Reverse Iterative Deepening for Finite-Horizon MDPs with Large Branching Factors

Andrey Kolobov[⋆] **Peng Dai**[† *] **Mausam**[⋆] **Daniel S. Weld**[⋆]
{akolobov, daipeng, mausam, weld}@cs.washington.edu
[⋆]Dept. of Computer Science and Engineering
University of Washington
Seattle, USA, WA-98195
[†]Google Inc.
1600 Amphitheater Pkwy
Mountain View, USA, CA-94043

Abstract

In contrast to previous competitions, where the problems were goal-based, the 2011 International Probabilistic Planning Competition (IPPC-2011) emphasized finite-horizon reward maximization problems with large branching factors. These MDPs modeled more realistic planning scenarios and presented challenges to the previous state-of-the-art planners (e.g., those from IPPC-2008), which were primarily based on domain determinization — a technique more suited to goal-oriented MDPs with small branching factors. Moreover, large branching factors render the existing implementations of RTDP- and LAO*-style algorithms inefficient as well.

In this paper we present GLUTTON, our planner at IPPC-2011 that performed well on these challenging MDPs. The main algorithm used by GLUTTON is LR^{2}TDP, an LRTDP-based optimal algorithm for finite-horizon problems centered around the novel idea of *reverse iterative deepening*. We detail LR^{2}TDP itself as well as a series of optimizations included in GLUTTON that help LR^{2}TDP achieve competitive performance on difficult problems with large branching factors — subsampling the transition function, separating out natural dynamics, caching transition function samples, and others. Experiments show that GLUTTON and PROST, the IPPC-2011 winner, have complementary strengths, with GLUTTON demonstrating superior performance on problems with few high-reward terminal states.

Introduction

New benchmark MDPs presented at the International Probabilistic Planning Competition (IPPC) 2011 (Sanner 2011) demonstrated several weaknesses of existing solution techniques. First, the dominating planners of past years (FF-Replan (Yoon, Fern, and Givan 2007), RFF (Teichteil-Koenigsbuch, Infantes, and Kuter 2008), etc.) had been geared towards goal-oriented MDPs with relatively small branching factors. To tackle such scenarios, they had relied on fully determinizing the domain (small branching factor made this feasible) and solving the determinized version of the given problem. For the latter part, the performance of these solvers critically relied on powerful classical planners (e.g., FF (Hoffmann and Nebel 2001)) and heuristics, all of which assumed the existence of a goal, the uniformity of action costs, benign branching factors, or all three. In contrast, the majority of IPPC-2011 MDPs were problems with a finite horizon, non-uniform action costs, large branching factors, and no goal states — characteristics to which determinization-based planners are hard to adapt. Incidentally, large branching factors made the existing implementations of heuristic search algorithms such as LRTDP (Bonet and Geffner 2003) or AO* (Nilsson 1980) obsolete as well. These algorithms are centered around the Bellman backup operator, which is very expensive to compute when state-action pairs have many successors.

Second, previous top-performers optimized for the probability of their policy reaching the MDP's goal, which was the evaluation criterion at preceding IPPCs (Bryce and Buffet 2008), not *the expected reward* of that policy. At IPPC-2011 the evaluation criterion changed for the latter, more subtle objective, and thus became more stringent.

Thus, overall, IPPC-2011 introduced much more realistic MDPs and evaluation criteria than before. Indeed, in real-world systems, large branching factors are common and are often caused by *natural dynamics*, effects of exogenous events or forces of nature that cannot be directly controlled but that need to be taken into account during planning. Moreover, the controller (e.g., on a robot) may only have limited time to come up with a policy, a circumstance IPPC-2011 also attempted to model, and the expected reward of the produced policy is very important. To succeed under these conditions, a planner needs to be not only scalable but also sensitive to the expected reward maximization criterion and, crucially, have a strong anytime performance.

The main theoretical contribution of this paper is LR^{2}TDP, an algorithm that, with additional optimizations, can stand up to these challenges. LR^{2}TDP is founded on a crucial observation that for many MDPs $M(H)$ with horizon H, one can produce a successful policy by solving $M(h)$, the same MDP but with a much smaller horizon h. Therefore, under time constraints, trying to solve the sequence of MDPs $M(1), M(2), \cdots$ with increasing horizon will often yield a near-optimal policy even if the computation is interrupted long before the planner gets to tackle MDP $M(H)$. this strategy, which we call *reverse iterative deepening*, forms the basis of LR^{2}TDP.

Although the above intuition addresses the issue of anytime performance, by itself it does not enable LR^{2}TDP to

[*]Peng Dai completed this work while at the University of Washington.

handle large branching factors. Accordingly, in this paper we introduce GLUTTON, a planner derived from LR^{2}TDP and our entry in IPPC-2011. GLUTTON endows LR^{2}TDP with optimizations that help achieve competitive performance on difficult problems with large branching factors – subsampling the transition function, separating out natural dynamics, caching transition function samples, and using primitive cyclic policies as a fall-back solution.

Thus, this paper makes the following contributions:

- We introduce the LR^{2}TDP algorithm, an extension of LRTDP to finite-horizon problems based on the idea of reverse iterative deepening.
- We describe the design of GLUTTON, our IPPC-2011 entry built around LR^{2}TDP. We discuss various engineering optimizations that were included in GLUTTON to improve LR^{2}TDP's performance on problems with large branching factors due to natural dynamics.
- We present results of empirical studies that demonstrate that LR^{2}TDP performs much better than the straightforward extension of LRTDP to finite-horizon MDPs. In addition, we carry out ablation experiments showing the effects of various optimizations on GLUTTON's performance. Finally, we analyze the comparative performance of GLUTTON and PROST (Keller and Eyerich 2012), the winner of IPPC-2011, and find that the two have complementary strengths.

