Classical Planning in MDP Heuristics: with a Little Help from Generalization

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

Abstract

Heuristic functions make MDP solvers practical by reducing their time and memory requirements. Some of the most effective heuristics (e.g., the FF heuristic function) first determinize the MDP and then solve a relaxation of the resulting classical planning problem (e.g., by ignoring delete effects). While these heuristic functions are fast to compute, they frequently yield overly optimistic value estimates. It is natural to wonder, then, whether the improved estimates of using a full classical planner on the (non-relaxed) determinized domain will provide enough gains to compensate for the vastly increased cost of computation.

This paper shows that the answer is "No and Yes". If one uses a full classical planner in the obvious way, the cost of the heuristic function's computation outweighs the benefits. However, we show that one can make the idea practical by *generalizing* the results of classical planning successes and failures. Specifically, we introduce a novel heuristic function called GOTH that amortizes the cost of classical planning by 1) extracting basis functions from the plans discovered during heuristic computation, 2) using these basis functions to generalize the heuristic value of one state to cover many others, and 3) thus invoking the classical planner many fewer times than there are states. Experiments show that GOTH can provide vast time and memory savings compared to the FF heuristic function — especially on large problems.

INTRODUCTION

Heuristic functions, or heuristics, for short, are a popular means of reducing space and memory requirements of state space search-based probabilistic planning algorithms. In MDP solvers guided by heuristic functions, e.g. LRTDP [1] and LAO* [7], heuristics help avoid visiting many states (and memoizing corresponding state-value pairs) that are not part of the final policy.

While there are many ways of constructing a good heuristic for probabilistic domains, some of the most effective ones are derived from classical planning analogs. A notable example is the FF heuristic [8], denoted h_{FF} in this paper, which calculates the value of a state in three conceptual steps: (1) *determinizing* the probabilistic MDP actions into a set of classical actions, (2) relaxing the domain further by eliminating the *delete* lists of all actions, and (3) finding the cost of the cheapest sequence of these modified actions to the goal using a relaxed planning graph. This cost is taken as the heuristic value of the state in the original probabilistic problem. Although efficiently computable and quite informative, it is liable to highly underestimate the state's true value because of multiple levels of relaxation: once due to determinization and once more due to ignoring delete effects.

On the other hand, a lot of promise has been shown recently by several probabilistic planners that solve *full* (non-relaxed) determinizations, e.g., FF-Replan [17], HMDPP [9], and others. It is natural to wonder, then, whether the improved heuristic estimates of using a full classical planner on the non-relaxed determinized domain would provide enough gains to compensate for the potentially increased cost of heuristic computation.

As we show in this paper, the answer is "No and Yes". We propose a new heuristic called GOTH (**G**eneralization **O**f **T**rajectories **H**euristic), 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 novelty of our work is in showing that there is a way to amortize these expensive planner calls by generalizing the resulting heuristic values to provide guidance on similar states. We adapt the idea of generalization from a recent planner, ReTrASE [10], although there it is used in a somewhat ad hoc search procedure, whereas we employ it for a heuristic computation that guides a principled decision-theoretic search algorithm.

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. GOTH performs extremely well, especially on large problems. The rest of the paper explains this idea in more detail, focusing on the generalization process. Specifically, we make the following contributions:

- We describe the approach of using non-relaxed determinization for computing the heuristic. With its naive implementation we obtain much more informative heuristic values but at a large performance loss.
- We apply a generalization procedure to amortize the classical planner invocations leading to a very efficient procedure for heuristic computation. We implement it over

miniGPT's version of labeled RTDP [2], and empirically demonstrate that this new approach can be hundreds of times faster than the naive one with very little sacrifice in heuristic quality. Thus, we show generalization to be key to GOTH's time efficiency.

