SixthSense: Fast and Reliable Recognition of Dead Ends in MDPs

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

Abstract

The results of the latest International Probabilistic Planning Competition (IPPC-2008) indicate that the presence of dead ends, states with no trajectory to the goal, makes MDPs hard for modern probabilistic planners. Implicit dead ends, states with executable actions but no path to the goal, are particularly challenging; existing MDP solvers spend much time and memory identifying these states.

As a first attempt to address this issue, we propose a machine learning algorithm called SIXTHSENSE. SIXTHSENSE helps existing MDP solvers by finding nogoods, conjunctions of literals whose truth in a state implies that the state is a dead end. Importantly, our learned nogoods are sound, and hence the states they identify are true dead ends. SIXTHSENSE is very fast, needs little training data, and takes only a small fraction of total planning time. While IPPC problems may have millions of dead ends, they may typically be represented with only a dozen or two no-goods. Thus, nogood learning efficiently produces a quick and reliable means for dead-end recognition. Our experiments show that the nogoods found by SIXTHSENSE routinely reduce planning space and time on IPPC domains, enabling some planners to solve problems they could not previously handle.

Introduction

Recent work on "probabilistic interestingness" has suggested that probabilistic planners have difficulty on domains with avoidable *dead ends*, non-goal states with no potential trajectories to the goal (Little and Thiebaux 2007). This conjecture is supported by the recent International Probabilistic Planning Competition (IPPC) (Bryce and Buffet 2008), in which domains with a complex dead-end structure, *e.g.*, Exploding Blocks World, have proven the most challenging. Surprisingly, however, there has been little research on methods for quickly and reliably avoiding such dead ends in Markov Decision Processes (MDPs).

It is useful to distinguish between two types of dead ends. If a non-goal state doesn't satisfy the preconditions of any action, we term it an *explicit* dead end. *Implicit* dead ends, on the other hand, have executable actions (just no workable path to the goal) and are much harder for MDP solvers to detect than the explicit type.

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 (Bonet and Geffner 2003)) apply a heuristic value function, 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, causing the planner to waste much time in subsequent search. Other MDP solvers use state-value estimation approaches that recognize dead ends reliably but are very expensive; for example, RFF (Teichteil-Koenigsbuch, Infantes, and Kuter 2008), HMDPP (Keyder and Geffner 2008) and ReTrASE (Kolobov, Mausam, and Weld 2009) 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-end states. Indeed, most probabilistic planners would run faster if they could quickly recognize implicit dead ends.

This paper presents SIXTHSENSE, a novel mechanism to do exactly that — quickly and reliably identify dead-end states in MDPs. Underlying SIXTHSENSE is a key insight: large sets of dead-end states can usually be characterized by a compact logical conjunction, called 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 domains can be described with a small number of nogoods. SIXTHSENSE learns nogoods by generating candidates with a bottom-up greedy search (akin to that used in rule induction (Clark and Niblett 1989)) and tests them to avoid false positives with a planning graph-based procedure. We make the following contributions:

- We identify implicit dead ends as an important problem for probabilistic planning and present a fast domain-independent machine learning algorithm for finding nogoods, very compact representations of large sets of dead-end states.
- We show that SIXTHSENSE is sound — every nogood output represents a set of true dead-end states.
- We empirically demonstrate that SIXTHSENSE speeds up two different types of MDP solvers on several IPPC domains with implicit dead ends. 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 some planners to solve problems they couldn't previously handle.

Background

Markov Decision Processes (MDPs). In this paper, we focus on probabilistic planning problems that are modeled by factored goal-oriented indefinite-horizon MDPs. They are defined as tuples of the form $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{T}$ is a transition function $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ giving the probability of moving from s_i to s_j by executing a, $\mathcal{C}$ is a map $\mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^+$ specifying action costs, s_0 is the start state, and $\mathcal{G}$ is a set of (absorbing) goal states. *Indefinite horizon* refers to the fact that the total action cost is accumulated over a finite-length action sequence whose length is *apriori* unknown. Our method also handles discounted infinite-horizon MDPs, which reduce to the goal-oriented case (Bertsekas and Tsitsiklis 1996).

In a factored MDP, each state is represented as a conjunction of domain literals. We concentrate on MDPs with goals that are literal conjunctions as well. Solving an MDP means finding a good (*i.e.*, cost-minimizing) policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ 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 satisfies the following conditions, called *Bellman equations*:

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

Given $V^*(s)$, an optimal policy may be computed as follows: $\pi^*(s) = \text{argmin}_{a \in \mathcal{A}} [\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') V^*(s')]$.