Background

MDPs. In this paper, we focus on probabilistic planning problems modeled by finite-horizon MDPs with a start state, defined as tuples of the form $M(H) = \langle\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, s_0\rangle, H\rangle$ 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 $\mathcal{S} \times \mathcal{A} \to \mathcal{R}$ that specifies action rewards, s_0 is the start state, and H is the horizon, the number of time steps after which the process stops.

In this paper, we will reason about the *augmented state space* of $M(H)$, which is a set $\mathcal{S} \times \{0, \ldots, H\}$ of state-number of steps-to-go 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 time steps.

In particular, 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 steps, $h \leq H$. Ideally, we would like to find an optimal policy π^*, one whose value function V^* for all $s \in \mathcal{S}$ 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 $s \in \mathcal{S}$:

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

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

WLOG, we assume the optimal action selection rule π^* to be deterministic Markovian, 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 such policy is guaranteed to exist (Puterman 1994). If V^* is known, a deterministic π^* can be derived from it by choosing a V^*-greedy action in each state all $1 \leq h \leq H$.

Solution Methods. Equation 1 suggests 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 $t = 1, 2, \ldots H$.

The version of VI for infinite-horizon goal-oriented stochastic shortest path MDPs (Bertsekas 1995) has given rise to many improvements. AO* (Nilsson 1980) is an algorithm that works specifically with loop-free MDPs (of which finite-horizon MDPs are a special case). Trial-based methods, e.g., RTDP (Barto, Bradtke, and Singh 1995) and LRTDP (Bonet and Geffner 2003), try to reach the goal from the initial state multiple times (in multiple trials) and update the value function over the states in the trial path using Bellman backups. Unlike VI, these algorithms memorize only states reachable from s_0, thereby typically requiring much less space. As we show in this paper, LRTDP can adapted and optimized for finite-horizon MDPs.

LR^{2}TDP

We begin by introducing LR^{2}TDP, an extension of LRTDP for finite-horizon problems. Like its predecessor, LR^{2}TDP solves an MDP for the given initial state s_0 optimally in the limit. Extending LRTDP to finite-horizon problems may seem an easy task, but its most natural extension performs worse than the one we propose, LR^{2}TDP.

As a reminder, LRTDP (Bonet and Geffner 2003) for goal-oriented MDPs operates in a series of trials starting at the initial state s_0. Each trial consists of choosing the greedy best action in the current state according to the current value function, performing a Bellman backup on the current state, sampling an outcome of the chosen action, transitioning to the corresponding new state, and repeating the cycle. A trial continues until it reaches a goal, a dead end (a state from which reaching the goal is impossible), or a converged state. At the end of each trial, LRTDP performs a special convergence check on all states in the trial to prove, whenever possible, the convergence of these states' values. Once it can prove that s_0 has converged, LRTDP halts.

Thus, a straightforward adaptation of LRTDP to a finite-horizon MDP $M(H)$, which we call LRTDP-FH, is to let each trial start at (s_0, H) and run for at most H time steps. Indeed, if we convert a finite-horizon MDP to its goal-oriented counterpart, all states H steps away from s_0 are goal states. However, as we explain below, LRTDP-FH's anytime performance is not very good, so we turn to a more sophisticated approach.

Our novel algorithm, LR^{2}TDP, follows a different strategy, which we name *reverse iterative deepening*. As its pseudocode in Algorithm 1 shows, it uses LRTDP-FH in a loop to solve a sequence of MDPs $M(1), M(2), \cdots, M(H)$, in that order. In particular, LR^{2}TDP first decides how to act optimally in $(s_0, 1)$, i.e.

Input: MDP $M(H)$ with initial state s_0
Output: Policy for $M(H)$ starting at s_0

function *LR^{2}TDP*(MDP $M(H)$, initial state s_0)
begin
  foreach $h = 1, \ldots, H$ *or until time runs out* **do**
    Run LRTDP-FH($M(h)$, s_0)
  end
end

function *LRTDP-FH*(MDP $M(h)$, initial state s_0)
begin
  Convert $M(h)$ into the equivalent goal-oriented MDP M_g^h, whose goals are states of the form $(s, 0)$.
  Run LRTDP(M_g^h, s_0), memoizing the values of all the augmented states encountered in the process
end

Algorithm 1: LR^{2}TDP

assuming there is only one more action to execute — this is exactly equivalent to solving $M(1)$. Then, LR^{2}TDP runs LRTDP-FH to decide how to act optimally starting at $(s_0, 2)$, i.e. two steps away from the horizon — this amounts to solving $M(2)$. Then it runs LRTDP-FH again to decide how to act optimally starting in $(s_0, 3)$, thereby solving $M(3)$, and so on. Proceeding this way, LR^{2}TDP either eventually solves $M(H)$ or, if operating under a time limit, runs out of time and halts after solving $M(h')$ for some $h' < H$.

Crucially, in the spirit of dynamic programming, LR^{2}TDP reuses state values computed while solving $M(1), M(2), \ldots, M(h-1)$ when tackling the next MDP in the sequence, $M(h)$. Namely, observe that any (s, h') in the augmented state space of any MDP $M(h'')$ also belongs to the augmented states spaces of all MDPs $M(h''')$, $h''' \geq h''$, and $V^*(s, h')$ is the same for all these MDPs. Therefore, by the time LR^{2}TDP gets to solving $M(h)$, values of many of its states will have been updated or even converged as a result of handling some $M(i)$, $i < h$. Accordingly, LR^{2}TDP memoizes values and convergence labels of all augmented states ever visited by LRTDP-FH while solving for smaller horizon values, and reuses them to solve subsequent MDPs in the above sequence. Thus, solving $M(h)$ takes LR^{2}TDP only an incremental effort over the solution of $M(h-1)$.

