Memory-Efficient Symbolic Online Planning for Factored MDPs

Aswin Raghavan
School of EECS
Oregon State University
Corvallis, OR, USA
nadamuna@eecs.orst.edu

Roni Khardon
Department of Computer Science
Tufts University
Medford, MA, USA
roni@cs.tufts.edu

Prasad Tadepalli
School of EECS
Oregon State University
Corvallis, OR, USA
tadepall@eecs.orst.edu

Alan Fern
School of EECS
Oregon State University
Corvallis, OR, USA
afern@eecs.orst.edu

Abstract

Factored Markov Decision Processes (MDP) are a de facto standard for compactly modeling sequential decision making problems with uncertainty. Offline planning based on symbolic operators exploits the factored structure of MDPs, but is memory intensive. We present new memory-efficient symbolic operators for online planning, prove the soundness of the operators, and show convergence of the corresponding planning algorithms. An experimental evaluation demonstrates superior scalability on benchmark problems.

1 INTRODUCTION

The success of online planning in Markov Decision Processes (MDPs) depends crucially on the extent to which the information gathered from search is generalized to unseen states. In the absence of generalization and heuristic guidance, the planner must explore the entire reachable state space. In factored MDPs, the size of the state and action spaces is exponential in the number of state and action variables, causing algorithms that are polynomial in the number of states and/or actions to be impractical. The current state-of-the-art online algorithms based on e.g. , Real-Time Dynamic Programming (RTDP) (Barto *et al.*, 1995) and UCT (Kocsis & Szepesvári, 2006), search in terms of atomic or "flat" states. They are unable to take advantage of the factored structure present in the MDP descriptions which allows strong generalization among states (Boutilier *et al.*, 1999).

In contrast, symbolic decision theoretic planners, such as SPUDD (Hoey *et al.*, 1999), do take advantage of the factored structure. These algorithms interleave Dynamic Programming (DP) (Bertsekas & Tsitsiklis, 1996) updates with steps of model minimization (Givan *et al.*, 2003) in a selected representation such as Algebraic Decision Diagrams (ADD) (Bahar *et al.*, 1993). These offline planners sometimes scale to large MDPs, but depend on compactly representing the optimal value function of the entire MDP (Hoey *et al.*, 1999). Due to this requirement, these algorithms exceed practical memory and time limits in many problems of interest.

Symbolic Real-Time Dynamic Programming (sRTDP) (Feng *et al.*, 2002; Feng & Hansen, 2002) aims to combine the benefits of the symbolic methods and online planning by incorporating symbolic state generalization into the computation of the online planner. This effectively imposes state constraints capturing reachability from the current world state into the symbolic computation. However, sRTDP is a general framework, and its performance is sensitive to the definition of generalized states. Existing definitions in prior work lead to algorithms that exceed memory limits in many cases. Despite the aim for generalization, the resulting planner is often inferior to the corresponding algorithms working in the flat state space (e.g. RTDP).

Our main contribution is the introduction of new symbolic generalization operators that guarantee a more moderate use of space and time, while providing non-trivial generalization. Using these operators, we present symbolic online planning algorithms that combine forward search from an initial state with backwards generalized DP updates. The first algorithm, Path Dynamic Programming (PDP) (Section 3.1), samples fixed-length trajectories by acting greedily and refines *one path* in an ADD for each visited state. It uses either an operator based on value invariance (PDP-V) or one based on policy invariance (PDP-π). Both operators yield anytime algorithms that guarantee convergence to the optimal value and action for the current world state, while maintaining bounded growth in the size of the symbolic representation.

In spite of the slow growth of the value function representation, intermediate computations in PDP leading to that representation can potentially exceed memory limits. This motivates our second operator that performs a more careful control of space in its generalization. The resulting planning algorithm, Pruning Path Dynamic Programming (pPDP) (Section 4), applies the pruning procedure of

Raghavan *et al.* (2013) to control the size of intermediate results. The algorithm is convergent and provides generalization only when it does not increase space requirements compared to flat state search. Thus, it is guaranteed not be worse than flat state space search. It is the first symbolic algorithm to yield a sound generalization while guaranteeing not to use more memory than flat RTDP.