Solution Methods. The above equations suggest a dynamic programming-based way of finding an optimal policy, called *value iteration* (VI), which iteratively updates state values using Bellman equations in a *Bellman backup* and follows the resulting policy until the values converge. VI has given rise to many improvements. Trial-based methods, *e.g.*, RTDP, 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' (Bonet and Geffner 2003).

Basis Functions. Most modern factored MDP solvers operate in trials aimed at achieving the goal while looking for a policy. Successful trials produce goal trajectories, which are, in turn, a rich source of information about the structure of the problem at hand. For instance, regressing the goal through such a trajectory yields a set of literal conjunctions, called *basis functions* (Kolobov, Mausam, and Weld 2009), with an important property: each such conjunction is a precondition for the trajectory suffix that was regressed to generate it. Thus, if a basis function applies in a state, this state can't be a dead end.

Determinization. Whenever an MDP solver finds a successful trajectory to the goal, regression may be used to generate basis functions. However, there is an often faster way to get a trajectory — by running a classical planner on the *all-outcomes* determinization (Yoon, Fern, and Givan 2007) D_d of the domain D at hand. For each action a with precondition c and outcomes $o_1, \ldots, o_n$ with respective probabilities $p_1, \ldots, p_n$, all-outcome determinization produces a set of deterministic actions $a_1, \ldots, a_n$, each with precondition c and effect o_i. A plan from a given state to the goal in the classical domain D_d exists if and only if a corresponding trajectory exists in probabilistic domain D. Therefore, one may use a classical planner on D_d to quickly generate basis functions as needed.

Planning Graph. Blum & Furst (Blum and Furst 1997) define the planning graph as a directed graph alternating between proposition and action levels. The 0^{th} level contains all literals present in an initial state s. Odd levels contain 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 (Blum and Furst 1997) uses the graph as a polynomial-time reachability test for the goal, and we soon show how it may be used as a sufficient condition for nogoods.

Approach

A domain may have an exponential number of dead end states, but usually 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.

The method we are proposing, 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 (or, the states it *represents*) 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 for use by 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 1 contains the learning algorithm's pseudocode.

Generation of Candidate Nogoods. There are many ways to generate a candidate but if, as we surmise, 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 basis functions are literal conjunctions that result from regressing the goal through a trajectory. Thus, the set of all basis functions in a problem is an exhaustive set of sufficient conditions that make states in the problem transient. Since a basis function is a certificate of a positive-probability trajectory to the goal, any state it represents can't be a dead end. On the other hand, any state represented by a nogood by definition *must* be a dead end. These facts combine into a theorem:

Theorem 1. *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 haven't specified how exactly defeating literals should be sampled. 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 "mothership" 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 into account information from the solver's experience but also lets literals' co-occurrence patterns direct creation of the candidate.

Nogood Verification. Let us denote the problem of establishing whether a given conjunction is a nogood as *NOGOOD-DECISION*. A *NOGOOD-DECISION* instance can be broken down into a large set of problems on the all-outcome determinization of the problem at hand, each checking the existence of a path from a state the nogood

Algorithm 1 SIXTHSENSE

1: **Input:** training set of known non-generalized dead ends $setDEs$, set of basis functions $setBFs$, set of nogoods $setNG$, goal conjunction g, set of all domain literals $setL$
2:
3: **function learnNogood**($setDEs$, $setBFs$, $setNGs$, g)
4: // construct a candidate
5: declare candidate conjunction $c = \{\}$
6: **for all** $b \in setBFs$ **do**
7:     **if** c doesn't defeat b **then**
8:         declare literal L = sampleDefeatingLiteral($setDEs$, b, c)
9:         $c = c \cup \{L\}$
10:     **end if**
11: **end for**
12: // check candidate with planning graph, and prune it
13: **if** checkWithPlanningGraph($setL$, c, g) **then**
14:     **for all** literals $L \in c$ **do**
15:         **if** checkWithPlanningGraph($setL$, $c \setminus \{L\}$, g) == $success$ **then**
16:             $c = c \setminus \{L\}$
17:         **end if**
18:     **end for**
19: **else**
20:     return $failure$
21: **end if**
22: // if we got here then the candidate is a valid nogood
23: // remove all dead ends from the training set
24: empty $setDEs$
25: add c to $setNG$
26: **return** $success$
27:
28: **function checkWithPlanningGraph**($setL$, c, g)
29: **for all** literals G in $(g \setminus c)$ **do**
30:     declare conjunction $c' = c \cup ((setL \setminus (\neg c)) \setminus \{G\})$
31:     **if** PlanningGraph(c') == $success$ **then**
32:         return $failure$
33:     **end if**
34: **end for**
35: return $success$
36:
37: **function sampleDefeatingLiteral**($setDEs$, b, c)
38: declare counters $C_{\neg L}$ for all $L \in b \setminus c$
39: **for all** $d \in setDEs$ **do**
40:     **if** c generalizes d **then**
41:         **for all** $L \in b$ s.t. $\neg L \in d$ **do**
42:             $C_{\neg L} + +$
43:         **end for**
44:     **end if**
45: **end for**
46: sample a literal L' according to $P(L') \sim C_{L'}$
47: return L'