LR^{2}TDP can be viewed as backchaining from the goal in a goal-oriented MDP with no loops. Indeed, a finite-horizon MDP $M(H)$ is simply a goal-oriented MDP whose state space is the augmented state space of $M(H)$, and whose goals are all states of the form (s, H). It has no loops because executing any action leads from some state (s, h) to another state $(s', h-1)$. LR^{2}TDP essentially solves such MDPs by first assuming that the goal is one step away from the initial state, then two steps from the initial state, and so on, until it addresses the case when the goal is H steps away from the initial state. Compare this with LRTDP-FH's behavior when solving $M(H)$. LRTDP-FH does not backtrack from the goal; instead, it tries to forward-chain from the initial state to the goal (via trials) and propagates state values backwards whenever it succeeds. As an alternative perspective, LRTDP-FH iterates on the search depth, while LR^{2}TDP iterates on the distance from the horizon. The benefit of the latter is that it allows for the reuse of computation across different iterations.

Clearly, both LRTDP-FH and LR^{2}TDP eventually arrive at the optimal solution. So, what are the advantages of LR^{2}TDP over LRTDP-FH? We argue that if stopped prematurely, the policy of LR^{2}TDP is likely to be much better for the following reasons:

- In many MDPs $M(H)$, the optimal policy for $M(h)$ for some $h << H$ is optimal or near-optimal for $M(H)$ itself. E.g., consider a manipulator that needs to transfer blocks regularly arriving on one conveyor belt onto another belt. The manipulator can do one pick-up, move, or put-down action per time step. It gets a unit reward for moving each block, and needs to accumulate as much reward as possible over 50 time steps. Delivering one block from one belt to another takes at most 4 time steps: move manipulator to the source belt, pick up a block, move manipulator to the destination belt, release the block. Repeating this sequence of actions over 50 time steps clearly achieves maximum reward for $M(50)$. In other words, $M(4)$'s policy is optimal for $M(50)$ as well.

  Therefore, explicitly solving $M(50)$ for all 50 time steps is a waste of resources — solving $M(4)$ is enough. However, LRTDP-FH will try to do the former — it will spend a lot of effort trying to solve M for horizon 50 at once. Since it "spreads" its effort over many time steps, it will likely fail to completely solve $M(h)$ for any $h < H$ by the deadline. Contrariwise, LR^{2}TDP solves the given problem incrementally, and may have a solution for $M(4)$ (and hence for $M(50)$) if stopped prematurely.
- When LRTDP-FH starts running, many of its trials are very long, since each trial halts only when it reaches a converged state, and at the beginning reaching a converged state takes about H time steps. Moreover, at the beginning, each trial causes the convergence of only a few states (those near the horizon), while the values of augmented states with small time step values change very little. Thus, the time spent on executing the trials is largely wasted. In contrast, LR^{2}TDP's trials when solving an MDP $M(h)$ are very short, because they quickly run into states that converged while solving $M(h-1)$ and before, and often lead to convergence of most of trial's states. Hence, we can expect LR^{2}TDP to be faster.
- As a consequence of large trial length, LRTDP-FH explores (and therefore memorizes) many augmented states whose values (and policies) will not have converged by the time the planning process is interrupted. Thus, it risks using up available memory before it runs out of time, and to little effect, since it will not know well how to behave in most of the stored states anyway. In contrast, LR^{2}TDP typically knows how to act optimally in a large fraction of augmented states in its memory.

Note that, incidentally, LR^{2}TDP works in much the same way as VI, raising a question: why not use VI in the first place? The advantage of asynchronous dynamic programming over VI is similar in finite-horizon settings and in goal-oriented settings. A large fraction of the state space may be unreachable from s_0 in general and by the optimal policy in particular. LR^{2}TDP avoids storing information about many

of these states, especially if guided by an informative heuristic. In addition, in finite-horizon MDPs, many states are not reachable from s_0 *within* H *steps*, further increasing potential savings from using LR^{2}TDP.

So far, we have glossed over a subtle question: if LR^{2}TDP is terminated after solving $M(h)$, $h < H$, what policy should it use in augmented states (s, h') that it has never encountered? There are two cases to consider — a) LR^{2}TDP may have solved s for some $h'' < \min\{h, h'\}$, and b) LR^{2}TDP has not solved (or even visited) s for any time step. In the first case, LR^{2}TDP can simply find the largest value $h'' < \min\{h, h'\}$ for which (s, h'') is solved and return the optimal action for (s, h''). This is the approach we use in GLUTTON, our implementation of LR^{2}TDP, and it works well in practice. Case b) is more complicated and may arise, for instance, when s is not reachable from s_0 within h steps. One possible solution is to fall back on some simple *default policy* in such situations. We discuss this option when describing the implementation of GLUTTON.

Max-Reward Heuristic

To converge to an optimal solution, LR^{2}TDP needs to be initialized with an admissible heuristic, i.e., an upper bound on V^*. For this purpose, GLUTTON uses an estimate we call the *Max-Reward* heuristic. Its computation hinges on knowing the maximum reward R_{max} any action can yield in any state, or an upper bound on it. R_{max} can be automatically derived for an MDP at hand with a simple domain analysis.

To produce a heuristic value $V_0(s, h)$ for (s, h), Max-Reward finds the largest horizon value $h' < h$ for which GLUTTON already has an estimate $V(s, h')$. Recall that GLUTTON is likely to have $V(s, h')$ for some such h', since it solves the given MDP in the reverse iterative deepening fashion with LR^{2}TDP. If so, Max-Reward sets $V_0(s, h) = V(s, h') + R_{max}(h - h')$; otherwise, it sets $V(s, h) = R_{max}h$. The bound obtained in this way is often very loose but is guaranteed to be admissible.

