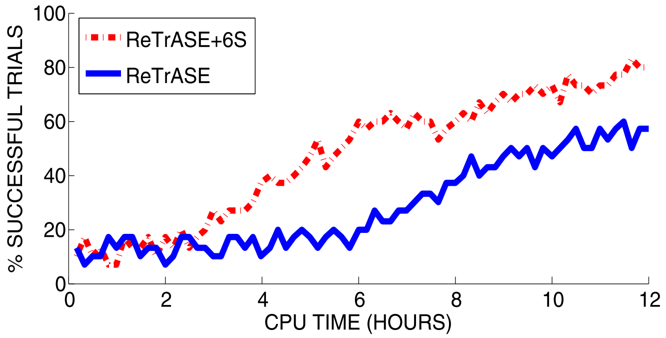


Fig. 15. SixthSense speeds up ReTrASE by as much as 60% on problems with dead ends. The plot shows this trend on the example of problem 12 of EBW-06.

in complexity within each domain with the increasing ordinal. However, the increase in difficulty is not guaranteed for any two adjacent problems, especially in domains with a rich structure, causing the jaggedness of graphs for EBW-06 and -08.

As the graphs demonstrate, the memory savings on average grow very gradually but can reach a staggering 90% on the largest problems. In fact, on the problems marked with a ×, they enable LRTDP + h_{FF} + 6S to do what LRTDP + h_{FF} cannot. The crucial qualitative distinction of LRTDP + h_{FF} + 6S from LRTDP + h_{FF} explaining this is that since nogoods help the former recognize more states as dead ends it does not explore (and hence memoize) their descendants. Notably, the time savings are lagging for the smallest and some medium-sized problems (approximately 1–7). However, each of them takes only a few seconds to solve, so the overhead of SixthSense may be slightly noticeable. On large problems, SixthSense fully comes into its element and saves 30% or more of the planning time.

Benefits to sensitive but slow. Planners of this type include top IPPC performers RFF and HMDPP, as well as ReTrASE and others. Most of them use a deterministic planner, e.g., FF, on a domain determinization to find a plan from the given state to the goal and use such plans in various ways to construct a policy. Whenever the deterministic planner can prove non-existence of a path to the goal or fails to simply find one within a certain time, these MDP solvers consider the state from which the planner was launched to be a dead end. Due to the properties of classical planners, this method of dead-end identification is reliable but expensive. To model it, we employed LRTDP with the GOTH heuristic. GOTH evaluates states with classical planners, so including or excluding SixthSense from GOTH allows for simulating the effects SixthSense has on the above algorithms. As SixthSense is part of the standard GOTH implementation, GOTH without it is denoted as GOTH/NO 6S. Fig. 14 illustrates LRTDP + GOTH's behavior. Qualitatively, the results look similar to LRTDP + h_{FF} + 6S but there is a subtle critical difference — the time savings in the latter case grow faster. This is a manifestation of the fundamental distinction of SixthSense in the two settings. For the "Sensitive but Slow", SixthSense helps recognize implicit dead ends faster (and obviates memoizing them). For the "Fast but Insensitive", it also obviates exploring many of the implicit dead ends' descendants, causing a faster savings growth with problem size.

Benefits to ReTrASE. ReTrASE is perhaps the most natural of MDP solver to be augmented with SixthSense. It already uses basis functions to store information about non-dead-end states, and utilizing nogoods would allow it to capitalize on the abstraction framework even more, providing additional insights into the benefits for other planners that might employ the abstraction framework to serve all of their state space representation needs.