candidate represents to the goal. Each of these deterministic plan existence decision problems is polynomial in the size of the original MDP, by definition of the all-outcome determinization, and $PSPACE$-complete, as shown in (Bylander 1994). All of them together can be solved (and therefore, trivially, verified) in polynomial space by a Turing machine that handles them in sequence and reuses space on the tape after completing each instance, until either in some instance a path to the goal is established to exist or until it negatively decides all instances. Hence, *NOGOOD-DECISION* can be verified in polynomial space, proving that *NOGOOD-DECISION* $\in PSPACE$ and, coupled with a straightforward polynomial reduction from the same deterministic plan existence decision problem, yields the following result:

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

Thus, we may realistically expect an efficient algorithm for *NOGOOD-DECISION* to be either sound or complete, but not both. Accordingly, one key contribution of our paper lies in noticing that all the checks in the naive scheme above can be replaced by a single sound operation whose time is polynomial in the problem size and that remains effective in practice despite its theoretical incompleteness. The check is carried out on several *superstates*, amalgamations of states represented by the candidate under verification. Each superstate is a conjunction of the candidate, the negation of one of the goal literals that are not present in the candidate, and all literals over all other variables in the domain. Thus, the number of superstates is linear in the number of literals in the goal conjunction. 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 set of superstates our algorithm constructs is $\{A \wedge \mathbf{B} \wedge C \wedge D \wedge \neg D \wedge E \wedge \neg E, A \wedge B \wedge \neg B \wedge C \wedge D \wedge \neg D \wedge \neg \mathbf{E}\}$. Note that every state in which the candidate holds and the goal doesn't hold (i.e., every state in which the candidate holds and that *could* be a dead end) represents one of these superstates. To find out whether the candidate is a nogood, our procedure runs the planning graph algorithm on each derived superstate. Each instance returns $success$ iff 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. Since the planning graph is sound, $failure$ on all superstate expansions indicates the candidate is a true nogood (lines 28-35), as we state in the following theorem.

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.*

If the 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 don't know *apriori* 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, thereby wasting very little computation time after all common nogoods have been discovered.

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 doesn't 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's 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, 100, and a small number of dead ends, 10, to be accumulated before invoking the first generalization round.
- 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, we witnessed very large training sets to decrease the probability of learning a nogood. This phenomenon can be explained by training sets of large sizes usually containing dead ends from different parts of the state space and hence caused by different nogoods. As an upshot, the literal occurrence statistics induced by such sets make it hard to generate reasonable candidates. This finding led us to restrict the training batch size to 10000. If, due to exponential backoff, the scheduler is forced to wait for the arrival of more than $n > 10000$ dead ends, it skips the first $(n - 10000)$ and retains only the latest 10000 for training. For the same locality considerations, each training set is emptied at the end of each round (line 24).

Algorithm's Properties Before presenting the experimental results, we analyze SIXTHSENSE's properties. One of the most important is summarized in the following theorem, which follows directly from the definition of a nogood:

Theorem 4. *The procedure of identifying dead ends as states in which at least one nogood holds is sound.*

Importantly, SIXTHSENSE puts no bounds on the nogood length, being theoretically capable of discovering any nogood. However, nontrivial theoretical guarantees on the amount of training data needed to construct a nogood of a particular length (even length 1) with at least a certain probability, unsurprisingly, seem to require strong assumptions about reachability of dead ends and about properties of the classical planner used to obtain the basis functions. Such assumptions would cause these guarantees to be of no use in practice. At the same time, we can prove another important property of SIXTHSENSE:

Theorem 5. *Once a nogood has been discovered and memorized by* SIXTHSENSE*,* SIXTHSENSE *will never rediscover it again.*

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. As a result, since 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, and because no training dead end is represented by any known nogood, the probability of sampling a known nogood (lines 5-11) is strictly 0.

Regarding SIXTHSENSE's speed, the number of common 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.

Experimental Results

Our goal in the experiments was to explore the benefits SIXTHSENSE brings to different types of planners, as well as to gauge 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 no-goods are typically easy for SIXTHSENSE to derive.

