Hidden Structure of Factored 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

1 INTRODUCTION

A central issue that limits practical applicability of automated planning under uncertainty is the scalability of available techniques. Despite decades of progress in this field, humans can reasonably well solve many planning problems beyond the reach of general MDP solvers. Among the factors enabling people to do it is our ability to see structure in a given problem and utilize it to avoid unnecessary reasoning. For instance, we know that to put a nail into the wall with a hammer in the living room it is enough to pick up both and hit the former with the latter. If we want to do it in the bedroom we use just the same plan without thinking further. This is not "obvious" to most MDP solvers, which, when faced with producing a plan for something as simple as putting up two paintings in two rooms, will do twice as much planning as necessary, a gross inefficiency.

The research described in this dissertation empowers MDP solvers by adding the capacity to extract problem structure in a domain-independent way and use it to *generalize* information gained by exploring one part of the MDP's state space to many others. The means of such knowledge transfer are the elements of problem structure we call *basis functions*. From a high-level perspective, basis functions are preconditions for sequences of actions that take the agent from some state to the goal. Like preconditions of single actions, they apply in many states, and therein lies their power.

We have developed three techniques that leverage basis functions and a host of machine learning algorithms to uncover the problems' even richer structure, the knowledge these techniques convert into large time and memory savings:

- RETRASE, an MDP solver that simultaneously discovers basis functions and performs decision-theoretic analysis on them to estimate their "usefulness" expressed as a numeric weight. By aggregating the weights, RETRASE constructs better policies than IPPC participants on many domains while using little memory.
- GOTH, a heuristic for MDP solvers that uses full-fledged classical planners to come up with state estimates. Such planners would be too expensive to call from every state that needs evaluation. GOTH uses basis functions to share values among states and avoid many of these invocations. Comparative empirical evaluation shows GOTH to be an informative heuristic that saves MDP solvers a lot of time and memory.
- SIXTHSENSE, a technique for quickly and reliably identifying *implicit dead ends* in MDPs. These states have no trajectories to the goal but are expensive to identify as such due to the presence of executable actions. Acting as a submodule of an MDP solver, SIXTHSENSE helps it easily and soundly detect implicit dead ends. It does so by employing basis functions in a machine learning algorithm to derive literal conjunctions whose presence in a state guarantees the state to be a dead end. The resource savings due to SIXTHSENSE can reach 90%.

In the rest of the paper, we briefly describe each of the above algorithms and outline directions for future research.

2 BACKGROUND

Markov Decision Processes (MDPs). We focus on probabilistic planning problems modeled by factored 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 $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ giving the probability of moving from s_i to s_j by executing a, $\mathcal{C}$ is a map $\mathcal{S} \times \mathcal{A} \to \mathbb{R}^+$ specifying action costs, s_0 is the start state, and $\mathcal{G}$ is a set of (absorbing) goal states. *Indefinite horizon* refers to the total action cost being accumulated over an action sequence whose length is finite but unknown. In factored MDPs, states are represented as conjunctions of domain variable values.

Solving an MDP, i.e. finding a cost-minimizing policy $\pi : \mathcal{S} \to \mathcal{A}$ that specifies the actions the agent should take to eventually reach the goal, can be done by a variety of *value-iteration* (VI) based or *policy-iteration* (PI) based methods. VI-based techniques use Bellman equations in a *Bellman backup* to update the *value function* and follow the resulting policy until convergence. Trial-based improvements on VI, RTDP [1] and LRTDP [3], serve as testbeds in some of our experiments.

Heuristics. We define a *heuristic* as a value function used to initialize the state values before the first time an algorithm updates these values. In heuristic-guided algorithms (e.g., RTDP, LAO*), heuristics help avoid visiting irrelevant states. To guarantee convergence to an optimal policy, MDP solvers require a heuristic to be admissible. However, inadmissible heuristics tend to be more *informative* in practice. Informativeness often translates into a smaller number of explored states (and the associated memory savings) with reasonable sacrifices in policy optimality.