To measure the effect of SixthSense on ReTrASE and get a different perspective on the role of SixthSense than in the previous experiments, we ran ReTrASE and ReTrASE + 6S for at most 12 hours on each of the 45 problems of the EBW-06, -08, and Drive sets, and noted the policy quality, as reflected by the success rate, at fixed time intervals. For smaller problems, we measured policy quality every few seconds, whereas for larger ones — every 5–10 minutes. Qualitatively, the trends on all the problems were similar, so here we study them on the example of problem 12 from the EBW-06 set, one of the hardest problems attempted. For this problem, after 12 hours of CPU running time ReTrASE + 6S extracted and learned weights for 62 267 basis functions; it also discovered 79 623 dead ends states. Out of these dead ends, 18 392 were identified by ReTrASE + 6S running a deterministic planner starting at them having this planner fail to find a path to the goal. The remainder, i.e. 77%, were discovered with 15 nogoods that SixthSense derived. Since every deterministic planner call from a non-dead-end state typically yields several basis functions, SixthSense saved ReTrASE at least $(79\,623 - 18\,392)/(62\,267 + 79\,623) \approx 43\%$ of classical planner invocations, with accompanying time savings. On the other hand, ReTrASE's running time is not occupied solely by basis function extraction — a significant fraction of it consists of basis function weight learning and state space exploration. Besides, SixthSense, although fast, was not instantaneous. Therefore, based on this model we expected the overall speedup caused by SixthSense to be less than 40% and likely also less than 30%.

With this in mind, please refer to Fig. 15 showing the plots of policy quality yielded by ReTrASE and ReTrASE + 6S versus time. As expected intuitively, the use of SixthSense does not change ReTrASE's pattern of convergence, and the shape of the two plots are roughly similar. (If allowed to run for long enough both planners should converge to policies of the same quality, although the plots do not show this.) However, surprisingly, the time it takes ReTrASE + 6S to arrive at a policy of the quality ReTrASE gets after 12 hours of execution turns out to be about 5.5 hours. Thus, the speedup SixthSense has yielded is considerably larger than predicted by our model, roughly 60% versus the expected 30 or less.

Additional code instrumentation revealed an explanation for this discrepancy. The model just sketched implicitly assumes that the time cost of a successful deterministic planner call (one that yields basis functions) and one that proves the state to be a dead end to be the same. This appears to be far from reality; the latter, on average, was over 4 times more expensive. With this factor taken into account, the model would forecast a 77% time savings on classical planner calls due to employment of SixthSense. With the adjustments we described earlier that need to be made to this figure, it agrees well with the actual data.

Regarding memory savings, SixthSense helps ReTrASE as well, but the picture here is much clearer. Indeed, since ReTrASE memoizes only basis functions (with weights), dead ends, and nogoods, a 43% reduction in the total number of these as predicted by our model straightforwardly translates to the corresponding memory reduction our experiments showed. We point out, however, that even without SixthSense, ReTrASE's memory requirements are very low compared to other MDP solvers, and reducing them even further is a less significant performance gain than the boost in speed.

Last but not least, we found that SixthSense almost never takes more than 10% of LRTDP + h_{FF} + 6S's or LRTDP + GOTH's running time. For LRTDP + h_{FF} + 6S, this fraction includes the time spent on deterministic planner invocations to obtain the basis functions, whereas in LRTDP + GOTH, the classical plans are available to SixthSense for free. In fact, as the problem size grows, SixthSense eventually gets to occupy less than 0.5% of the total planning time. As an illustration of SixthSense's operation, we found out that it always finds the single nogood in the Drive domain after using just 10 dead ends for training, and manages to acquire most of the statistically significant *immediate* dead ends in EBW. In the available EBW problems, their number is always less than several dozens, which, considering the space savings they bring, attests to nogoods' high efficiency.

5.4. Summary

GOTH is a machine learning algorithm for discovering the counterpart of basis functions, nogoods. The presence of a nogood in a state guarantees the state to be a dead end. Thus, nogoods help a planner quickly identify dead ends without memoizing them, helping save memory and time. GOTH serves as a submodule of a planner that periodically attempts to "guess" nogoods using dead ends the planner visited and basis functions the planner discovered as training data. It checks each guess using a sound planning graph-based verification procedure. Depending on the type of MDP solver, GOTH vastly speeds it up, reduces its memory footprint, or both, on MDPs with dead-end states.

6. Discussion

The experiments indicate that the proposed abstraction framework is capable of advancing the state of the art in planning under uncertainty. Nonetheless, there are several promising directions for future improvement.

Making structure extraction faster. Even though employment of basis functions in GOTH renders GOTH much faster than otherwise, the relatively few classical planner invocations that have to be made are still expensive, and GOTH's advantage in informativeness is not always sufficient to secure an overall advantage in speed for the MDP solver that uses it. Incidentally, we noticed that on some of the domains ReTrASE spends a lot of time discovering basis functions that end up having high weights (i.e., are not very "important"). We see two ways of handling the framework's occasional lack of speed in discovering useful problem structure.