EBW-06 and -08's dead ends are much more complex, and their structure is unique among the IPPC domains. In this 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 can't be achieved. For example, if block $b4$ needs to be on block $b8$ in the goal configuration then any state in which $b4$ or $b8$ explodes before being picked up by the manipulator is a dead end, represented either by nogood $(not\ (no-destroyed\ b4)) \wedge (not\ (holding\ b4)) \wedge (not\ (on\ b4\ b8))$ or by $(not\ (no-destroyed\ b8)) \wedge (not\ (on\ b4\ b8))$. We call such nogoods *immediate* and point out that EBW has other types of nogoods as well. The variety and structural complexity of EBW nogoods makes them challenging to learn.

Planners. As pointed out in the beginning, 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 FF heuristic (Bonet and Geffner 2005). Denoting the FF heuristic as h_{FF}, we will call this combination LRTDP+h_{FF}, and LRTDP+h_{FF} equipped with SIXTHSENSE — LRTDP+h_{FF}+6S for short. Implementation-wise, SIXTHSENSE is incorporated into h_{FF}, whereby h_{FF}, when evaluating a newly encountered state, first consults the available no-goods; 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.

Figure 1 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 sav-

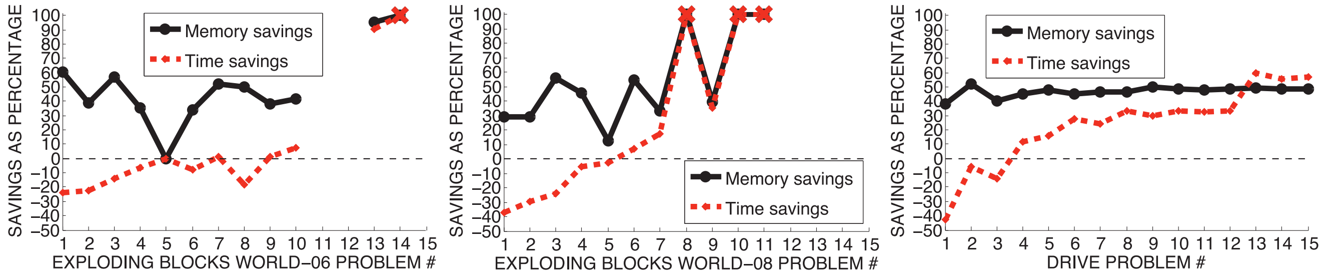

Figure 1: Time and memory savings due to nogoods for LRTDP+h_{FF} (representing the "Fast but Insensitive" type of planners) on 3 domains, as a percentage of resources needed to solve these problems without SIXTHSENSE (higher curves indicate bigger savings; points below zero require more resources with 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 $\times$).

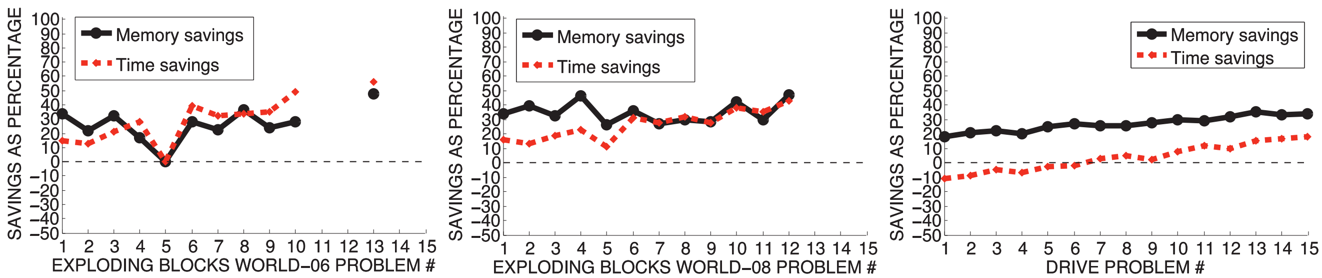

Figure 2: Resource savings from SIXTHSENSE for LRTDP+GOTH (representing the "Sensitive but Slow" type of planners).

ings). No data points for some problems indicate that neither LRTDP+h_{FF} nor LRTDP+h_{FF}+6S could solve them with only 2GB 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 $\times$ and savings for them are set at 100% (*e.g.*, on problem 14 of EBW-06) as a matter of visualization, because we don't know how much resources LRTDP+h_{FF} would need to solve them. Additionally, we point out that as a general trend, problems grow 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 $\times$, they enable LRTDP+h_{FF}+6S to do what LRTDP+h_{FF} can't. 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 doesn't explore (and hence memorize) 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 nonexistence of a path to the goal or fails to simply find one within a certain time, these MDP solvers know 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 (Kolobov, Mausam, and Weld 2010). GOTH evaluates states with classical planners, so incorporating SIXTHSENSE into GOTH allows for simulating the effects SIXTHSENSE has on the above algorithms. Figure 2 illustrates LRTDP+GOTH+6S's behavior. Qualitatively, the results look similar to LRTDP+h_{FF}+6S but in fact there is a 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.

