LRTDP vs. UCT for Online Probabilistic Planning

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

Abstract

UCT, the premier method for solving games such as Go, is also becoming the dominant algorithm for probabilistic planning. Out of the five solvers at the International Probabilistic Planning Competition (IPPC) 2011, four were based on the UCT algorithm. However, while a UCT-based planner, PROST, won the contest, an LRTDP-based system, GLUTTON, came in a close second, outperforming other systems derived from UCT. These results raise a question: what are the strengths and weaknesses of LRTDP and UCT in practice?

This paper starts answering this question by contrasting the two approaches in the context of finite-horizon MDPs. We demonstrate that in such scenarios, UCT's lack of a sound termination condition is a serious practical disadvantage. In order to handle an MDP with a large finite horizon under a time constraint, UCT forces an expert to guess a non-myopic lookahead value for which it should be able to converge on the encountered states. Mistakes in setting this parameter can greatly hurt UCT's performance. In contrast, LRTDP's convergence criterion allows for an iterative deepening strategy. Using this strategy, LRTDP automatically finds the largest lookahead value feasible under the given time constraint. As a result, LRTDP has better performance and stronger theoretical properties. We present an online version of GLUTTON, named GOURMAND, that illustrates this analysis and outperforms PROST on the set of IPPC-2011 problems.

Introduction

The introduction of Monte-Carlo based tree search and the UCT algorithm that exemplifies it (Kocsis and Szepesvári 2006) has significantly advanced several fields of AI. Among other achievements, these method have drastically improved machines' ability to play Go (Gelly and Silver 2008) and Solitaire (Bjarnason, Fern, and Tadepalli 2009). Recently, UCT has received close attention from the probabilistic planning community as well. Out of the five solvers at the International Probabilistic Planning Competition (IPPC) 2011, four were based on UCT, including the winner, PROST (Keller and Eyerich 2012). UCT's success is at least partly attributable to its model-free nature — it does not need to know state transition probabilities explicitly in order to estimate state values. Indeed, in games such probabilities are typically unknown, as they depend on the behavior of the opponent. In planning, they may be available, but if each state, on average, allows transitions to many others, the probabilities cannot be explicitly used to compute Bellman backups efficiently, as many conventional, Value Iteration-based (Bellman 1957) MDP algorithms require. In the meantime, a large average number of transitions per state, i.e., a high *branching factor*, is characteristic of many interesting planning problems, e.g. the benchmarks at IPPC-2011. Therefore, UCT seems ideally suited for such complicated planning scenarios as well as games.

At the same time, the runner-up at IPPC-2011, GLUTTON (Kolobov et al. 2012), was the only planner built around an algorithm different from UCT. GLUTTON's core is an iterative-deepening version of LRTDP (Bonet and Geffner 2003), an optimal heuristic search algorithm. GLUTTON circumvents the issue of large number of possible state transitions by having LRTDP subsample the transition function, and uses a number of other optimizations to make this idea work. Besides the use of UCT and LRTDP respectively, another major difference between PROST and GLUTTON was *how* they used these algorithms. While PROST employed UCT in an *online* manner, interleaving planning and execution, GLUTTON constructed policies *offline*. GLUTTON's performance is very close to UCT-based PROST's, and vastly better than that of all other IPPC-2011 competitors (Sanner 2011), which also use UCT.

Critically analyzing these results, in this paper we ask: is the nascent trend of using UCT as the dominant probabilistic planning algorithm justified? Which, of UCT and LRTDP, performs better if both are used online? Does LRTDP have any practical advantages over UCT in online mode?

To start answering these questions, we compare the suitability of UCT and LRTDP to solving finite-horizon MDPs under time constraints. A finite-horizon MDP $M_{s_0}(H)$ with horizon H and initial state s_0 represents a probabilistic planning problem in which an agent is trying to maximize the total expected reward from executing a sequence of H actions starting at s_0. Finite-horizon MDPs with large horizons are useful for modeling processes that are in fact nearly infinite, such as controlling traffic lights on a road grid, managing the delivery of packages that keep arriving at a distribution center, etc. Incidentally, many such scenarios impose time constraints on policy computation and execution. E.g., in the package delivery example, the planner may need to produce a new policy for how to group packages for delivery at least every half an hour.

Settings as above naturally lend themselves to online planning, i. e., interleaving planning with execution. A practical way of solving a non-goal-oriented MDP online is to come up with a policy starting from the current state assuming the horizon of the problem is some fairly small value L, execute the first action of that policy, transition to a new state, and proceed this way until the process stops. It is easy to see that if L is equal to the number of remaining decision epochs, this approximation scheme yields an optimal policy for $M_{s_0}(H)$. However, solving a state (i.e., coming up with an optimal L-lookahead policy for it) for such a large L is typically infeasible, as the size of the reachable state space is generally exponential in L. However, barring pathological cases, we would like to solve each state for as large an L as possible under the given time constraint. Unfortunately, for a given problem and timeout, this "optimal" value of L is usually unknown a-priori.