The first approach is motivated by noticing that the speed of basis function extraction depends critically on how fast the available deterministic planners are on the deterministic version of the domain at hand. Therefore, the speed issue can be alleviated by adding more modern classical planners to the portfolio and launching them in parallel in the hope that at least one will be able to cope quickly with the given domain. Of course, this method may backfire when the number of employed classical planners exceeds the number of cores on the machine where the MDP solver is running, since the planners will start contending for resources. Nonetheless, up to that limit, increasing the portfolio size should only help. In addition, using a reasonably-sized portfolio of planners may help reduce the variance and the average of the time it takes to arrive at a deterministic plan.

The above idea is an extensional approach to accelerate the domain structure extraction, one that increases the performance of the algorithm by making more computational resources available to it. There is also an intensional one, that improves the algorithm itself. The ultimate reason for frequent discovery of "useless" basis functions via deterministic planning is the fact that a basis function's importance is largely determined by the probabilistic properties of the corresponding trajectory, something the all-outcomes determinization completely discards. An alternative would be to give classical planners a domain determinization that retains at least some of its probabilistic structure. Although seemingly paradoxical, such determinizations exist, e.g. the one proposed by the authors of HMDPP. Its use could improve the quality of obtained basis functions and thus reduce the deterministic planning time spent on discovering subpar ones. Different determiniza-

```
(:action be-evil
 :parameters ()
 :precondition (and (gremlin-alive))
 :effect (and
             (if (and (has Screwdriver) (has Wrench))
                 (and (plane-broken)))
             (if (and (has Hammer))
                 (and (plane-broken)
                      (probabilistic 0.9
                      (and (not (gremlin-alive)))))))
```

Fig. 16. Action with conditional effects.

tion strategies may also ease the task of the classical planners provided that the determinization avoids enumerating all outcomes of every action without significant losses in solution quality.

Lifting representation to first-order logic. Another potentially fruitful research direction is increasing the power of abstraction by lifting the representation of basis functions and nogoods to first-order logic. Such representation's benefits are apparent, for example, in the EBW domain. In EBW, besides the immediate nogoods, there are others of the form "block *b* is not in its goal position and has an exploded block somewhere in the stack above it". Indeed, to move *b* one would first need to remove all blocks, including the exploded one, above it in the stack, but in EBW exploded blocks cannot be relocated. Expressed in first-order logic, the above statement would clearly capture many dead ends. In propositional logic, however, it would translate to a disjunction of many ground conjunctions, each of which is a nogood. Each such ground nogood separately accounts for a small fraction of dead ends in the MDP and is almost undetectable statistically, preventing SixthSense from discovering it.

Handling conditional effects. So far, we have assumed that an action's precondition is a simple conjunction of literals. PPDDL's most recent versions allow for a more expressive way to describe an action's applicability via conditional effects. Fig. 16 shows an action with this feature. In addition to the usual precondition, this action has a separate precondition controlling each of its possible effects. Depending on the state, any subset of the action's effects can be executed.

The presented algorithms currently do not handle problems with this construct for two reasons.

First, regression as defined in Section 2.2 does not work for conditional effects. However, its definition can be easily extended to such cases. As a starting step, consider a goal trajectory $t(e)$ and suppose that outcome $out(a_i, i, e)$, part of $t(e)$, is the result of applying action a_i in state s_{i-1} of e. Denote the precondition of k-th conditional effect of a_i as $cond\_prec_k(a_i)$. When e was sampled, conjunction $out(a_i, i, e)$ was generated in the following way. For every k, it was checked whether $cond\_prec_k(a_i)$ holds in s_{i-1}. If it did, the dice were rolled to select the outcome of the corresponding conditional effect. Denote this outcome as $cond\_out_k(a_i, i, e)$. Furthermore, let $lit(cond\_out_k(a_i, i, e)) = \emptyset$ (i.e., let $cond\_out_k(a_i, i, e)$ be an empty conjunction) if $cond\_prec_k(a_i)$ does not hold in s_i.

By definition, $cond\_prec_k(a_i)$ can be empty in either of two cases:

- If $cond\_prec_k(a_i)$ does not hold in s_{i-1};
- If $cond\_prec_k(a_i)$ holds in s_{i-1} but while sampling $cond\_out_k(a_i, i, e)$ we happened to pick an outcome that does not modify s_{i-1}.