Determinization. Some of the most effective domain-independent MDP solvers known today are based on *determinizing* the domain D at hand, i.e. removing the uncertainty about D's action outcomes. For example, the *all-outcomes* determinization, for each action a with precondition c and

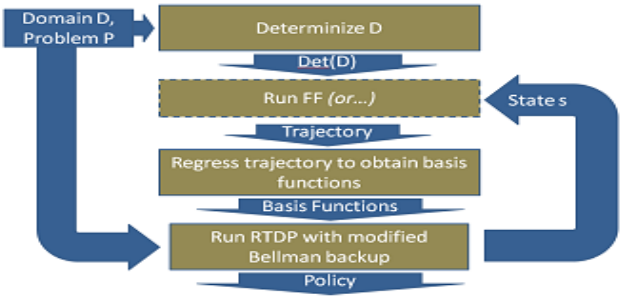

Figure 1: RETRASE.

outcomes $o_1, \ldots, o_n$ with probabilities $p_1, \ldots, p_n$, produces a set of deterministic actions $a_1, \ldots, a_n$, each with precondition c and effect o_i, yielding a classical domain D_d.

3 BASIS FUNCTIONS

Consider a *goal trajectory* $t = s, a_1(o_{j_1}), \ldots, a_n(o_{j_n})$, where s is the trajectory's starting state, each action $a_k(o_{j_k})$ represents the j_k-th outcome of the probabilistic action a_k, and s modified by t's action sequence is a goal state. Formally, a basis function b is the conjunction of literals resulting from regressing the goal conjunction through some trajectory t. Whenever all literals of b are present in a state s, we say that b *represents* s. To a first approximation, b is a precondition for the trajectory t that was regressed to obtain it. It is not t's precondition in the strict sense, since t is merely a sequence of actions' probabilistic outcomes, and hence trying to execute t from a state represented by b may fail. Nonetheless, the crucial properties of basis functions are that (1) like ordinary action preconditions, they typically represent many states of the problem but (2) unlike action preconditions, they *guarantee* the existence of a positive-probability trajectory to the goal from any state they represent. These properties allow basis functions to serve as a means to share goal reachability information among many states.

4 ReTrASE

As the first example of how basis functions can help in planning, we present an approximate MDP solver RETRASE (**Re**gressing **Tra**jectories for **A**pproximate **S**tate **E**valuation). This algorithm successfully circumvents an issue plaguing many VI-based techniques — the number of states whose values need to be estimated is often too large to store in memory.

On a high level, RETRASE explores the state space in the same manner as RTDP, but, instead of performing Bellman backups on states themselves, backups are performed over the basis functions that represent the visited states. For each basis function, RETRASE learns a weight that reflects the quality of the trajectory/-ies enabled by that basis function. Indeed, trajectories differ in their probability of reaching the goal, and the weights are a numerical characterization of these distinctions. A state's value may then be computed by aggregating the weights of the basis functions that represent it.

Operation of RETRASE is schematically depicted in Figure 1. At any moment during RETRASE's state space exploration, there are three kinds of states: ones that have been deemed dead ends, ones for which some representing basis functions are known, and the rest. When RETRASE encounters a state s of the third type, it applies a classical planner (e.g., FF [6]) to a determinized version of the domain starting from s. If no classical plan exists, the state is marked a dead end. If FF finds a plan, however, RETRASE via regression generates a basis function b holding in s. Thereafter, RETRASE learns a weight for b by modified Bellman backups while continuing to visit other states.

As it turns out, there exists a relationship between the weights of the basis functions representing a state and that state's value. The relationship is too expensive to compute exactly but can be approximated by the minimum of the state's representing basis functions' weights. Critically, since a given basis function b represents many states, the information encoded in its weight gets automatically shared among all of them. Thus, b's weight helps us approximately evaluate many states, even those that RETRASE has not touched yet. As a result, the number of basis functions needed to get a value function approximation (and hence a policy derived from it in the usual decision-theoretic way) is in practice much smaller than the number of states themselves. As the experiments demonstrate, this approximation gives RETRASE both a vast advantage in memory consumption over planners like RTDP and a policy quality advantage over IPPC winners on many benchmark domains.