Last but not least, we found that SIXTHSENSE almost never takes more than 10% of LRTDP+h_{FF}+6S's or LRTDP+GOTH+6S'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+6S, 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 a few several dozens, which, considering the space savings they bring, attests to nogoods' high efficiency.

Discussion

Although our preliminary experiments clearly indicate the benefits of nogoods and SIXTHSENSE, we believe that SIXTHSENSE's effectiveness in some settings will increase if the algorithm is extended to generate nogoods in first-order logic. This capability would be helpful, for example, in the EBW domain, where, 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 can't be relocated. Expressed in first-order logic, this 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 domain and is thus almost statistically unnoticeable, preventing SIXTHSENSE from discovering it. Granted, our experiments imply that first-order nogoods are not numerically significant in the benchmark EBW problems. However, one can construct instances where this would not be true.

Related Work

To our knowledge, there have been no explicit previous attempts to handle identification of dead ends in MDPs. The "Sensitive but Slow" and "Fast but Insensitive" mechanisms weren't actually designed specifically for this purpose and are unsatisfactory in many ways described in the Experiments section. The general approach of SIXTHSENSE somewhat resembles work on explanation-based learning (EBL) (C. Knoblock and Etzioni 1991). In EBL, the planner would try to derive control rules for action selection by analyzing its own execution traces. Besides EBL, SIXTHSENSE can also be viewed as a machine learning algorithm for rule induction, similar in purpose, for example, to CN2 (Clark and Niblett 1989). 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 a concept from the area of constraint satisfaction problems (CSPs). However, the semantics of nogoods is CSPs is different, and the methodology for finding them, largely summarized in (Dechter 2003), has nothing in common with ours. The idea of leveraging basis functions was inspired by their use in (Gretton and Thiebaux 2004) and ReTrASE, as well as the evidence provided by solvers like FFReplan and RFF that many deterministic plans that we derive basis functions from can be computed very quickly in quantities.

Conclusion

We have identified recognition of implicit dead ends in MDPs as a source of time and memory savings for probabilistic planners. To materialize these benefits, we proposed SIXTHSENSE, a machine learning algorithm that uncovers the few concise "explanations" of implicit deads inherent in most real-life and artificial scenarios requiring planning under uncertainty. The explanations (nogoods) help the planner recognize most dead ends in a problem quickly and reliably, removing the need for a separate analysis of each such state and expensive methods to do it. We feel that in the future SIXTHSENSE could be improved further by being extended to handle first-order logic expressions, which may be useful in domains like EBW. We empirically illustrate SIXTHSENSE's operation and show how, even as it is, it can help a wide range of existing planners save a large fraction of resources on problems with dead ends. Moreover, these gains are achieved with very little overhead.

Acknowledgments. We would like to thank William Cushing, Peng Dai, Jesse Davis, Rao Kambhampati, and the anonymous reviewers for insightful comments and discussions. This work was supported by ONR grant N000140910051 and the WRF/TJ Cable Professorship.

References

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

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

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

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

Bryce, D., and Buffet, O. 2008. International planning competition, uncertainty part: Benchmarks and results. In *http://ippc-2008.loria.fr/wiki/images/0/03/Results.pdf*.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69:165–204.

C. Knoblock, S. M., and Etzioni, O. 1991. Integrating abstraction and explanation-based learning in PRODIGY. In *Ninth National Conference on Artificial Intelligence*.

Clark, P., and Niblett, T. 1989. The CN2 induction algorithm. In *Machine Learning*, 261–283.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann Publishers.

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

Keyder, E., and Geffner, H. 2008. The HMDPP planner for planning with probabilities. In *Sixth International Planning Competition at ICAPS'08*.

Kolobov, A.; Mausam; and Weld, D. 2009. ReTrASE: Integrating paradigms for approximate probabilistic planning. In *IJCAI'09*.

Kolobov, A.; Mausam; and Weld, D. 2010. Classical planning in MDP heuristics: With a little help from generalization. In *ICAPS'10*.

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

Teichteil-Koenigsbuch, F.; Infantes, G.; and Kuter, U. 2008. RFF: A robust, FF-based MDP planning algorithm for generating policies with low probability of failure. In *Sixth International Planning Competition at ICAPS'08*.

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