In the light of this fact, define the *cumulative precondition* of $out(a_i, i, e)$ as

$$cu\_prec\big(out(a_i, i, e)\big) = prec(a_i) \wedge \Big[\bigwedge_k \big\{cond\_prec_k(a_i) \bigm| lit\big(cond\_out_k(a_i, i, e)\big) \neq \emptyset\big\}\Big]$$

and observe that

$$out(a_i, i, e) = \bigwedge_k \big\{cond\_out_k(a_i, i, e)\big\}.$$

Thus, $cu\_prec(out(a_i, i, e))$ is a conjunction of preconditions of those conditional effects of a_i that contributed at least one literal to $out(a_i, i, e)$. In other words, it is the *minimum necessary precondition* of $out(a_i, i, e)$. Therefore, to extend regression to actions with conditional effects we simply substitute $cu\_prec(out(a_i, i, e))$ for $prec(a_i)$ into the formulas for generating basis functions from Section 2.2 to obtain

$$b_0 = \mathcal{G},$$
$$b_i = \bigwedge \big[\big[lit(b_{i-1}) \cup lit\big(out(a_{n-i+1}, n-i+1, e)\big)\big] \setminus lit\big(cu\_prec\big(out(a_i, i, e)\big)\big)\big] \quad \text{for } 1 \leqslant i \leqslant n.$$

Unfortunately, there is a second, practical difficulty with making GOTH, ReTrASE, and SixthSense work in the presence of conditional effects. Recall that our primary way of obtaining goal trajectories for regression is via deterministic planning.

Determinizing an ordinary probabilistic action yields the number of deterministic actions equal to the number of original action's outcomes. In the presence of conditional effects this statement needs qualification. Each conditional effect can be thought of as describing an "action within an action" with its own probabilistic outcomes. These "inside actions" need not be mutually exclusive. Therefore, the number of outcomes of an action with conditional effects is generally *exponential* in the latter's number. As a consequence, determinizing such actions may lead to a blowup in problem representation size. Further research is needed to identify special cases in which the determinization of conditional effects can be done efficiently.

Beyond stochastic shortest path MDPs. So far, the probabilistic planning community has predominantly concentrated on stochastic shortest path (SSP) MDPs and its subclasses (e.g., the discounted cost MDPs). However, there are interesting problems beyond SSP MDPs as well. As an example, consider the SysAdmin domain [9], in which the goal is to keep a network of computers running for as long as possible. This type of reward maximization problems has received little attention up till now, although there has been a recent attempt to tackle it efficiently with heuristic search [34].

Extending the abstraction framework to reward-maximization MDPs is a potentially impactful research direction. However, it meets with a serious practical as well as theoretical difficulty. Recall that the natural deterministic analog of SSP MDPs are shortest path problems. Researchers have studied them extensively and developed a wide range of very efficient tools for solving them, such as FF, LPG, LAMA, and others. As shown earlier, techniques presented here critically rely on these tools for extracting the basis functions and estimating their weights. However, the closest classical counterpart of probabilistic reward-maximization scenarios are *longest* path problems. Known algorithms for various formulations of this setting are at best exponential *in the state space size*, explaining the lack of fast solvers for them. In their absence, the invention of alternative efficient ways of extracting important causal information is the first step on the way to extending abstraction beyond SSP MDPs.