In this paper, we claim that because of the difficulty of choosing L LRTDP is generally better suited for solving finite-horizon MDPs under time constraints than UCT. In particular, UCT does not have a convergence condition, making it hard to determine the time it takes it to converge for a given L and effectively forcing the practitioner to specify L based on a guess. In the meantime, a good L is heavily problem-dependent. Setting it too high will cause UCT to fail to solve the state completely for this lookahead and pick an action largely at random. Setting it too low may make UCT's behavior too myopic. On the other hand, as we demonstrate in this paper, LRTDP, thanks to its convergence condition, can determine a good lookahead value automatically via a reverse iterative deepening strategy of the kind used in GLUTTON. Moreover, if the time constraint is specified for executing the entire process, not on a per-epoch-basis, iterative deepening LRTDP allows for a strategy that distributes the available computation time among different decision epochs in the process in a problem-independent manner and without human intervention.

The contributions of this paper are the following:

- We analyze strengths and weaknesses of UCT and LRTDP when solving large finite-horizon MDPs under time constraints.
- We present a novel algorithm, GOURMAND, that exploits LRTDP's termination condition to find a good lookahead value automatically for the given time constraint and thus robustly solve finite-horizon MDP online.
- We compare the performance of GOURMAND, PROST, and GLUTTON, the exponents of online LRTDP, online UCT, and offline LRTDP respectively, on the set of all IPPC-2011 benchmarks. As the experimental results demonstrate, GOURMAND significantly outperforms the other two algorithms across these eighty diverse problems, indicating that online LRTDP makes for a very potent MDP solver.

Background

MDPs. Our analysis in this paper focuses on probabilistic planning problems modeled by finite-horizon MDPs with a start state, defined as tuples of the form $M_{s_0}(H) = \langle\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, s_0\rangle, H\rangle$ where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{T}$ is a transition function $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ that gives the probability of moving from s_i to s_j by executing a, $\mathcal{R}$ is a map $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that specifies action rewards, s_0 is the start state, and H is the number of decision epochs after which the process stops.

In this paper, we will use the concept of the *augmented state space* of $M(H)$, which is a set $\mathcal{S} \times \{0, \ldots, H\}$ of state-number of remaining decision epochs pairs. Solving $M(H)$ means finding a *policy*, i.e. a rule for selecting actions in augmented states, s.t. executing the actions recommended by the policy starting at the augmented initial state (s_0, H) results in accumulating the largest expected reward over H decision epochs.

Specifically, let a *value function* be any mapping $V : \mathcal{S} \times \{0, \ldots, H\} \to \mathbb{R}$, and let the *value function of policy* π be the mapping $V^\pi : \mathcal{S} \times \{0, \ldots, H\} \to \mathbb{R}$ that gives the expected reward from executing π starting at any augmented state (s, h) for h epochs to go till the end of the process, $h \leq H$. Ideally, we would like to find an optimal policy π^* closed with respect to s_0, i.e. a policy that specifies an action for every state reachable from s_0 via this policy, and value function V^* for all such states s obeys $V^*(s, h) = \max_\pi \{V^\pi(s, h)\}$ for $0 \leq h \leq H$.

As it turns out, for a given MDP V^* is unique and satisfies *Bellman equations* (Bellman 1957) for all $s \in \mathcal{S}$:

$$V^*(s, h) = \max_{a \in \mathcal{A}} \mathcal{R}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') V^*(s', h - 1)$$

for $1 \leq h \leq H$ and $V^*(s, 0) = 0$ otherwise.

WLOG, we assume the optimal action selection rule π^* to be deterministic, i.e. of the form $\pi^* : \mathcal{S} \times \{1, \ldots, H\} \to \mathcal{A}$, since for every finite-horizon MDP at least one optimal deterministic policy is guaranteed to exist (Puterman 1994). If V^* is known, a deterministic π^* can be derived from it via the optimality equation for all $1 \leq h \leq H$:

$$\pi^*(s, h) = \arg\max_{a \in \mathcal{A}} \mathcal{R}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') V^*(s', h - 1)$$

Solution Methods. The above equations suggest a dynamic programming-based way of finding an optimal policy, called Value Iteration (VI) (Bellman 1957). VI uses Bellman equations as an assignment operator, *Bellman backup*, to compute V^* in a bottom-up fashion for $h = 1, 2, \ldots, H$ and then derives π^* from it via the optimality equation.

A more efficient modification of VI are trial-based methods, which explore the state space by executing multiple trials from the initial state.

One popular trial-based method, LRTDP (Bonet and Geffner 2003), updates states its trials pass through with Bellman backups and, crucially, has a convergence condition that tells it when a state's value has stabilized.