5 GOTH

Reuse of information due to basis functions can bring not just memory savings as in RETRASE but time savings as well. The MDP heuristic function named GOTH, **G**eneralization **O**f **T**rajectories **H**euristic, that we present next uses full-fledged classical planners on the all-outcome determinization of the domain at hand to produce heuristic state values. Naively calling a deterministic planner from each state that needs evaluation would be prohibitively expensive, and the main idea underlying GOTH is to do such invocations from only a few states and generalize the experience to the rest.

The trick of discarding the probabilistic aspects of an MDP for the purpose of efficiently computing a heuristic is not new. One of the most successful MDP heuristics, the FF heuristic [6], which we will refer to as h_{FF}, dispenses not only with probabilities but also with delete effects of the actions to quickly evaluate a state. It sets the value of a state to be the length of a classical plan consisting of such relaxed actions from this state to the goal. While fast in practice, its ignorance of actions' negative consequences causes it to struggle on domains where effects of some actions clobber the effects of others, e.g. Machine Shop [11].

GOTH, on the other hand, operates on a non-relaxed version of the determinized domain at hand. When a planner that uses GOTH starts solving a problem, GOTH determinizes the domain and initializes a set of basis functions, originally empty. Whenever the MDP solver queries GOTH for a state value, the latter first checks if any basis functions in its set represent this state. If none do, it invokes a classical planner, which either returns a plan or concludes that the state is a dead end. If it returns a plan, GOTH regresses it to obtain a basis function, sets the weight of the basis function to be the cost of the regressed, stores the basis function and its weight for future use, and returns the plan's cost as the state value.

If by the time GOTH evaluates a given state it does already "know" a few basis functions that represent it, this effectively means that GOTH is aware of several trajectories from this state to the goal, since each basis function is a certificate of a goal trajectory. Therefore, GOTH lets the value of the state be the cost of the cheapest such trajectory, i.e. the smallest

weight of any known basis function that represents it. The resulting heuristic value is not always admissible, as the classical planner may be suboptimal and because GOTH is not aware of *all* basis functions (and hence goal trajectories) in the domain. However, GOTH empirically proves to be significantly more informative than h_{FF}, often causing the MDP solver to be both faster and more memory-efficient.

Note the different roles of generalization in GOTH and RETRASE. In the former case, it serves primarily to save time whereas in the latter — to save memory. In addition, GOTH does not *learn* state values, merely providing an initial estimate for them.

6 SIXTHSENSE

As described, generalization schemes of RETRASE and GOTH contain a large caveat. By definition, basis functions support information transfer only between states with goal trajectories. However, probabilistically interesting [10] MDPs also contain states with no such trajectories, dead ends. While *explicit* dead ends, i.e. states with no applicable actions, are easy to identify, *implicit* dead ends are not because they have successor states.

To approach the problem of quickly and reliably recognizing dead ends, we note that both in real life and in benchmark IPPC domains, most dead ends can often be explained by only a few "reasons". E.g., in the Drive domain of IPPC-06, the agent can't achieve the goal (and is hence in a dead end) if its car has crashed. If we knew these explanations we could swiftly conclude a state as a dead end if any of them apply in it. In fact, we could do such checks on the fly and avoid caching dead ends, thereby saving memory.

We propose a novel machine learning algorithm, SIXTHSENSE, to discover these explanations. It can act as a submodule of any existing MDP solver and efficiently and soundly tell its owner whether a given state is a dead end. To do so, it discovers *nogoods*, conjunctions of literals that guarantee *non-existence* of a goal trajectory from the states they represent. Semantically, nogoods are opposites of basis functions, which guarantee *existence* of such trajectories. Our procedure for learning nogoods follows the generate-and-test scheme and uses a small number of basis functions and dead ends (both discovered in a standard way like running a deterministic planner from a few states). It is fast but fairly involved, and here we merely outline its main ideas.