Abstraction framework and existing planners. Despite improvements being possible, our abstraction framework is useful even in its current state, as evidenced by both the experimental and theoretical results. Moreover, it has a property that makes its use very practical; the framework is complementary to the other powerful ideas incorporated in successful solvers of the recent years, e.g., HMDPP, RFF, FFHop, and others. Thus, abstraction can greatly benefit many of these solvers with no or few sacrifices on their part, and also inspire new ones. As an example, note that FFReplan could be enhanced with abstraction in the following way. It could extract basis functions from deterministic plans it is producing while trying to reach the goal and store each of them along with their weight and the last action regressed before obtaining that particular basis function. Upon encountering a state subsumed by at least one of the known basis functions, "generalized FFReplan" would select the action corresponding to the basis function with the smallest weight. Besides an accompanying speed boost, which is a minor point in the case of FFReplan since it is very fast as is, FFReplan's robustness could be greatly improved, since this way its action selection would be informed by several trajectories from the state to the goal, as opposed to just one. Employed analogously, basis functions could speed up FFHop, an MDP solver that has great potential but is somewhat slow in its current form. In fact, we believe that *virtually any* algorithm for solving SSP MDPs could have its convergence accelerated if it regresses the trajectories found during policy search and carries over information from well explored parts of the state space to the weakly probed ones with the help of basis functions and nogoods. We hope to verify this conjecture in the future. At the same time, solvers of discounted-reward MDPs are unlikely to gain much from the kind of abstraction proposed in this paper, even though mathematically the described techniques will work even on this MDP class. Discounted-reward MDPs can be viewed as SSP MDPs where each action has some probability of leading directly to the goal [4]. As a result, any sequence of actions in a discounted-reward MDP is a goal trajectory. This leads to an overabundance of basis functions, potentially making their number comparable to the number of states in the problem.

A different approach for having abstraction benefit existing planners is to let ReTrASE produce a value function estimate and to allow another planner, e.g. LRTDP, complete the solution of the problem starting from this estimate. This idea is motivated by the fact that it is hard to know when ReTrASE has converged on a given problem (and whether it ever will). Therefore, it makes sense to have an algorithm with convergence guarantees take over from ReTrASE at a certain point. Empirical research is needed to determine when the switch from ReTrASE to another solver should happen.

7. Related work

In spirit, the concept of extracting useful state information in the form of basis functions is related to explanation-based learning (EBL) [30,27]. In EBL, the planner would try to derive control rules for action selection by analyzing its own execution traces. In practice, EBL systems suffer from accumulating too much of such information, whereas the approaches we have presented do not. The idea of using determinization followed by regression to obtain basis functions 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 [21,39]. However, our function aggregation and weight learning methods are completely different from theirs.

ReTrASE, in essence, exploits basis functions to perform dimensionality reduction, but basis functions are not the only known alternative to serve this purpose. 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 [25,15]. While they are a significant improvement in efficiency over their non-symbolic counterparts, these optimal algorithms still do not scale to large problems. APRICODD, an approximation scheme developed over SPUDD [40], showed promise, but it is not clear whether it is competitive with today's top methods since it has not been applied to the competition domains.

Some researchers have applied non-linear techniques like exponential-PCA and NCA for dimensionality reduction [38,28]. These methods assume the original state space to be continuous and hence are not applicable to typical planning benchmarks.

In fact, most basis function-based dimensionality reduction techniques are not applied in nominal domains. A notable exception is FPG [10], which performs policy search and represents the policy compactly with a neural network. Our experiments demonstrate that ReTrASE outperforms FPG consistently on several domains.

The use of determinization for solving MDPs in general was inspired by advances in classical planning, most notably the FF solver [26]. The practicality of the new technique was demonstrated by FFReplan [42] that used the FF planner on an MDP determinization for direct selection of an action to execute in a given state. More recent planners to employ determinization that are, in contrast to FF-Replan, successful at dealing with probabilistically interesting problems include RFF-RG/BG [41]. At the same time, the latter kind of algorithms typically invokes a deterministic planner many more times than our techniques do. This forces them to avoid all-outcomes determinization as these invocations would be too costly otherwise. Other related planners include Temptastic [44], precautionary planning [17], and FFHop [43].

The employment of determinization for heuristic function computation was made famous by the FF heuristic, h_{FF} [26], originally part of a classical planner by the same name. LRTDP [7] and HMDPP [29] adopted this heuristic with no modifications as well. In particular, HMDPP runs h_{FF} on a "self-loop determinization" of an MDP, thereby forcing h_{FF}'s estimates to take into account some of the problem's probabilistic information.