Another powerful trial-based method is UCT. When executing a trial, in each state UCT picks an action based on its current quality estimate and an "exploration term". The exploration term forces UCT to choose actions that have been tried rarely in the past, even if their estimated quality is low. This strategy makes UCT suitable for some scenarios other MDP algorithms cannot handle (e.g., when the transition function is not known explicitly). However, it also extorts

a price — since UCT from time to time tries suboptimal actions, it has no reliable termination condition indicating when UCT is near convergence.

Online versus Offline Planning. Many planning algorithms, including VI, LRTDP, and UCT, allow offline planning mode, when the planner tries to find a complete π^* closed w.r.t. s_0 before executing it. In many cases, doing so is infeasible and unnecessary — the problem may have so many states that they cannot all be visited over the lifetime of the system, so finding π^* for them is a waste of time. E.g., over its lifetime a robot will ever find itself in only a small fraction of possible configurations. Such problems may be better solved *online*, i.e. by finding π^* or its approximation for the current state, executing the chosen action, and so on. In many scenarios, the MDP needs to be solved online and under time constraint. Ways of adapting MDP solvers to the online setting vary depending on the algorithm. This paper compares online versions of UCT and LRTDP.

Solving Finite-Horizon MDPs Under Time Constraints

As already mentioned, solving finite-horizon MDPs offline may not be a feasible or a worthwhile strategy. Generally, the number of states reachable from the initial state via a given policy by executing t actions grows exponentially with t (although in MDPs with finite state spaces it tapers off as t goes to infinity). Thus, the number of states for which the policy may need to be stored may exceed available memory even for moderate horizon values. Moreover, computing policy for all of them may be wasteful, since over its lifetime, a system such as a robot may be able to visit only a small fraction of the states reachable by any given policy.

Instead, consider solving finite-horizon MDPs online. In an online setting the planner decides on the best or near-optimal action in the current state, executes it, decides on an action in the state where it ends up, and so on. The advantage of this approach is the that planner spends much less resources on analyzing states it never visits — they are only analyzed as a side-effect of computing an action for a state it does visit. Many real-life scenarios conform to this setting. E.g., consider a robot trying to navigate an environment with a lot of people. Instead of computing a policy offline by considering the probabilities of people showing up in its path, it can decide on the direction in which to move until the next decision epoch, e.g., for 1 second, in order to avoid running into anyone. Executing the action will bring it to a different state, where it can repeat the decision process.

There are many possible ways of choosing an action in the current state s. In a finite-horizon MDP $M_{s_0}(H)$, a principled way to do so is to optimally solve MDP $M_s(L)$, a problem that is identical to the original one except for the start state, which is now s, and the horizon $L < H$, and then select action $\pi^*_L(s, L)$ recommended by its optimal policy. This action selection rule has the intuitive property that if the number L, which we call *lookahead*, is as large as the number of decision epochs remaining till the end of the process, an optimal policy resulting from it, an *L-lookahead policy*, is optimal for the original MDP. Note that this *does not* imply that as L approaches H, the quality of an optimal L-lookahead policy, as measured by its value function, *monotonically* approaches that of π^*_H. Indeed, one can construct pathological examples in which increasing lookahead up to a certain point results in policies of deteriorating quality. However, in many non-contrived real-life examples, such as robot navigation, traffic light grid control, or package delivery management, larger lookahead generally translates to a better policy. Thus, given a time constraint, obtaining the best approximation in practice according to this scheme requires answering the question: at each decision epoch t, $0 \leq t \leq H-1$, what is the largest lookahead value for which we can reasonably afford to compute the optimal action in the current state?

The answer to this question is influenced by the type of time constraint we are dealing with:

- Per-epoch constraint. In this case, the system is told how much time it can spend computing an action at each of the H decision epochs of the process.
- Per-process constraint. There is a final deadline for the execution to stop. In effect, the system is told the total amount of planning time T that it has; the system is then free to allocate this amount to different process epochs in an arbitrary way.

We consider the second scenario, since the solution to the first one is just a special case of it. In the next section, we discuss a method for determining a good lookahead value at each decision epoch automatically, given a per-process time constraint T.

The GOURMAND Algorithm

We now present an algorithm called GOURMAND, a solver for finite-horizon MDPs that demonstrates how the termination condition can help LRTDP find a good lookahead value automatically in an online setting. GOURMAND is analogous to PROST in that both use a version of a basic algorithm, LRTDP and UCT respectively, to choose an action in an augmented state (s, h) encountered at the t-th decision epoch of the process (t and h are related, since $t = H - h$) by trying to solve the state for some lookahead. However, while PROST needs an engineer to specify the value of the lookahead and the timeout to devote to choosing an action at epoch t, GOURMAND determines both of these values without human intervention. GOURMAND is also related to GLUTTON — they use the same version of LRTDP and engineering optimizations such as subsampling the transition function (Kolobov et al. 2012). A major difference between the two is the mode in which they use LRTDP. GLUTTON uses it in an offline fashion. As a result, when the time T allocated for solving the MDP runs out, GLUTTON may not have solved all the states reachable by its policy and has to resort to ad-hoc methods of action selection when it encounters such states during policy execution. GOURMAND does not have this difficulty — thanks to its online use of LRTDP and time allocation strategy, it makes an informed choice in any state where it ends up.