The Challenge of Large Branching Factors

In spite of its good anytime behavior, LR^{2}TDP by itself would not perform well on many IPPC-2011 benchmarks due to large branching factors in these MDPs. In real-world systems, large branching factors often arise due to the presence of *natural dynamics*. Roughly, the natural dynamics of an MDP describes what happens to various objects in the system if the controller does not act on them explicitly in a given time step. In physical systems, it can model laws of nature, e.g. the effects of radioactive decay on a collection of particles. It can also capture effects of exogenous events.

For instance, in the MDPs of Sysadmin (Sanner 2011), one of IPPC-2011 domains, the task is to control a network of servers. At any time step, each server is either up or down. The controller can restart one server per time step, and that server is guaranteed to be up at the next time step. The other servers can change their state spontaneously — those that are down can go back up with some small probability, and those that are up can go down with a probability proportional to the fraction of their neighbors that are down. These random transitions are the natural dynamics of the system, and they cause the MDP to have a large branching factor. Imagine a Sysadmin problem with 50 servers. Due to the natural dynamics, the system can transition to any of the 2^{50} states from any given one in just one time step.

The primary effect of a large branching factor on the effectiveness of algorithms such as VI, RTDP, or AO* is that computing Bellman backups (Equation 1) explicitly becomes prohibitively expensive, since the summation in it has to be carried out over a large fraction of the state space. We address this issue in the next section.

GLUTTON

In this section, we present GLUTTON, our LR^{2}TDP-based entry at the IPPC-2011 competition that endows LR^{2}TDP with mechanisms for efficiently handling natural dynamics and other optimizations. Below we describe each of these optimizations in detail. A C++ implementation of GLUTTON is available at http://www.cs.washington.edu/ai/planning/glutton.html.

Subsampling the Transition Function. GLUTTON's way of dealing with a high-entropy transition function is to subsample it. For each encountered state-action pair (s, a), GLUTTON samples a set $U_{s,a}$ of successors of s under a, and performs Bellman backups using states in $U_{s,a}$:

$$V^*(s,h) \approx \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \sum_{s' \in U_{s,a}} \mathcal{T}(s,a,s') V^*(s', h-1) \right\} \quad (2)$$

The size of $U_{s,a}$ is chosen to be much smaller than the number of states to which a could transition from s. There are several heuristic ways of setting this value, e.g. based on the entropy of the transition function. At IPPC-2011 we chose $|U_{s,a}|$ for a given problem to be a constant.

Subsampling can give an enormous improvement in efficiency for GLUTTON at a reasonably small reduction in the solution quality compared to full Bellman backups. However, subsampling alone does not make solving many of the IPPC benchmarks feasible for GLUTTON. Consider, for instance, the aforementioned Sysadmin example with 50 servers (and hence 50 state variables). There is a total of 51 ground actions in the problem, one for restarting each server plus a *noop* action. Each action can potentially change all 50 variables, and the value of each variable is sampled independently from the values of others. Suppose we set $|U_{s,a}| = 30$. Even for such a small size of $U_{s,a}$, determining the current greedy action in just one state could require $51 \cdot (50 \cdot 30) = 76,500$ variable sampling operations. Considering that the procedure of computing the greedy action in a state may need to be repeated billions of times, the need for further improvements, such as those that we describe next, quickly becomes evident.

Separating Out Natural Dynamics. One of our key observations is the fact that the efficiency of sampling successor states for a given state can be drastically increased by reusing some of the variable samples when generating successors for multiple actions. To do this, we separate each action's effect into those due to natural dynamics (*exogenous* effects), those due to the action itself (*pure* effects), and those due to some interaction between the two (*mixed*

effects). More formally, assume that an MDP with natural dynamics has a special action *noop* that captures the effects of natural dynamics when the controller does nothing. In the presence of natural dynamics, for each non-*noop* action a, the set $\mathcal{X}$ of problem's state variables can be represented as a disjoint union

$$\mathcal{X} = \mathcal{X}_a^{ex} \cup \mathcal{X}_a^{pure} \cup \mathcal{X}_a^{mixed} \cup \mathcal{X}_a^{none}$$

Moreover, for the *noop* action we have

$$\mathcal{X} = (\cup_{a \neq noop}(\mathcal{X}_a^{ex} \cup \mathcal{X}_a^{mixed})) \cup \mathcal{X}_{noop}^{none}$$

where X_a^{ex} are variables acted upon only by the exogenous effects, X_a^{pure} — only by the pure effects, X_a^{mixed} — by both the exogenous and pure effects, and X_a^{none} are not affected by the action at all. For example, in a Sysadmin problem with n machines, for each action a other than the *noop*, $|X_a^{pure}| = 0$, $|\mathcal{X}_a^{ex}| = n - 1$, and $|X_a^{none}| = 0$, since natural dynamics acts on any machine unless the administrator restarts it. $|X_a^{mixed}| = 1$, consisting of the variable for the machine the administrator restarts. Notice that, at least in the Sysadmin domain, for each non-*noop* action a, $|X_a^{ex}|$ is much larger than $|X_a^{pure}| + |X_a^{mixed}|$. Intuitively, this is true in many real-world domains as well — natural dynamics affects many more variables than any single non-*noop* action. These observations suggest generating $|U_{s,noop}|$ successor states for the *noop* action, and then modifying these samples in order to obtain successors for other actions by resampling some of the state variables using each action's pure and mixed effects.