Generate step. Our foundational insight is that a literal conjunction is a nogood if and only if it *defeats* all basis functions in a problem, i.e. contains a negation of some literal in each basis function. Thus, finding a nogood means finding such a defeating literal conjunction. Since checking the candidate against all basis functions is infeasible, to construct it we sample literals according to their frequencies in the training dead ends (these statistics are intuitively indicative of the literals in nogoods), aiming to defeat the few (100-200) basis functions we have. This produces a valid nogood with a high chance but not always, and we do a test to verify it.

Test step. For verification, we form a *superstate* by conjoining the candidate with the negation of the goal and all literals whose negation is not contained in the resulting conjunction, and expand the planning graph [2] starting from this superstate. If the planning graph fails to achieve all the goal literals, or fails to resolve all mutexes among them, the candidate is considered to be a nogood.

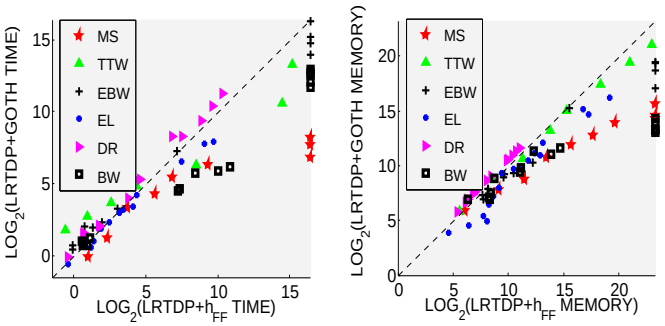

Figure 2: The big picture: GOTH has a significant advantage on large problems. (Note that the axes are on the Log scale.)

The procedure is provably sound, requires little training data in practice, and, with a few additional optimizations, finds nogoods that account for most dead ends in a problem.

7 EXPERIMENTAL RESULTS

RETRASE. Our experiments explored two important aspects of RETRASE – (1) scalability, to ensure that the positive effect of generalization is as strong as we predicted, and (2) quality of solutions in complex domains. We ran RETRASE on six probabilistically interesting hard 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. The experiments were conducted under the restrictions resembling those of IPPC.

RETRASE's scalability proved far better than that of heuristically guided optimal or suboptimal planners. E.g., LRTDP with h_{FF} ran out of memory on problems 8, 9, and 10 of TTW-08, whereas RETRASE solved them easily.

On five out of six domains, RETRASE dominated the IPPC participants in solution quality as well, i.e. its success rate was higher than the IPPC winner on that domain, and showed average performance on Elevators. RETRASE's advantage is especially impressive on EBW-06 because it grows with the complexity of the problem . A more complete presentation of the results is in [7].

GOTH. The purpose of the experiments with GOTH was to compare it in terms of informativeness and overall value to MDP solvers against other heuristics, as well as to show that without generalization, GOTH would be impractical. In our experience, h_{FF} is among the most informative existing MDP heuristics, and we used it as the benchmark. We ran both heuristics in combination with LRTDP, denoted as LRTDP+h_{FF} and LRTDP+GOTH on the same domains as RETRASE but excluding EBW-06, and Machine Shop, a domain in which actions' effects have a particularly adversarial structure for h_{FF}.

The plots in Figure 2 provide the big picture of the comparison. For each problem we tried, they contain a point whose coordinates are the logarithms of the amount of time/memory that LRTDP+RETRASE 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 Figure 2 extend to $\log_2(86400)$, the logarithm of the time cutoff (86400s, 24 hours) that we used. Similarly, the axes of the memory plot reach $\log_2(10000000)$, 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 86400s 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 GOTH enjoys a comfortable advantage on most large problems, and in terms of memory, this advantage extends to many medium-sized and small problems as well.

In another experiment, we turned off generalization and ran LRTDP+GOTH on several problems. As we expected, LRTDP+GOTH without generalization is 30-40 times slower than with it, and hence it is generalization that accounts for GOTH's successful operation.