Algorithm 1 shows GOURMAND's pseudocode. Roughly, GOURMAND initially distributes the total timeout T equally among the H decision epochs. While choosing an action during the initial decision epoch (epoch 0), it estimates

1 **Input:** MDP $M_{s_0}(H)$, timeout T
2 **Output:** Actions for states encountered at epochs $0, \ldots, H-1$

3 // Running averages of the amount of time it takes to
4 // solve a state for lookahead L
5 $Ts_0 \leftarrow 0$;
6 $Ts_L \leftarrow \infty$ for all $L = 1, \ldots, H$;
7 $\overline{T} \leftarrow T$;
8 **function** GOURMAND(MDP $M_{s_0}(H)$, timeout T)
9 **begin**
10 $s \leftarrow s_0$;
11 **foreach** $t = 0, \ldots, H-1$ **do**
12 $T_t \leftarrow \frac{\overline{T}}{H-t}$;
13 **if** $t == 0$ **then**
14 Run LR^{2}TDP$(M_s(H), T_t)$
15 $\overline{T} \leftarrow \overline{T} - T_t$;
16 $\hat{L}_0 \leftarrow$ largest L for which $M_s(L)$ is solved;
17 **end**
18 **else**
19 $L_t \leftarrow$ largest L s.t. $Ts_L < T_t$;
20 $\hat{T}_t \leftarrow T_t + (T_t - Ts_{L_t})(H-t-1)$;
21 $\hat{L}_t \leftarrow L_t$;
22 **if** $Ts_{L_t+1} < \hat{T}_t$ *or* $Ts_{L_t+1} == \infty$ **then**
23 $\hat{L}_t \leftarrow L_t + 1$;
24 **end**
25 $t_{start} \leftarrow$ current time ;
26 Run LR^{2}TDP$(M_s(\hat{L}_t), \hat{T}_t)$;
27 $t_{end} \leftarrow$ current time ;
28 $SolutionTime \leftarrow t_{end} - t_{start}$;
29 $\overline{T} \leftarrow \overline{T} - SolutionTime$;
30 **end**
31 $Action_t \leftarrow \pi^*_{M_s(\hat{L}_t)}(s, \hat{L}_t)$;
32 $s \leftarrow$ execute $Action_t$ in s;
33 **end**
34 **end**

35 **function** *LR^{2}TDP*(MDP $M_{s'_0}(H')$, timeout T)
36 **begin**
37 $Ts_h \leftarrow 0$;
38 $t_{start} \leftarrow$ current time ;
39 $t_{end} \leftarrow$ current time ;
40 **foreach** $h = 1, \ldots, H'$ *or until time T runs out* **do**
41 Run LRTDP$_{FH}(M_{s'_0}(h), T - (t_{end} - t_{start}))$;
42 $t_{end} \leftarrow$ current time ;
43 $Ts_h \leftarrow$ update average with $(t_{end} - t_{start})$;
44 **end**
45 **end**

46 **function** LRTDP$_{FH}$(MDP $M_{s'_0}(h)$, timeout T)
47 **begin**
48 Convert $M_{s'_0}(h)$ into the equivalent goal-oriented MDP $M^h_{g\,s'_0}$, whose goals are all states of the form $(s, 0)$.
49 Run LRTDP($M^h_{g\,s'_0}$) until time T runs out, memoizing the values of all encountered augmented states
50 **end**

Algorithm 1: GOURMAND

how long solving a state takes for different lookahead values. During each subsequent epoch t, GOURMAND first divides up the total remaining time $\overline{T}$ for solving the problem equally among the remaining epochs, i.e. virtually allocates time $T_t = \frac{\overline{T}}{H-t}$ to each of them, including the current one. Then, using the previously obtained estimates for time it takes to solve for different lookahead values, GOURMAND determines the largest lookahead L_t for which it can almost certainly solve any remaining epoch if time T_t is allocated to it. Finally, GOURMAND checks whether in the current decision epoch, it could solve for an even larger lookahead $\hat{L}_t$ by taking away a small amount of computation time from future decision epochs while still guaranteeing that it can solve for lookahead L_t in each of them. If so, it solves the current epoch for lookahead $\hat{L}_t$, otherwise — for L_t. Proceeding this way lets GOURMAND adaptively pick a good lookahead value at each decision epoch, given the per-process time constraint T.