We illustrate this technique on the example of approximately determining the greedy action in some state s of the Sysadmin-50 problem. Namely, suppose that for each action a in s we want to sample a set of successor states $U_{s,a}$ to evaluate Equation 2. First, we generate $|U_{s,noop}|$ *noop* sample states using the natural dynamics (i.e., the *noop* action). Setting $|U_{s,noop}| = 30$ for the sake of the example, this takes $50 \cdot 30 = 1500$ variable sampling operations, as explained previously. Now, for each resulting $s' \in U_{s,noop}$ and each $a \neq noop$, we need to re-sample variables $\mathcal{X}_a^{pure} \cup \mathcal{X}_a^{mixed}$ and substitute their values into s'. Since $|\mathcal{X}_a^{pure} \cup \mathcal{X}_a^{mixed}| = 1$, this takes one variable sampling operation per action per $s' \in U_{s,noop}$. Therefore, the total number of additional variable sampling operations to compute sets $U_{s,a}$ for all $a \neq noop$ is 30 *noop* state samples$\cdot$ 1 variable sample per non-*noop* action per *noop* state sample$\cdot$ 50 non-*noop* actions $= 1500$. This gives us 30 state samples for each non-*noop* action. Thus, to evaluate Equation 2 in a given state with 30 state samples per action, we have to perform $1500 + 1500 = 3000$ variable sampling operations. This is about 25 times fewer than the 76,500 operations we would have to perform if we subsampled naively. Clearly, in general the speedup will depend on how "localized" actions' pure and mixed effects in the given MDP are compared to the effects of natural dynamics.

The caveat of sharing the natural dynamics samples for generating non-*noop* action samples is that the resulting non-*noop* action samples are not independent, i.e. are biased. However, in our experience, the speedup from this strategy (as illustrated by the above example) and associated gains in policy quality when planning under time constraints outweigh the disadvantages due to the bias in the samples.

We note that several techniques similar to subsampling and separating natural dynamics have been proposed in the reinforcement learning (Proper and Tadepalli 2006) and concurrent MDP (Mausam and Weld 2004) literature. An alternative way of increasing the efficiency of Bellman backups is performing them on a symbolic value function representation, e.g., as in symbolic RTDP (Feng, Hansen, and Zilberstein 2003). A great improvement over Bellman backups with explicitly enumerated successors, it nonetheless does not scale to many IPPC-2011 problems.

Caching the Transition Function Samples. In spite of the already significant speedup due to separating out the natural dynamics, we can compute an approximation to the transition function even more efficiently. Notice that nearly all the memory used by algorithms such as LR^{2}TDP is occupied by the state-value table containing the values for the already visited (s, h) pairs. Since LR^{2}TDP populates this table lazily (as opposed to VI), when LR^{2}TDP starts running the table is almost empty and most of the available memory on the machine is unused. Instead, GLUTTON uses this memory as a cache for samples from the transition function. That is, when GLUTTON analyzes a state-action pair (s, a) for the first time, it samples successors of s under a as described above and stores them in this cache (we assume the MDP to be stationary, so the samples do not need to be cached separately for each $((s, h), a)$ pair). When GLUTTON encounters (s, a) again, it retrieves the samples for it from the cache, as opposed to re-generating them. Initially the GLUTTON process is CPU-bound, but due to caching it quickly becomes memory-bound as well. Thus, the cache helps it make the most of available resources. When all of the memory is filled up, GLUTTON starts gradually shrinking the cache to make room for the growing state-value table. Currently, it chooses state-action pairs for eviction and replacement randomly.

Default Policies. Since GLUTTON subsamples the transition function, it may terminate with an incomplete policy — it may not know a good action in states it missed due to subsampling. To pick an action in such a state (s, h'), GLUTTON first attempts to use the trick discussed previously, i.e. to return either the optimal action for some solved state (s, h''), $h'' < h'$, or a random one. However, if the branching factor is large or the amount of available planning time is small, GLUTTON may need to do such random "substitutions" for so many states that the resulting policy is very bad, possibly worse than the uniformly random one.

As it turns out, for many MDPs there are simple *cyclic policies* that do much better than the completely random policy. A cyclic policy consists in repeating the same sequence of steps over and over again. Consider, for instance, the robotic manipulator scenario from before. The optimal policy for it repeats an action cycle of length 4. In general, near-optimal cyclic policies are difficult to discover. However, it is easy to evaluate the set of *primitive cyclic policies* for a problem, each of which repeats a *single* action.

This is exactly what GLUTTON does. For each action, it evaluates the cyclic policy that repeats that action in any state by simulating this policy several times and averaging

the reward. Then, it selects the best such policy and compares it to three others, also evaluated by simulation: (1) the "smart" policy computed by running LR^{2}TDP with substituting random actions in previously unencountered states, (2) the "smart" policy with substituting the action from the best primitive cyclic policy in these states, and (3) the completely random policy. For the actual execution, GLUTTON uses the best of these four. As we show in the Experiments section, on several domains, pure primitive cyclic policies turned out to be surprisingly effective.

Performance Analysis

Our goals in this section are threefold — a) to show the advantage of LR^{2}TDP over LRTDP-FH, b) to show the effects of the individual optimizations on GLUTTON's performance, and c) to compare the performance of GLUTTON at IPPC-2011 to that of its main competitor, PROST.

We report results using the setting of IPPC-2011 (Sanner 2011). At IPPC-2011, the competitors needed to solve 80 problems. The problems came from 8 domains, 10 problems each. Within each domain, problems were numbered 1 through 10, with problem size/difficulty roughly increasing with its number. All problems were reward-maximization finite-horizon MDPs with the horizon of 40. They were described in the new RDDL language (Sanner 2010), but translations to the older format, PPDDL, were available and participants could use them instead. The participants had a total of 24 hours of wall clock time to allocate in any way they wished among all the problems. Each participant ran on a separate large instance of Amazon's EC2 node (4 virtual cores on 2 physical cores, 7.5 GB RAM).