To our knowledge, there have been no previous attempts to handle identification of dead ends in MDPs. The "Sensitive but Slow" and "Fast but Insensitive" mechanisms were not actually designed specifically for the purpose of identifying dead ends and are unsatisfactory in many ways. One possible reason for this omission may be that most MDPs studied by the Artificial Intelligence and Operations Research communities until recently had no dead ends. However, MDPs with dead ends have been receiving attention in the past few years as researchers realized their *probabilistic interestingness* [35]. Besides the analogy to EBL, SixthSense can also be viewed as a machine learning algorithm for rule induction, similar in purpose, for example, to CN2 [12]. While this analogy is valid, SixthSense operates under different requirements than most such algorithms, because we demand that SixthSense-derived rules (nogoods) have zero false-positive rate. Last but not least, our term "nogood" shares its name with and closely mirrors the concept from the areas of truth maintenance systems (TMSs) [13] and constraint satisfaction problems (CSPs) [14]. However, our methodology for finding nogoods has little in common with algorithms used in that literature.

8. Conclusions

A central issue that limits practical applicability of automated planning under uncertainty is scalability of available techniques. In this article, we have presented a powerful approach to tackle this fundamental problem — an abstraction framework that extracts problem structure and exploits it to spread information gained by exploring one part of the MDP's state space to many others.

The components of the framework are the elements of problem structure called *basis functions* and *nogoods*. The basis functions are preconditions for those sequences of actions (*trajectories*) that take the agent from some state to the goal with positive probability. As such, each applies in many of the MDP's states, sharing associated reachability information across them. Crucially, basis functions are easy to come by via fast deterministic planning or even as a byproduct of the normal probabilistic planning process. While basis functions describe only MDP states from which reaching the goal is possible, their counterparts, nogoods, identify dead ends, from which the goal cannot be reached. Crucially, the number of basis functions and nogoods needed to characterize the problem space is typically vastly smaller than the problem's state space. Thus, the framework can be used in a variety of ways that increase the scalability of the state of the art methods for solving MDPs.

We have described three approaches illustrating the framework's operation, GOTH, ReTrASE, and SixthSense. The experimental results show the promise of the outlined abstraction idea. Although we describe several ways to enhance our existing framework, even as is it can be utilized to qualitatively improve scalability of virtually any modern MDP solver and inspire the techniques of tomorrow.

References

[1] D. Aberdeen, S. Thiébaux, L. Zhang, Decision-theoretic military operations planning, in: ICAPS'04, 2004.
[2] A. Barto, S. Bradtke, S. Singh, Learning to act using real-time dynamic programming, Artificial Intelligence 72 (1995) 81–138.
[3] R. Bellman, Dynamic Programming, Princeton University Press, 1957.
[4] D. Bertsekas, J. Tsitsiklis, Neuro-Dynamic Programming, Athena Scientific, 1996.
[5] D. Bertsekas, Dynamic Programming and Optimal Control, Athena Scientific, 1995.
[6] A. Blum, M. Furst, Fast planning through planning graph analysis, Artificial Intelligence 90 (1997) 281–300.
[7] B. Bonet, H. Geffner, Labeled RTDP: Improving the convergence of real-time dynamic programming, in: ICAPS'03, 2003, pp. 12–21.