The key to GOURMAND's resource allocation strategy is the knowledge of how long it takes LRTDP to solve a state for lookahead values $L = 1, 2, \ldots, \hat{L} \leq H$ for some $\hat{L}$. To collect these data, GOURMAND uses a reverse iterative deepening LRTDP version called LR^{2}TDP, first introduced in GLUTTON (Kolobov et al. 2012). LR^{2}TDP's pseudocode is shown on lines 35 - 45 of Algorithm 1 for completeness. Its main idea is to arrive at a policy for an MDP $M_{s'_0}(H')$ by solving a sequence of MDPs $M_{s'_0}(1), M_{s'_0}(2), \ldots, M_{s'_0}(H')$, where values of states computed when solving $M_{s'_0}(h)$ are used to help compute state values for $M_{s'_0}(h+1)$. LR^{2}TDP has two advantages over a straightforward adaptation of LRTDP (Bonet and Geffner 2003) to finite-horizon MDPs, which we call LRTDP$_{FH}$, that would run trials of length H' from the augmented state (s'_0, H') until $V(s'_0, H')$ converges:

- LR^{2}TDP is generally more efficient than LRTDP$_{FH}$ because its trial length is short. Typically, when solving MDP $M_{s'_0}(L)$ from the above sequence, LR^{2}TDP's trials run into a state solved as part of some $M_{s'_0}(L')$, $L' < L$, already after a few action executions. For LRTDP$_{FH}$, on the other hand, the early trials are of length L.
- More importantly, by solving a sequence of MDPs with an increasing horizon, LR^{2}TDP allows us to measure *how long* on average solving an augmented state (s, L) takes for various values of L.

In fact, LR^{2}TDP uses LRTDP$_{FH}$ in its inner loop (line 41).

GOURMAND starts solving for each decision epoch by assuming it will allocate the remaining time uniformly over the remaining decision epochs, including the current one (line 12). If it is the initial epoch, it runs LR^{2}TDP for the initial state (lines 14 - 16), timing how long LR^{2}TDP takes to solve $M_{s_0}(1), M_{s_0}(2), \ldots$ and thereby initializing the averages $Ts_1, Ts_2, \ldots$ (lines 41-43), as long as the computation time T_0 allocated to the first decision epoch has not run out. Crucially, conducting these measurements is possible due to LR^{2}TDP's stopping condition that makes LR^{2}TDP proceed to solving MDP $M_{s_0}(L+1)$ once MDP $M_{s_0}(L)$ has been solved.

By the time T_0 runs out, GOURMAND achieves two things. First, it solves s_0 for some lookahead $\hat{L}_0$ (line 16), and can select an action in s_0 according to the optimal policy for $M_{s_0}(\hat{L}_0)$ (line 31). Second, in the process of solving for lookahead $\hat{L}_0$ it gets estimates Ts_L of the time it takes to solve a state completely for lookaheads $L = 1, 2, \ldots, \hat{L}_0$ (lines 42 - 43).

In each epoch t past the initial one, GOURMAND figures out the finite lookahead value L_t for which it *should* be able to solve the current and all subsequent epochs if it allocated time $T_t = \frac{\overline{T}}{H-t}$ to each (line 19). It does this based on the estimates Ts_L it has obtained previously. Then, GOURMAND decides whether it realistically *may be able to* solve the current decision epoch for an even larger lookahead $\hat{L}_t$ without impacting performance guarantees for future decision epochs, i.e. while ensuring that it can solve them for lookahead L_t. To see the intuition for how GOURMAND can achieve this, observe that since $Ts_{L_t} < T_t$, if GOURMAND solved the current epoch just for lookahead L_t, there would probably be some extra time of approximately $(T_t - Ts_{L_t})$ left. By itself, this extra time chunk does not let GOURMAND solve for a lookahead bigger than L_t. However, if GOURMAND "borrows" similar extra time chunks from *future* decision epochs $t' > t$, solving for a larger lookahead *now* may well be possible. Since there are $(H - t - 1)$ decision epochs after t, the total amount of additional time GOURMAND can gain via such borrowing is $(T_t - Ts_{L_t})(H - t - 1)$. Accordingly, GOURMAND adds $(T_t - Ts_{L_t})(H - t - 1)$ to T_t (line 20) and determines whether it can increase the target lookahead to $L_t + 1$ thanks to the borrowed time. Establishing this may be complicated by the fact that GOURMAND does not necessarily know how long solving for $L_t + 1$ takes, in which case its estimate for Ts_{L_t+1} is ∞ (line 6) . However, both in the case when Ts_{L_t+1} is unknown and in the case when it is known to be less that $\hat{T}_t$, GOURMAND takes the risk and sets the target lookahead $\hat{L}_t$ to $L_t + 1$ (line 23).

GOURMAND then sets off solving MDP $M_s(\hat{L}_t)$ until it either manages to solve s for lookahead $\hat{L}_t$ or the allocated time $\hat{T}_t$ runs out (line 26). Throughout the process it has LR^{2}TDP measure how long solving s takes for lookaheads $L = 1, 2, \ldots, \hat{L}_t$ and update the running averages Ts_L accordingly (lines 42 - 43).