The 8 benchmark domains at IPPC-2011 were Sysadmin (abbreviated as Sysadm in figures in this section), Game of Life (GoL), Traffic, Skill Teaching (Sk T), Recon, Crossing Traffic (Cr Tr), Elevators (Elev), and Navigation (Nav). Sysadmin, Game of Life, and Traffic domains are very large (many with over 2^{50} states). Recon, Skill Teaching, and Elevators are smaller but require a larger planning lookahead to behave near-optimally. Navigation and Crossing Traffic essentially consist of goal-oriented MDPs. The goal states are not explicitly marked as such; instead, they are the only states visiting which yields a reward of 0, whereas the highest reward achievable in all other states is negative.

A planner's solution policy for a problem was assessed by executing the policy 30 times on a special server. Each of the 30 rounds would consist of the server sending the problem's initial state, the planner sending back an action for that state, the server executing the action, noting down the reward, and sending a successor state, and so on. After 40 such exchanges, another round would start. A planner's performance was judged by its average reward over 30 rounds.

In most of the experiments, we show planners' normalized scores on various problems. The normalized score of planner Pl on problem p always lies in the $[0, 1]$ interval and is computed as follows:

$$score_{norm}(Pl, p) = \frac{\max\{0, s_{raw}(Pl, p) - s_{baseline}(p)\}}{\max_i\{s_{raw}(Pl_i, p)\} - s_{baseline}(p)}$$

where $s_{raw}(Pl, p)$ is the average reward of the planner's policy for p over 30 rounds, $max_i\{s_{raw}(Pl_i, p)\}$ is the maximum average reward of any IPPC-2011 participant on p, and $s_{baseline}(p) = \max\{s_{raw}(random, p), s_{raw}(noop, p)\}$ is the baseline score, the maximum of expected rewards yielded by the $noop$ and random policies. Roughly, a planner's score is its policy's reward as a fraction of the highest reward of any participant's policy on the given problem.

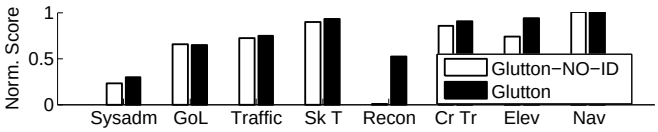

Figure 1: Average normalized scores of GLUTTON and GLUTTON-NO-ID on all of the IPPC-2011 domains.

We start by presenting the experiments that illustrate the benefits of various optimizations described in this paper. In these experiments, we gave different variants of GLUTTON at most 18 minutes to solve each of the 80 problems (i.e., divided the available 24 hours equally among all instances).

Reverse Iterative Deepening. To demonstrate the power of iterative deepening, we built a version of GLUTTON denoted GLUTTON-NO-ID that uses LRTDP-FH instead of LR^{2}TDP. A-priori, we may expect two advantages of GLUTTON over GLUTTON-NO-ID. First, according to the intuition in the section describing LR^{2}TDP, GLUTTON should have a better anytime performance. That is, if GLUTTON and GLUTTON-NO-ID are interrupted T seconds after starting to solve a problem, GLUTTON's solution should be better. Second, GLUTTON should be faster because GLUTTON's trials are on average shorter than GLUTTON-NO-ID. The length of the latter's trials is initially equal to the horizon, while most of the former's end after only a few steps. Under limited-time conditions such as those of IPPC-2011, both of these advantages should translate to better solution quality for GLUTTON. To verify this prediction, we ran GLUTTON-NO-ID under IPPC-2011 conditions (i.e. on a large instance of Amazon EC2 with a 24-hour limit) and calculated its normalized scores on all the problems as if it participated in the competition.

Figure 1 compares GLUTTON and GLUTTON-NO-ID's results. On most domains, GLUTTON-NO-ID performs worse than GLUTTON, and on Sysadmin, Elevators, and Recon the difference is very large. This is a direct consequence of the above theoretical predictions. Both GLUTTON-NO-ID and GLUTTON are able to solve small instances on most domains within allocated time. However, on larger instances, both GLUTTON-NO-ID and GLUTTON typically use up all of the allocated time for solving the problem, and both are interrupted while solving. Since GLUTTON-NO-ID has worse anytime performance, its solutions on large problems tend to be worse than GLUTTON's. In fact, the Recon and Traffic domains are so complicated that GLUTTON-NO-ID and GLUTTON are almost always stopped before finishing to solve them. As we show when analyzing cyclic policies, on Traffic both planners end up falling back on such policies, so their scores are the same. However, on Recon cyclic policies do not work very well, causing GLUTTON-NO-ID to fail dramatically due to its poor anytime performance.

Separating out Natural Dynamics. To test the importance of separating out natural dynamics, we create a version of our planner, GLUTTON-NO-SEP-ND, lacking this

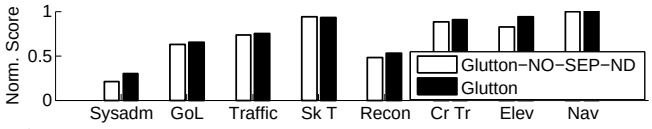

Figure 2: Average normalized scores of GLUTTON and GLUTTON-NO-SEP-ND on all of the IPPC-2011 domains.

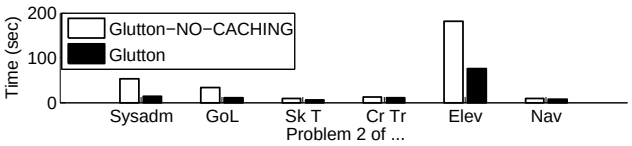

Figure 3: Time it took GLUTTON with and without caching to solve problem 2 of six IPPC-2011 domains.

feature. Namely, when computing the greedy best action for a given state, GLUTTON-NO-SEP-ND samples the transition function of each action independently. For any given problem, the number of generated successor state samples N per state-action pair was the same for GLUTTON and GLUTTON-NO-SEP-ND, but varied slightly from problem to problem. To gauge the performance of GLUTTON-NO-SEP-ND, we ran it on all 80 problems under the IPPC-2011 conditions. We expected GLUTTON-NO-SEP-ND to perform worse overall — without factoring out natural dynamics, sampling successors should become more expensive, so GLUTTON-NO-SEP-ND's progress towards the optimal solution should be slower.