- We experimentally compare our GOTH implementation against h_{FF}, a state of the art heuristic for probabilistic planning. Our results show that for large problems in five out of six benchmark domains we massively outperform h_{FF} in terms of memory requirements, attesting to GOTH's higher informativeness, as well as time efficiency. We find that LRTDP+h_{FF} exhausts memory on several problems that LRTDP+GOTH is able to solve easily. Additionally, our solution quality is never worse and often better than LRTDP+h_{FF}.

BACKGROUND

Markov Decision Processes (MDPs). In this paper, we focus on probabilistic planning problems that are 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 fact that the total action cost is accumulated over a finite-length action sequence whose length is unknown. In this paper, we assume that all actions have conjunctive preconditions, since the disjunctive ones can be compiled away, with the number of actions in the resulting domain increasing linearly in the number of disjuncts.

In factored MDPs, each state is represented as a conjunction of values of the domain variables. 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 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), that 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 using Bellman backups. A popular variant, LRTDP, adds a termination condition to RTDP by labeling those states whose values have converged as 'solved' [1]. 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*.

Heuristic Functions. We define a *heuristic function*, hereafter termed simply as *heuristic*, as a value function used to initialize the state values before the first time an algorithm updates these values. In heuristic-guided algorithms, heuristics help avoid visiting irrelevant states. To guarantee convergence to an optimal policy, MDP solvers require a heuristic to be admissible, i.e. to never overestimate the optimal value of a state (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 strive to obtain an informative heuristic even at the cost of admissibility, and adopt the number of states visited by a planner under the guidance of a heuristic as the measure of that heuristic's informativeness.

Determinization. Some of the most effective domain-independent heuristics known today are based on *determinizing* the probabilistic domain at hand. Determinizing domain D removes the uncertainty about D's action outcomes in a variety of ways. For example, the *all-outcomes* determinization, for each action a with precondition c and outcomes $o_1, \ldots, o_n$ with respective 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. 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 is generally NP-hard). For instance, h_{FF} ignores the delete effects of all actions in D_d and attempts to find the cost of the cheapest solution to this new relaxed problem.

GOTH HEURISTIC

Given a problem P over a probabilistic domain D, the 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 [17]. 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 a heuristic initialization, GOTH first checks if it is an *explicit dead end*, i.e. has no actions applicable in it. This check is in place for efficiency, since GOTH should not spend time on them.

For state s that isn't 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 $DetPlan$, and setting a timeout (in our setup, 25 seconds). If s is an *implicit dead end* (i.e., has actions applicable in it but no plan to the goal), $DetPlan$ either quickly proves this or unsuccessfully searches for a plan until the timeout. In either case, it returns without a plan, at which