Although as described, GOURMAND can decide on lookaheads $\hat{L}_t$ for each decision epoch automatically, the use of LRTDP introduces an important practical limitation. LRTDP updates state value via Bellman backups. A Bellman backup assigns the value of the best action a^* in a state (s, h) to (s, h) itself. To evaluate an action a, it iterates over all successors of s under a. In MDPs with exogenous events, the number of such successors may be astronomical, possibly the entire state space. Vanilla LRTDP and LR^{2}TDP would not be able to handle such problems in practice. On the other hand, UCT can cope with them more easily, since it never explicitly iterates over successors of a state.

To address this issue, GLUTTON's implementation of LR^{2}TDP heavily subsamples the set of successors of each state-action pair and uses other engineering optimizations such as separating out the natural dynamics to make subsampling even more efficient (Kolobov et al. 2012). Our implementation of GOURMAND adopts these modifications to the basic LR^{2}TDP as well.

Experimental Results

Experimental Setting. The goal of our empirical evaluation was to compare the performance of online LRTDP as used in GOURMAND to that of online UCT as used in PROST and offline LRTDP as used in GLUTTON across a diverse collection of finite-horizon MDPs. To this end, we present the results of these planners on the set of all benchmarks from the most recent International Probabilistic Planning Competition, IPPC-2011 (Sanner 2011). Since at present PROST and GLUTTON are not publicly available, we had to run GOURMAND under the IPPC-2011 conditions and compare its performance to PROST's and GLUTTON's competition results (Sanner 2011).

Overall, the IPPC-2011 setting models solving finite-horizon MDPs under time constraints fairly well. At the competition, each participant had 24 hours to solve the 80 available benchmark MDPs. The problems came from 8 sets (domains), 10 problems per set. MDPs in each domain were numbered 1 through 10, with size/difficulty increasing roughly with a problem's ordinal. Participants could divide up this time among the problems in any way they wished. Both PROST and GLUTTON chose to solve problems from the lowest-numbered to the highest-numbered ones. Initially, both PROST[1] and GLUTTON divided the time equally among all problems, but gradually redistributed time in similar ways to give more of it to larger problem instances, since smaller ones could be solved quickly. GOURMAND adopted this approach as well. As a result, planners ended up spending as little as several dozens of seconds on each of the small MDPs, and as long as 40 minutes on the largest ones. Details aside, all three planners had to solve finite-horizon MDPs under a time constraint, which was our intended comparison scenario.

Like competitors at IPPC-2011, GOURMAND ran on a separate Large Instance of Amazon EC2 node with 7.5 GB RAM. The policy of each participant on each problem was evaluated by having the participant execute its policy 30 times starting from the initial state to the horizon. The competition server, which simulated the actions sent by the planner, computed average reward of the policy of the 30 attempts. Afterwards, these rewards were converted to relative scores on a scale from 0 to 1 for each problem. The score of 0 corresponded to the average reward of a random policy or a policy that executed only the noop action, whichever was highest. The score of 1 corresponded to the highest reward any planner's policy earned on this problem. The winner was the planner with the highest relative score averaged across all 8 domains.

The IPPC-2011 winner, PROST, used UCT in the manner we already mentioned. Its authors specified a single tuned lookahead value, $L = 15$, for all of the benchmark problems. PROST also had a per-epoch timeout. To compute a

[1] From personal communication with the authors of PROST.

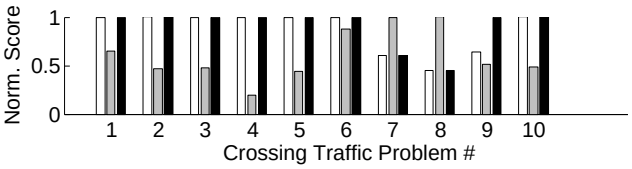

Figure 1: GOURMAND (avg. score 0.9052) vastly outperforms PROST (0.6099) on the Crossing Traffic domain.

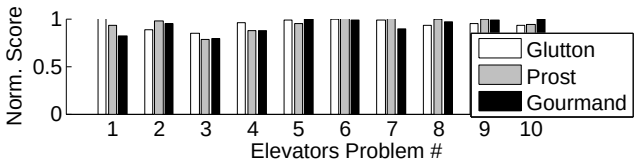

Figure 2: All planners are tied on the Elevators domain.

policy, PROST would run UCT with $L = 15$ for the time specified by the timeout, return the best action (according to the value function upon termination) to the server, which simulated the action and sent PROST a new state. If UCT happened to re-visit a state across the 30 policy execution attempts, it returned an action for it immediately, without waiting for the timeout. In this case, the freed-up time was redistributed among subsequent epochs. Because of this, UCT could also execute all 30 rounds before the time allocated to this problem was up. When this happened, the remaining time was distributed among the remaining problems.

Results and Analysis. The overall results for each domain are presented in Figures 1 - 8. Across all domains, GOURMAND earned the average score of 0.9183 ± 0.0222, PROST— 0.8608 ± 0.0220, and GLUTTON— 0.7701 ± 0.0235, i.e. GOURMAND outperforms the other two by a statistically significant amount.