Figure 2 compares the performance of GLUTTON and GLUTTON-NO-SEP-ND. As predicted, GLUTTON-NO-SEP-ND's scores are noticeably lower than GLUTTON's. However, we discovered the performance pattern to be richer than that. As it turns out, GLUTTON-NO-SEP-ND solves small problems from small domains (such as Elevators, Skill Teaching, etc.) almost as fast as GLUTTON. This effect is due to the presence of caching. Indeed, sampling the successor function is expensive during the first visit to a state-action pair, but the samples get cached, so on subsequent visits to this pair neither planner incurs any sampling cost. Crucially, on small problems, both GLUTTON and GLUTTON-NO-SEP-ND have enough memory to store the samples for *all* state-action pairs they visit in the cache. Thus, GLUTTON-NO-SEP-ND incurs a higher cost only at the initial visit to a state-action pair, which results in an insignificant speed increase overall.

In fact, although this is not shown explicitly in Figure 2, GLUTTON-NO-SEP-ND occasionally performs better than GLUTTON on small problems. This happens because for a given state, GLUTTON-NO-SEP-ND-produced samples for all actions are independent. This is not the case with GLUTTON since these samples are derived from the same set of samples from the *noop* action. Consequently, GLUTTON's samples have more bias, which makes the set of samples somewhat unrepresentative of the actual transition function.

The situation is quite different on larger domains such as Sysadmin. On them, both GLUTTON and GLUTTON-NO-SEP-ND at some point have to start shrinking the cache to make space for the state-value table, and hence may have to resample the transition function for a given state-action pair over and over again. For GLUTTON-NO-SEP-ND, this causes an appreciable performance hit, immediately visible in Figure 2 on the Sysadmin domain.

Caching Transition Function Samples. To demonstrate the benefits of caching, we pit GLUTTON against its clone without caching, GLUTTON-NO-CACHING. GLUTTON-NO-CACHING is so slow that it cannot handle most IPPC-2011 problems. Therefore, to show the effect of caching we run GLUTTON and GLUTTON-NO-CACHING on instance 2 of six IPPC-2011 domains (all domains but Traffic and Recon, whose problem 1 is already very hard), and record the amount of time it takes them to solve these instances. Instance 2 was chosen because it is harder than instance 1 and yet is easy enough that GLUTTON can solve it fairly quickly on all six domains both with and without caching.

As Figure 3 shows, even on problem 2 the speed-up due to caching is significant, reaching about $2.5\times$ on the larger domains such as Game of Life, i.e. where it is most needed. On domains with big branching factors, e.g. Recon, caching makes the difference between success and utter failure.

Cyclic Policies. The cyclic policies evaluated by GLUTTON are seemingly so simple that it is hard to believe they ever beat the policy produced after several minutes of GLUTTON's "honest" planning. Indeed, on most problems GLUTTON does not resort to them. Nonetheless, they turn out to be useful on a surprising number of problems. Consider, for instance, Figures 4 and 5. They compare the normalized scores of GLUTTON's "smart" policy produced at IPPC-2011, and the best primitive cyclic policy across various problems from these domains.

On Game of Life (Figure 4), GLUTTON's "smart" policies for the easier instances clearly win. At the same time, notice that as the problem size increases, the quality of cyclic policies nears and eventually exceeds that of the "smart" policies. This happens because the increase in difficulty of problems within the domain is not accompanied by a commensurate increase in time allocated for solving them. Therefore, the quality of the "smart" policy GLUTTON can come up with within allocated time keeps dropping, as seen on Figure 4. Granted, on Game of Life the quality of cyclic policies is also not very high, although it still helps GLUTTON score higher than 0 on all the problems. However, the Traffic domain proves (Figure 5) that even primitive cyclic policies can be very powerful. On this domain, they dominate anything GLUTTON can come up with on its own, and approach in quality the policies of PROST, the winner on this set of problems. It is due to them that GLUTTON performed reasonably well at IPPC-2011 on Traffic. Whether the success of primitive cyclic policies is particular to the structure of IPPC-2011 or generalizes beyond them is a topic for future research.

Comparison with PROST. On nearly all IPPC-2011 problems, either GLUTTON or PROST was the top performer, so we compare GLUTTON's performance only to PROST's. When looking at the results, it is important to keep in mind one major difference between these planners. PROST (Keller and Eyerich 2012) is an online planner, whereas GLUTTON is an offline one. When given n seconds to solve a problem, GLUTTON spends this entire time trying to solve the problem from the initial state for as large a horizon as possible (recall its reverse iterative deepening strategy).

Instead, PROST plans online, only for states it gets from the server. As a consequence, it has to divide up the n sec-

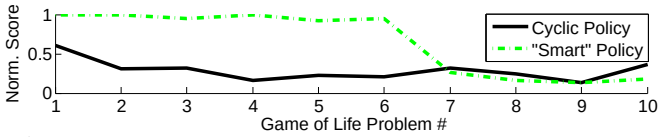

Figure 4: Normalized scores of the best primitive cyclic policies and of GLUTTON's "smart" policies on Game of Life.