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., 1000000
10: else if a nogood holds in $s$ then
11:     return a large penalty // e.g., 1000000
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., 1000000
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
```

point s is presumed to be a dead end and assigned a very high value. 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. In rare cases, $DetPlan$ 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 haven't 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 following insight. Regressing a successful deterministic plan in domain D_d yields a set of literal conjunctions with an important property: each such conjunction is a precondition for the plan suffix that was regressed to generate it. We call these conjunctions *basis functions*, and define the *weight* of a basis function to be the cost of the plan it enables. *Crucially, every deterministic plan in D_d corresponds to a positive-probability trajectory in the original domain D*; therefore, a basis function is a certificate of such a trajectory. Every state subsumed by a given basis function is thus proved to have a possible trajectory to the goal.

We make this process concrete in the pseudocode of Algorithm 1. Whenever GOTH computes a deterministic plan, it regresses it and caches the resulting basis functions with associated weights. When GOTH encounters a new state s, it minimizes over the weights of all basis functions stored so far that subsume 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, by the time state s needs to be evaluated GOTH may have no basis functions that subsume it. 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. For instance, LPG [5], 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. Thus, the basis function weight from any given invocation may be irrepresentative 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.

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.

Fortunately, 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. For this purpose, we compute conjunctions of literals called *nogoods* with the property that all states subsumed by a nogood are dead-ends. Just like basis functions guarantee the existence of a goal trajectory from any states they subsume, nogoods guarantee its non-existence. The algorithm for nogood construction and deciding when to perform it, SixthSense [11], is rather involved theoretically but very fast. SixthSense includes a scheduler that decides when learning should be attempted. Crucially, when the decision has been made (situation represented in line 30 of Algorithm 1), the technique makes use of the basis functions and implicit dead ends discovered so far, utilizing both as training data to induce nogoods (details are abstracted away in line 31). The produced nogoods are sound [11], i.e. all the states each of them subsumes are implicit dead ends. With nogoods available, deciding whether a state is a dead end is as simple as checking whether any of the known nogoods subsume it (lines 8-9 of Algorithm 1). Only if none do may deterministic planning be necessary to answer the question. Experiments indicate [11] that SixthSense significantly reduces the amount of resources GOTH uses. For instance, when GOTH is used with LRTDP, SixthSense can help reduce the running time and memory use (since most implicit dead ends don't need to be memoized anymore) of the combination by 50% and more.

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 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 function 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, compared to h_{FF}.

Timewise, GOTH's performance is largely determined 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 function. Although faster solutions are possible for this pattern-matching problem, all that we are aware of (e.g., [4]) pay for the increase in speed with degraded memory performance.

Theoretical properties. Two especially noteworthy 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, just like h_{FF}, GOTH is inadmissible, but for different reasons. 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 a 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}.