Several performance patterns deserve a special note. First, we revisit the intuition we stated at the beginning that if the value of L chosen for UCT is too large, by the timeout UCT will still be very far from convergence and pick an action largely at random, whereas online LRTDP will converge completely for a smaller lookahead and make a more informed decision. While this may be true in some situations, our results on the IPPC-2011 benchmarks do not confirm this. In particular, consider the Sysadmin, Game of Life, and Traffic domains. All of them require a very small lookahead, typically up to 8, to come up with a near-optimal policy. Moreover, they have extremely large branching factors (around 2^{50} for some states of the largest Sysadmin instances). Since UCT used $L = 15$, one might expect it to make hardly any progress due to the enormous number of extra states it has to explore. Nonetheless, it wins on these domains overall, despite the fact that on many instances GOURMAND routinely solves states for $L = 6$. We hypothesize that on these problems, UCT may be arriving at a good policy much sooner than its value function converges. The fact that UCT does not need to perform Bellman backups, which are expensive in MDPs with large branching factors as in these domains, probably contributed to UCT's convergence speed. Nonetheless, more experimentation is needed for a more conclusive explanation.

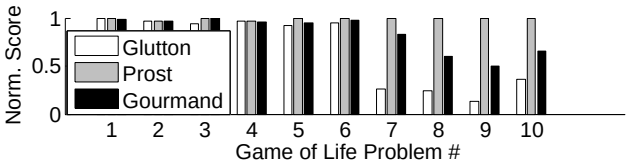

Figure 3: PROST (avg. score 0.9934) mildly outperforms GOURMAND (0.8438) on the Game of Life domain.

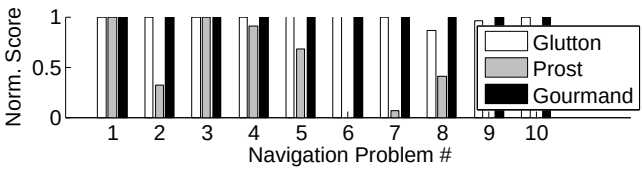

Figure 4: GOURMAND (avg. score 1.0) vastly outperforms PROST (0.4371) on the Navigation domain.

Second, we emphasize that GOURMAND's performance is more uniform than PROST's. The lookahead parameter for PROST was empirically picked to give good results across many of the competition domains (competition rules allowed this). Indeed, PROST performed very well on average and even outperformed GOURMAND on three domains above. Yet, due to its adaptive strategy, online LRTDP implemented by GOURMAND does not suffer sharp drops in performance on some problems sets as UCT implemented by PROST does, and is robust across all benchmark domains.

In fact, UCT's overall defeat was caused by very poor performance on two domains, Navigation (Figure 4) and Crossing Traffic (Figure 1). Incidentally, both of them are in effect goal-oriented domains — the agent incurs a cost for every decision epoch it is not in one of the special states staying in which is "free". Crucially, to reach these states successfully, one needs to select actions very carefully during the first epochs of the process. For instance, in Crossing Traffic, the agent is trying to cross a motorway. It can do it safely by making detours, or by boldly dashing across the moving stream of cars, which can kill the agent. Getting to the other side via detours takes longer, and the agent has to plan with a sufficient lookahead during the first few decision epochs of the process to realize this. This highlights the main drawback of guessing a value for L — even within the same domain, $L = 15$ is sufficient for some problems but not others, leading to catastrophic consequences for the agent in the latter case. Online LRTDP, if it has enough time, eventually arrives at a sufficiently large lookahead and solves many such problems successfully.

In comparison with GLUTTON, GOURMAND demonstrates an even more pronounced advantage. Since GLUTTON attempts to solve the problem offline, by the timeout it often fails to visit many states that its policy can visit from the initial state. In other words, its policy is not closed with respect to s_0. To compensate for this, during policy execution it uses various fallback cases in states it has never seen before. These default policies are usually not very good. GOURMAND does not have this problem, since it always makes an informed choice of action in states that it visits.

Last but not least, we point out that the presence of a termination condition in LRTDP can give rise to many adaptive time allocation strategies, of which GOURMAND exploits only one. Our objective in designing and evaluating

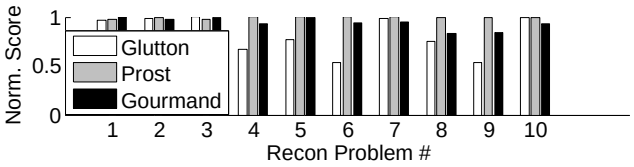

Figure 5: GOURMAND and PROST are tied on the Recon domain.

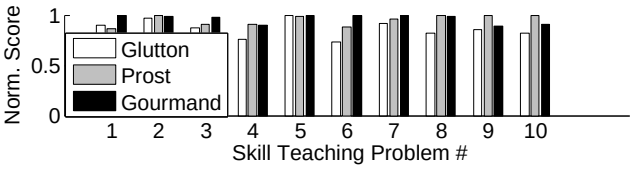

Figure 6: GOURMAND and PROST are tied on the Skill Teaching domain.