We empirically demonstrate (Section 5) the performance of PDP and pPDP on three benchmark domains from the recent International Probabilistic Planning Competitions (IPPC), where the proposed algorithms scale significantly better than previous results.

2 PROBLEM FORMULATION

2.1 Algebraic Decision Diagrams (ADD)

An Algebraic Decision Diagram (ADD) (Bahar *et al.*, 1993; Bryant, 1992) represents a real-valued function $\mathcal{B}^n \rightarrow \mathcal{R}$ over n boolean variables compactly in the form of a rooted Directed Acyclic Graph (DAG), where each interior node has an associated test variable and two outgoing edges labeled by true or false that lead to its children. An example ADD is shown on the right of Figure 1. Every assignment of variables to truth values traces a unique path to a leaf from the root. Each leaf node contains the value of the ADD function for all assignments $\mathbf{x}$ that reach that node. If D is the ADD, $D[\mathbf{x}]$ represents its evaluation on $\mathbf{x}$. In the example, the assignment reboot_c1=0, running_c1=0, and running_c1'=0 leads to the value 0.95. A Binary Decision Diagram (BDD) is an ADD with 0/1 leaves.

We assume that the ADD is *ordered* in that there is a fixed total ordering on the variables that all directed paths in the ADD follow. Ordered ADDs have a canonical representation for any function (although their compactness depends on the ordering) and they support polynomial time operations over the functions they represent. The unary "restrict operator" fixes the value of a variable $\mathbf{x}$ to x or $\bar{x}$ in ADD D and returns a new ADD denoted by $D_{\downarrow \mathbf{x}}$. In general, a binary ADD operation $C = A \; op \; B$ gives an ADD such that $C[\mathbf{x}] = A[\mathbf{x}] \; op \; B[\mathbf{x}]$ for every $\mathbf{x}$. The result C is computed symbolically and in polynomial time in the size of the ADDs A and B. Any binary operation can be used as *op*, for example, $\{+, -, \times, \div, \max, \min\}$. Marginalization operations such as $\max_V D, \min_V D, \sum_V D$ are defined naturally over all possible restrictions of D over values of variables in the set V, e.g. $\max_x D \equiv \max(D_{\downarrow x}, D_{\downarrow \bar{x}})$. We also use the operator $\oplus_C$ for ADDs, where $A \oplus_C B = (1 - C)A + CB$, for a binary valued ADD C. That is, the result takes the values from B if C is true and otherwise from A. This is similar to the ITE(C,B,A) notation in the BDD literature. This operation can cause merging of paths within the ADD due to reduction, e.g. if A and B agree on many values.

A partial assignment is a truth assignment to a subset of variables in D. An assignment is full if it assigns values to all variables. An extension of a partial assignment is a full assignment which is consistent with it. Every path in the ADD from the root to the leaf defines a partial assignment over the internal variables in that path. For example, the path to 0.95 traversed in the example above, defines a partial assignment to three of the variables but leaves other variables, for example running_c2, unspecified.

The *path function* for an ADD D maps a full assignment $\mathbf{x}$ to the partial assignment defined by the path traced by $\mathbf{x}$ in D and is denoted by $\Phi(D, \mathbf{x})$. The path function is represented as an ordered BDD that returns 1 for all assignments consistent with the partial assignment and 0 otherwise. For an assignment $\mathbf{x}$, ADD D and $\phi = \Phi(D, \mathbf{x})$, let $\epsilon(\phi)$ denote the set of all extensions of ϕ.

Two additional transformations are useful. The first converts a BDD B to an ADD D by mapping the 0-leaf in B to $-\infty$, denoted by $D = \underline{B}$. The second, a complementing operation, converts an ADD D to a BDD B by mapping the 0-leaf in D to 1 and all other real-valued leaves to 0, denoted by $B = \overline{D}$.