SIXTHSENSE. To understand the advantages of using SIXTHSENSE, we divided the existing MDP solvers into two groups by their treatment of dead ends. The first group, which we denote as "Fast but Insensitive", tries to recognize implicit dead ends via a fast but unreliable means. E.g., LRTDP+h_{FF} runs h_{FF} on the newly encountered state; h_{FF} is fast to compute but, if the state is an implicit dead end it will often nonetheless find a relaxed plan, and thus will make the state look like a non-dead end to LRTDP. The second group, "Sensitive but Slow", uses more sophisticated but computationally expensive methods for dead end recognition. RFF, for instance, calls a deterministic planner on each state. Doing so is costly but if the classical planner fails to find a plan, the state is almost certainly a dead end. We run LRTDP+GOTH to simulate the behavior of this group.

Our experiments with LRTDP+h_{FF} and LRTDP+GOTH on domains with one nogood (Drive) and several nogoods (EBW) show that SIXTHSENSE helps both kinds of planners to reduce planning time sometimes by as much as a factor of 2 and used memory by as much as 10 times. These numbers are not the limit of SIXTHSENSE's performance and depend largely on what fraction of reachable state space consists of implicit dead ends in current IPPC benchmarks. However, the mechanism by which the two groups benefit from SIXTHSENSE are different. For the "Fast but Insensitive", SIXTHSENSE greatly increases recognition accuracy. As an upshot, LRTDP+h_{FF} hardly ever wastes time exploring the successors of implicit dead ends, something it does a lot without SIXTHSENSE; moreover, since these successors don't need to be stored anymore, it also saves memory. For the "Sensitive but Slow", nogoods bring savings primarily by sharing dead-endness information across many states and preempting unnecessary expensive state analysis, i.e. their role is similar to basis functions.

8 DISCUSSION

The main research direction that will increase the demonstrated potential of generalization is casting it in first-order logic. Many IPPC domains contain repetitive structure (e.g., many blocks can be acted upon in the same way in EBW) that can be succinctly described in first-order logic, and a lot of structure that *cannot* be concisely expressed propositionally.

Other fruitful ideas include developing a richer theory of basis functions. For instance, our current knowledge does not tell us how many basis functions are enough for what kind of approximations. It is also interesting to study how the use of basis changes the computational complexity of existing MDPs. Indeed, solving MDPs is known to be *PSPACE*-complete; however, a large fraction of the complexity may be due to the difficulty in finding the basis functions. If a human already knows the problem structure and is willing to specify it, then learning a policy with it may not be that hard.

9 RELATED WORK

Perhaps the most systematic attempt to extract MDP problem structure has been explanation-based learning (EBL) [4]. However, EBL systems have tended to suffer from the proliferation of rules they discovered during problem solving, which significantly slowed them down. This does not happen with the algorithms we presented. The idea of discovering basis functions by regression was originally described in [5] but there they were only used for state space compactification (similarly in spirit to RETRASE but vastly different in detail). For work more specifically related to RETRASE, GOTH, and SIXTHSENSE, we refer the reader to the corresponding sections of [7], [8], and [9].

10 CONCLUSION

This dissertation concentrates on exploiting problem structure in the form of basis functions to dramatically increase the scalability of methods for solving MDPs. We have showed three examples of how basis functions, used in their own right (as in RETRASE and GOTH) or as a means to extract even richer regularities (as in SIXTHSENSE), can generalize knowledge across different parts of an MDP's state space. The experiments demonstrate basis functions to indeed make solving MDPs much more efficient than before, and coming up with their first-order logic analogue promises to extend the potential of this idea much further.

References

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

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

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

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

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

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

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

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

[9] A. Kolobov, Mausam, and D. Weld. SixthSense: Fast and reliable recognition of dead ends in MDPs. In *'Under Review*, 2010.

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

[11] Mausam, P. Bertoli, and D. Weld. A hybridized planner for stochastic domains. In *IJCAI'07*, 2007.