GOURMAND was not to pick the best such strategy. Rather, it was to demonstrate that at least some of them can turn a relatively inefficient *offline* planning algorithm with a termination condition into an *online* planner that is significantly more robust, performant, and easier to deploy than UCT. GOURMAND's results on IPPC-2011 domains showcase this message.

Related Work

There has not been much literature on analyzing the performance of UCT in solving MDPs. However, there have been attempts to examine its properties in the context of adversarial planning (Ramanujan and Selman 2011).

Finite-horizon MDPs (Puterman 1994) is a well-studied MDP class. Besides VI and LRTDP, another well-known algorithm for solving them is AO* (Nilsson 1980). However, using any of these methods offline on large finite-horizon MDPs is infeasible because of their time and space requirements. A promising class of approaches for solving finite-horizon MDPs offline that would circumvent these limitations is automatic dimensionality reduction ((Buffet and Aberdeen 2006), (Kolobov, Mausam, and Weld 2009)). It has worked well for goal-oriented MDPs, but compactifying the value function of finite-horizon MDPs appears to follow different intuitions, and we are not aware of any such algorithms for this class of problems. The methods for solving finite-horizon MDPs online have been studied fairly little.

Conclusion

In the light of recent popularity of UCT for solving MDPs, this paper attempts to answer the question: is UCT fundamentally better than existing MDP solution algorithms such as LRTDP? In particular, does LRTDP have any advantages over UCT when both are used online? We identify one property that makes online LRTDP a more adaptable planning algorithm — its termination condition. To solve a finite-horizon MDP under a time constraint, UCT needs an expert to specify a lookahead L for which UCT should solve states it encounters. Mistakes in setting this parameter are easy to make and can be very costly. In contrast, LRTDP's termination condition allows us to devise an adaptive strategy that determines a good lookahead value automatically. We implement this strategy in an online planner GOURMAND and compare it to the IPPC winner PROST, which is based on online UCT, and GLUTTON, based on offline LRTDP. The experimental results show that, even with a carefully chosen lookahead, online UCT performs worse than online LRTDP, the latter's strategy of choosing the lookahead being more adaptive and robust.

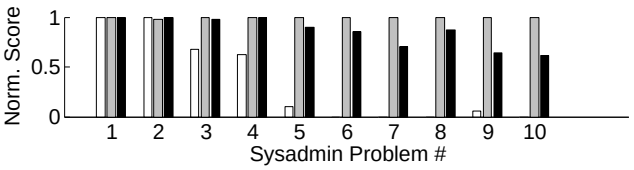

Figure 7: PROST (avg. score 0.9978) mildly outperforms GOURMAND (0.8561) on the Sysadmin domain.

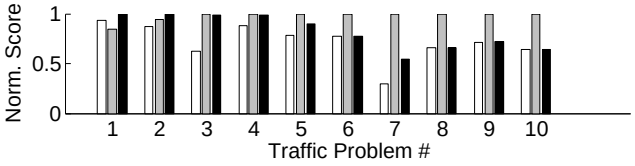

Figure 8: PROST (avg. score 0.9791) mildly outperforms GOURMAND (0.8216) on the Traffic domain.

Acknowledgments. We would like to thank Thomas Keller and Patrick Eyerich from the University of Freiburg for valuable information about PROST, and the anonymous reviewers for insightful comments. This work has been supported by NSF grant IIS-1016465, ONR grant N00014-12-1-0211, and the UW WRF/TJ Cable Professorship.

References

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bjarnason, R.; Fern, A.; and Tadepalli, P. 2009. Lower bounding klondike solitaire with Monte-Carlo planning. In *ICAPS'09*.

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

Buffet, O., and Aberdeen, D. 2006. The factored policy gradient planner (ipc-06 version). In *Fifth International Planning Competition at ICAPS'06*.

Gelly, S., and Silver, D. 2008. Achieving master level play in 9x9 computer Go. In *AAAI'08*, 1537–1540.

Keller, T., and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *ICAPS'12*.

Kocsis, L., and Szepesvári, C. 2006. 'bandit based monte-carlo planning. In *ECML'06*, 282–293.

Kolobov, A.; Dai, P.; Mausam; and Weld, D. S. 2012. Reverse iterative deepening for finite-horizon MDPs with large branching factors. In *ICAPS'12*.

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

Nilsson, N. 1980. *Principles of Artificial Intelligence*. Tioga Publishing.

Puterman, M. 1994. *Markov Decision Processes*. John Wiley & Sons.

Ramanujan, R., and Selman, B. 2011. Trade-offs in sampling-based adversarial planning. In *ICAPS'11*.

Sanner, S. 2011. ICAPS 2011 international probabilistic planning competition. http://users.cecs.anu.edu.au/∼ ssanner/IPPC_2011/.