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 [2], 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 [5]. 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. We tested GOTH and h_{FF} as a 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: Machine Shop [13], 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 weren't able to test on them. Additionally, we perform a brief comparison of LRTDP+GOTH against ReTrASE [10] and FF-Replan [17], since these share some insights with GOTH. In all experiments except measuring the effect of generalization, the planners had a 24-hour limit to solve each problem.

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 massively outperforms LRTDP+h_{FF}.

We start the comparison by looking at a domain whose structure is especially inconvenient for h_{FF}. The Machine Shop domain involves 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 doesn't distinguish good and bad action sequences as a result. Machine Shop has no dead ends.

Figures 1 and 2 show the speed and memory performance of LRTDP equipped with the two heuristics. As implied by the previous 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}

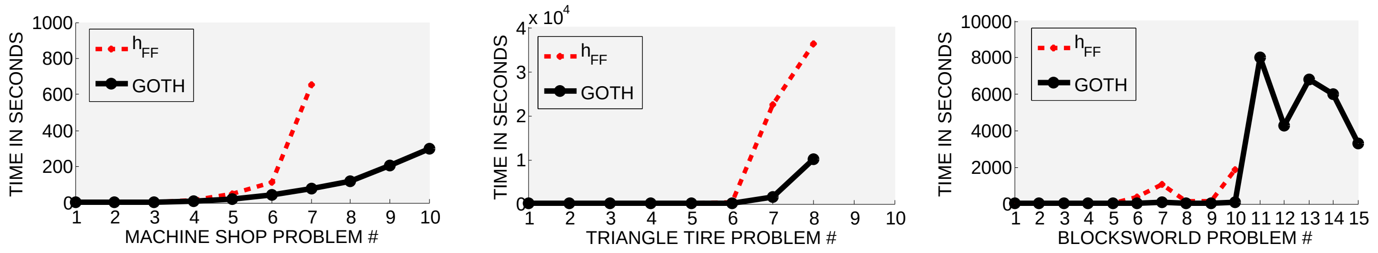


Figure 1: GOTH outperforms h_{FF} on Machine Shop, Triangle Tireworld, and Blocksworld by a large margin both in speed...

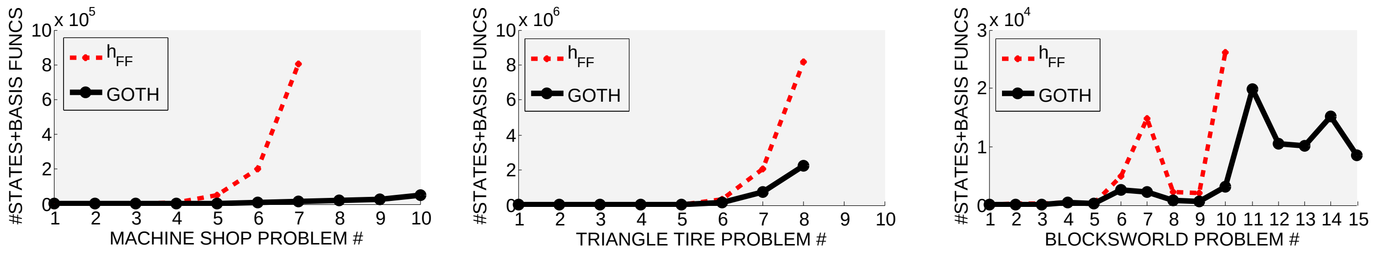


Figure 2: ... and in memory

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 the 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.

The Triangle Tireworld domain, unlike Machine Shop, doesn't have structure that is particularly adversarial for h_{FF}. However, LRTDP+GOTH noticeably outperforms LRTDP+h_{FF} on it too, as Figures 1 and 2 indicate. Nonetheless, neither heuristic saves enough memory to let LRTDP solve past problem 8.

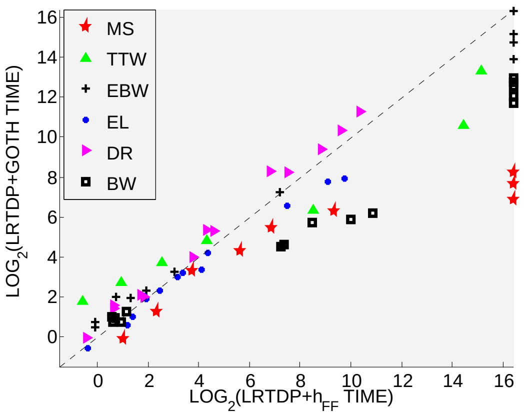
The results on Exploding Blocks World (EBW) are similar to those on Triangle Tireworld, where the LRTDP+GOTH's more economical memory consumption eventually translates to a speed advantage. Importantly, however, on several EBW problems LRTDP+GOTH is superior to LRTDP+h_{FF} in a more illustrative way: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

The Drive domain is small, and using GOTH on it may be an overkill. 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.

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. Figures 1 and 2 shows the performance on Blocksworld as an example. Classical planners in our portfolio cope with determinized versions of these domains very quickly, and generalization ensures that the obtained heuristic values are spread over many states. Similar to the situation on 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.

Figure 3 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 Figure 3 extend to $\log_2(86400)$, the logarithm of the time cutoff (86400s, i.e. 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, 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} can't.

Why does GOTH's and h_{FF}'s comparative performance differ from domain to domain? For an 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 ad-



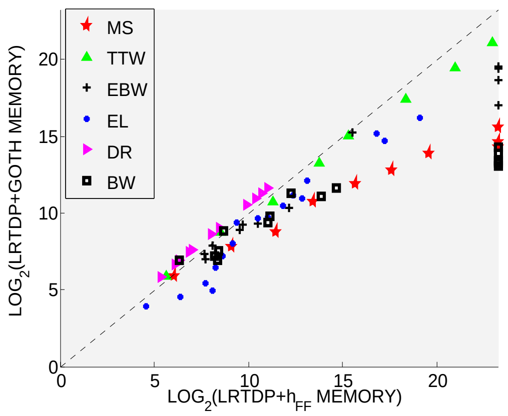


Figure 3: The big picture: GOTH provides a significant advantage on large problems. (Note that the axes are on the Log scale.)

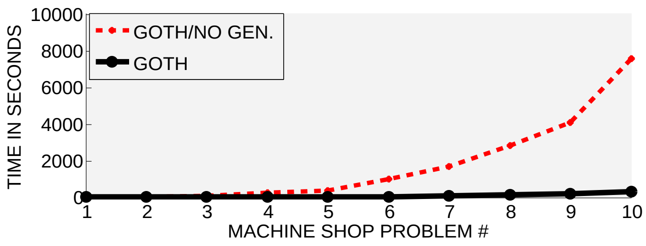


Figure 4: GOTH is much faster with generalization than without.

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

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}.

vantage 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 $1.71\times$ 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. A central hypothesis of this paper is that generalization is vital for making GOTH computationally feasible. To test it and measure the importance of basis functions 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 similar to the strategy of FF-Replan, 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 FF-Replan and GOTH further in the next section.)

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 (Figure 4) 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. While we don't know if the trend holds on the largest problems of most domains we tried, even if it is reversed the slowness of GOTH without generalization makes its use unjustifiable in practice.

One may wonder whether generalization can also benefit h_{FF} the way it helped GOTH. While we haven't 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 ReTrASE. Superficially, ReTrASE extracts and uses basis functions in a way similar to GOTH.