[8] B. Bonet, H. Geffner, mGPT: A probabilistic planner based on heuristic search, Journal of Artificial Intelligence Research 24 (2005) 933–944.
[9] D. Bryce, O. Buffet, International planning competition, uncertainty part: Benchmarks and results, http://ippc-2008.loria.fr/wiki/images/0/03/Results.pdf, 2008.
[10] O. Buffet, D. Aberdeen, The factored policy-gradient planner, Artificial Intelligence Journal 173 (2009) 722–747.
[11] Tom Bylander, The computational complexity of propositional STRIPS planning, Artificial Intelligence 69 (1994) 165–204.
[12] Peter Clark, Tim Niblett, The CN2 induction algorithm, in: Machine Learning, 1989, pp. 261–283.
[13] Johan de Kleer, An assumption-based tms, Artificial Intelligence 28 (1986) 127–162.
[14] R. Dechter, Constraint Processing, Morgan Kaufmann Publishers, 2003.
[15] Z. Feng, E. Hansen, Symbolic heuristic search for factored Markov decision processes, in: AAAI'02, 2002.
[16] C. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence 19 (1982) 17–37.
[17] J. Foss, N. Onder, D. Smith, Preventing unrecoverable failures through precautionary planning, in: ICAPS'07 Workshop on Moving Planning and Scheduling Systems into the Real World, 2007.
[18] A. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs, Journal of Artificial Intelligence Research 20 (2003) 239–290.
[19] J. Goldsmith, M. Littman, M. Mundhenk, The complexity of plan existence and evaluation in probabilistic domains, in: UAI'97, 1997.
[20] Geoff Gordon, Stable function approximation in dynamic programming, in: ICML, 1995, pp. 261–268.
[21] C. Gretton, S. Thiébaux, Exploiting first-order regression in inductive policy selection, in: UAI'04, 2004.
[22] C. Guestrin, D. Koller, C. Gearhart, N. Kanodia, Generalizing plans to new environments in relational MDPs, in: IJCAI'03, Acapulco, Mexico, 2003.
[23] Carlos Guestrin, Daphne Koller, Ronald Parr, Shobha Venkataraman, Efficient solution algorithms for factored MDPs, Journal of Artificial Intelligence Research 19 (2003) 399–468.
[24] E. Hansen, S. Zilberstein, LAO*: A heuristic search algorithm that finds solutions with loops, Artificial Intelligence 129 (1–2) (2001) 35–62.
[25] J. Hoey, R. St-Aubin, A. Hu, C. Boutilier, SPUDD: Stochastic planning using decision diagrams, in: UAI'99, 1999, pp. 279–288.
[26] J. Hoffman, B. Nebel, The FF planning system: Fast plan generation through heuristic search, Journal of Artificial Intelligence Research 14 (2001) 253–302.
[27] S. Kambhampati, S. Katukam, Q. Yong, Failure driven dynamic search control for partial order planners: An explanation based approach, Artificial Intelligence 88 (1996) 253–315.
[28] Philipp Keller, Shie Mannor, Doine Precup, Automatic basis function construction for approximate dynamic programming and reinforcement learning, in: ICML'06, 2006, pp. 449–456.
[29] E. Keyder, H. Geffner, The HMDPP planner for planning with probabilities, in: Sixth International Planning Competition at ICAPS'08, 2008.
[30] C. Knoblock, S. Minton, O. Etzioni, Integrating abstraction and explanation-based learning in PRODIGY, in: Ninth National Conference on Artificial Intelligence, 1991.
[31] A. Kolobov, Mausam, D. Weld, ReTrASE: Integrating paradigms for approximate probabilistic planning, in: IJCAI'09, 2009.
[32] A. Kolobov, Mausam, D. Weld, Classical planning in MDP heuristics: With a little help from generalization, in: ICAPS'10, 2010.
[33] A. Kolobov, Mausam, D. Weld, SixthSense: Fast and reliable recognition of dead ends in MDPs, in: AAAI'10, 2010.
[34] A. Kolobov, Mausam, D. Weld, Heuristic search for generalized stochastic shortest path mdps, in: ICAPS'11, 2011.
[35] Iain Little, Sylvie Thiébaux, Probabilistic planning vs. replanning, in: ICAPS Workshop on IPC: Past, Present and Future, 2007.
[36] Mausam, E. Benazara, R. Brafman, N. Meuleau, E. Hansen, Planning with continuous resources in stochastic domains, in: IJCAI'05, 2005, p. 1244.
[37] Mausam, P. Bertoli, D. Weld, A hybridized planner for stochastic domains, in: IJCAI'07, 2007.
[38] Nicholas Roy, Geoffrey Gordon, Exponential family PCA for belief compression in POMDPs, in: NIPS'02, MIT Press, 2003, pp. 1043–1049.
[39] S. Sanner, C. Boutilier, Practical linear value-approximation techniques for first-order MDPs, in: UAI'06, 2006.
[40] R. St-Aubin, J. Hoey, C. Boutilier, APRICODD: Approximate policy construction using decision diagrams, in: NIPS'00, 2000.
[41] F. Teichteil-Königsbuch, U. Kuter, G. Infantes, Incremental plan aggregation for generating policies in MDPs, in: AAMAS'10, 2010.
[42] S. Yoon, A. Fern, R. Givan, FF-Replan: A baseline for probabilistic planning, in: ICAPS'07, 2007, pp. 352–359.
[43] S. Yoon, A. Fern, S. Kambhampati, R. Givan, Probabilistic planning via determinization in hindsight, in: AAAI'08, 2008.
[44] H.L.S. Younes, R.G. Simmons, Policy generation for continuous-time stochastic domains with concurrency, in: ICAPS'04, 2004, p. 325.