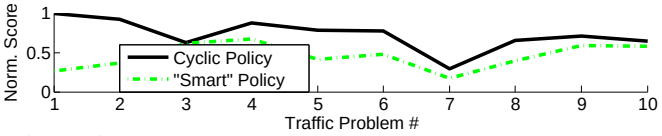

Figure 5: Normalized scores of the best primitive cyclic policies and of the "smart" policies produced by GLUTTON on Traffic.

onds into smaller time intervals, each of which is spent planning for a particular state it receives from the server. Since these intervals are short, it is unreasonable to expect PROST to solve a state for a large horizon value within that time. Therefore, PROST explores the state space only up to a preset depth from the given state, which, as far as we know from personal communication with PROST's authors, is 15. Both GLUTTON's and PROST's strategies have their disadvantages. GLUTTON may spend considerable effort on states it never encounters during the evaluation rounds. Indeed, since each IPPC-2011 problem has horizon 40 and needs to be attempted 30 times during evaluation, the number of distinct states for which performance "really matters" is at most $30 \cdot 39 + 1 = 1171$ (the initial state is encountered 30 times). The number of states GLUTTON visits and tries to learn a policy for during training is typically many orders of magnitude larger. On the other hand, PROST, due to its artificial lookahead limit, may fail to produce good policies on problems where most high-reward states can only be reached after > 15 steps from (s_0, H), e.g., goal-oriented MDPs.

During IPPC-2011, GLUTTON used a more efficient strategy of allocating time to different problems than simply dividing the available time equally, as we did for the ablation studies. Its high-level idea was to solve easy problems first and devote more time to harder ones. To do so, GLUTTON first solved problem 1 from each domain. Then it kept redistributing the remaining time equally among the remaining problems and picking the next problem from the domain whose instances on average had been the fastest to solve. As a result, the hardest problems got 40-50 minutes of planning.

Figure 6 shows the average of GLUTTON's and PROST's normalized scores on all IPPC domains, with GLUTTON using the above time allocation approach. Overall, GLUTTON is much better on Navigation and Crossing Traffic, at par on Elevators, slightly worse on Recon and Skill Teaching, and much worse on Sysadmin, Game of Life, and Traffic.

As it turns out, GLUTTON's success and failures have fairly a clear pattern. Sysadmin, Game of Life, and Traffic, although very large, do not require a large lookahead to produce a reasonable policy. That is, although the horizon of all these MDPs is 40, for many of them the optimal policy with a lookahead of only 4-5 has a good performance. As a result, GLUTTON's attempts to solve the entire problem offline do not pay off — by timeout, GLUTTON learns how to behave well only in the initial state and many of the states at depths 2-3 from it. However, during policy execution it often ends up in states it failed to even visit during the training stage, and is forced to resort to a default policy. It

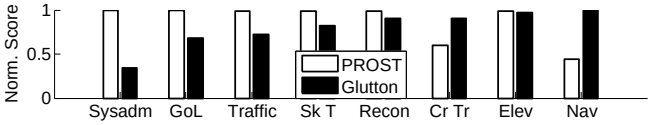

Figure 6: Average normalized scores of GLUTTON and PROST on all of the IPPC-2011 domains.

fails to visit these states not only because it subsamples the transition function, but also because many of them cannot be reached from the initial state within a small number of steps. On the other hand, PROST copes with such problems well. Its online nature ensures that it does not waste as much effort on states it ends up never visiting, and it knows what to do (at least to some degree) in all the states encountered during evaluation rounds. Moreover, trying to solve each such state for only horizon 15 allows it to produce a good policy even if it fails to converge within the allocated time.

Recon, Skill Teaching, and Elevators are smaller, so before timeout, GLUTTON manages to either solve them completely or explore their state spaces to significant horizon values and visit most of their states at some distance from s_0. Therefore, although GLUTTON still has to use default policies in some states, in most states it has a good policy.

In Navigation and Crossing Traffic, the distance from (s_0, H) to the goal (i.e., highest-reward states) is often larger than PROST's lookahead of 15. This means that PROST often does not see goal states during the learning stage, and hence fails to construct a policy that aims for them. Contrariwise, GLUTTON, due to its strategy of iterative deepening, can usually find the goal states and solve for a policy that reaches them with high probability.

Conclusion

Unlike previous planning competitions, IPPC-2011 emphasized finite-horizon reward maximization problems with large branching factors. In this paper, we presented LR^{2}TDP, a novel LRTDP-based optimal algorithm for finite-horizon problems centered around the idea of reverse iterative deepening and GLUTTON, our LR^{2}TDP-based planner at IPPC-2011 that performed well on these challenging MDPs. To achieve this, GLUTTON includes several important optimizations — subsampling the transition function, separating out natural dynamics, caching the transition function samples, and using primitive cyclic policies as the default solution. We presented an experimental evaluation of GLUTTON's core ideas and a comparison of GLUTTON to the IPPC-2011 top-performing planner, PROST.

GLUTTON and PROST have complementary strengths, with GLUTTON demonstrating superior performance on problems with goal states, although PROST won overall. Since PROST is based on UCT and GLUTTON — on LRTDP, it is natural to ask: is UCT a better algorithm for finite-horizon MDPs, or would LR^{2}TDP outperform UCT if LR^{2}TDP were used online? A comparison of an online version of GLUTTON and PROST should provide an answer.

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

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

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

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control*. Athena Scientific.

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

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

Feng, Z.; Hansen, E. A.; and Zilberstein, S. 2003. Symbolic generalization for on-line planning. In *UAI*, 109–116.

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

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

Mausam, and Weld, D. S. 2004. Solving concurrent markov decision processes. In *AAAI'04*.

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

Proper, S., and Tadepalli, P. 2006. Scaling model-based average-reward reinforcement learning for product delivery. In *ECML*, 735–742.

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

Sanner, S. 2010. Relational dynamic influence diagram language (RDDL): Language description. http://users.cecs.anu.edu.au/s̃sanner/IPPC_2011/RDDL.pdf.

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

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

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