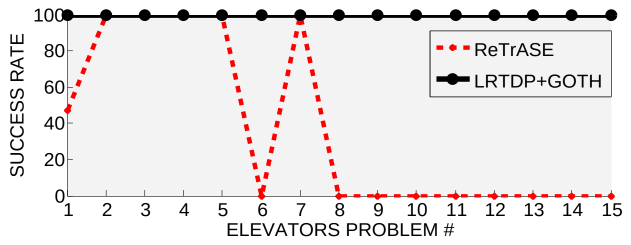


Figure 5: LRTDP+GOTH vastly outperforms ReTrASE on Elevators.

The major difference lies in the fact that ReTrASE tries to learn weights for the basis functions, whereas GOTH, being only a heuristic, employs basis functions to initialize state values and lets a conventional MDP solver improve on these values. In practice, this discrepancy translates to ReTrASE's learning procedure providing very few quality guarantees. While it is very memory-efficient on many hard problems, the solutions are poor on some domains with rather simple structure, e.g. Elevators from IPPC-06 [10]. In contrast, GOTH admits the use of conventional MDP solvers with strong theoretical machinery, making the outcome of its application more predictable. In particular, LRTDP+GOTH achieves a 100% success rate on all 15 Elevators problems (Figure 5) and takes at most 5 minutes per problem. This is not to say, however, that LRTDP+GOTH generally outmatches ReTrASE. For example, while LRTDP+GOTH achieves a 100% success rate on the first 8 problems of Triangle Tire, ReTrASE performs equally well on the first 8 problems but, unlike LRTDP+GOTH, can also solve problems 9 and 10. Thus, GOTH's use of basis functions yields qualitatively different results than ReTrASE's.

Comparison against FF-Replan. One can also find similarities between the techniques used by GOTH and FF-Replan. Indeed, both employ deterministic planners, FF-Replan — 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. FF-Replan, on the other hand, ignores them entirely. Due to this discrepancy, performance of FF-Replan and a planner guided by GOTH is typically vastly distinct. For instance, FF-Replan is faster than most decision-theoretic planners. On the other hand, FF-Replan has difficulty dealing with probabilistic subtleties and is known to come up with low success rate policies [12] on domains that contain them, e.g., Triangle Tireworld. LRTDP+GOTH can handle such domains much better, as our experiments demonstrate.

We conclude by stressing that, since GOTH is not a full planner, any performance comparison between it and various MDP solvers is inconclusive without the specification of and highly dependent upon the planner that uses GOTH.

DISCUSSION

Promise shown by GOTH indicates several directions for further development. The experiments have demonstrated that GOTH generally performs well but its advantage in informativeness isn't always sufficient to secure an advantage in speed. GOTH's speed, in turn, depends critically on how fast the deterministic planners from its portfolio are on the deterministic version of the domain at hand. Therefore, one way this issue can be alleviated is by adding more 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, and a prominent candidate for inclusion is LAMA [14], the winner of the deterministic track of IPC-2008. In addition, using a reasonably-sized portfolio of planners may help reduce the variance of the time it takes to arrive at a heuristic estimate.

Another direction is experimenting with domain determinizations GOTH could rely on. One alternative is proposed by the authors of HMDPP [9] and described in Related Work. Its use could improve GOTH's informativeness further, and possibly 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.

An entirely different set of ideas suggested by GOTH concerns applicability of generalization in other planning algorithms. We firmly believe that generalization has the ability to enhance many existing probabilistic planning techniques as well as inspire new ones. As an example, note that FF-Replan could benefit from generalization 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 FF-Replan" 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 FF-Replan since it is very fast as is, FF-Replan'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 FF-HOP [18], an MDP solver that has great potential but is somewhat slow in its current form. In fact, it appears that *virtually any* algorithm for solving 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. We hope to verify this conjecture in the future.

RELATED WORK

The use of determinization for solving MDPs was inspired by advances in classical planning, most notably the FF solver [8]. The practicality of the new technique was demonstrated by FF-Replan [17] 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 [16]. Unlike GOTH, they normally use deterministic planners to *learn* the state or action values, not just to initialize their values heuristically. As a consequence, they invoke FF many more times than we do. This forces them to avoid all-outcome determinization as invoking FF would be too costly otherwise.

In spirit, GOTH's strategy of extracting useful state information in the form of basis functions is related to explanation-based learning (EBL) [3]. However, EBL systems suffer from accumulating too much of such information, whereas GOTH doesn't.

To a large degree, the FF planner owes its performance to h_{FF} [8]. LRTDP [1] and HMDPP [9] 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.

Several algorithms generate basis functions by regression like we do, [6], [15], and [10] to name a few. However, the role of basis functions in them is entirely different. In these methods, basis functions serve to map the planning problems to smaller parameter spaces consisting of basis function weights. Parameter learning in such transformed spaces is usually approximate and gives few theoretical guarantees (see, for instance, [10]). In GOTH, basis functions are used to generalize heuristic values over multiple states and thereby to avoid invoking the classical planner too many times. Importantly, however, the parameter space in which learning takes place is unchanged — it is still the set of state values. We can therefore use conventional techniques like LRTDP in conjunction with GOTH that give substantial predictability of the solution quality. GOTH achieves the reduction in the number of required parameters through the increased informativeness of initial heuristic estimates, not through parameter space transformation.

CONCLUSION

We have proposed GOTH, a new heuristic that uses full-fledged deterministic planners to solve MDP determinizations. Although invoking a classical solver naively is too expensive to be practical, we show that one may amortize this cost by generalizing the resulting heuristic values to cover many states. When a deterministic trajectory to the goal is found, GOTH regresses the trajectory to calculate basis functions summarizing where such a trajectory is feasible. GOTH's use of basis functions greatly reduces the number of times the deterministic planner is called and renders our idea practical. While, like h_{FF}, the resulting heuristic is inadmissible, it usually gives more informative state value estimates than h_{FF} and provides significant memory savings to the MDP solvers. The experiments show that GOTH outperforms h_{FF} in time and memory on five out of six domains. We also demonstrate generalization to be the key idea enabling GOTH to compete with h_{FF} in terms of speed. We believe that GOTH's notion of generalization has considerable potential to improve other planning techniques as well.

Acknowledgments. We would like to thank Rao Kambhampati, William Cushing, the anonymous reviewers, and our colleagues at the University of Washington AI group for their insightful comments. This work was supported by ONR grants N000140610147 and N000140910051 and the WRF/TJ Cable Professorship.

References

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

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

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

[4] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Artificial Intelligence*, volume 19, pages 17–37, 1982.

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

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

[7] E. Hansen and S. Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. In *Artificial Intelligence*, pages 129(1–2):35–62, 2001.

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

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

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

[11] A. Kolobov, Mausam, and D. Weld. SixthSense: Fast and reliable recognition of dead ends in MDPs. In submission, 2010.

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

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

[14] S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *AAAI'08*, 2008.

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

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

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

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