2.2 Factored State and Action MDPs

An MDP (Puterman, 2014) is a tuple $(\mathcal{S}, \mathcal{A}, T, R)$ where $\mathcal{S}$ is a finite state-space, $\mathcal{A}$ is a finite action space, $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ denotes the transition function $T(s, a, s') = Pr(s'|s, a)$, $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R}$ denotes the immediate reward of taking action a in state s. In this paper, we focus on finite-horizon planning where the goal is to maximize the expected cumulative reward over a specified horizon H. A non-stationary policy $\pi = (\pi_1, \ldots, \pi_H)$ is a sequence of mappings such that each $\pi_i : \mathcal{S} \rightarrow \mathcal{A}$ determines the action to take in a state when there are i steps-to-go. The value function of a policy π with i steps-to-go is denoted by $V_i^\pi(s)$, which gives the expected total reward of following π starting in state s for i steps. The value function of the optimal i-horizon policy is denoted by $V_i^*(s)$.

In a factored MDP (Boutilier *et al.*, 1999) the state space $\mathcal{S}$ and action space $\mathcal{A}$ are specified by finite sets of state variables $\mathbf{X} = (X_1, \ldots, X_l)$ and action variables $\mathbf{A} = (A_1, \ldots, A_m)$. We will assume that the variables are discrete and binary so that $|\mathcal{S}| = 2^l$ and $|\mathcal{A}| = 2^m$.

The transition and reward functions are defined in terms of state and action variables using a Dynamic Bayesian Network (DBN), a two-time-step graphical model that captures the variables at time t that influence the value of each X_i' at time $t + 1$ via the conditional probability functions $P(X_i'|\text{Parents}(X_i'))$. The reward function is represented as a node at time $t+1$. Following Hoey *et al.* (1999), the functions are represented using ADDs (Bahar *et al.*, 1993) to

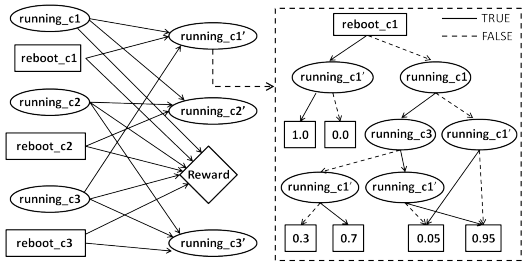

Figure 1: Example of a Factored MDP with Factored Actions. The ADD on the right shows the conditional probability distribution for running_c1.

compactly capture sparsity and context-specific dependencies often found in factored MDPs (Boutilier *et al.*, 1999).

For example, Figure 1 shows a DBN for the SysAdmin problem (Guestrin *et al.*, 2001) (see Section 5). The DBN encodes that the computers 'c1', 'c2' and 'c3' are arranged in a directed ring so that the running status of each is influenced by its reboot action and the status of its predecessor. The ADD (right panel) shows that the proposition 'running_c1' cannot become false if it is 'rebooted', and otherwise the next state depends on the status of the neighbors. If currently running, it fails w.p. 0.3 if its neighboring computer 'c3' is not operational, and w.p. 0.05 otherwise. A failed computer becomes operational w.p. 0.05.

In previous work we generalized the symbolic approach of SPUDD (Hoey *et al.*, 1999) and introduced algorithms that handle factored states and factored actions. This includes Factored Action Regression (FAR) (Raghavan *et al.*, 2012), a symbolic version of Value Iteration (VI). By leveraging ADD operations (near-)optimal value functions/policies are *deduced* in propositional logic without enumerating the states and actions. Symbolic VI using FAR works by iterating Equation 1,

$$Q = [R + \gamma \sum_{X'_1} P^{X'_1} \times \ldots \sum_{X'_l} P^{X'_l} \times (V_n)'] \qquad (1)$$

with $V_0 = 0$ and $V_{n+1} = \max_{A_1 \ldots A_m} Q$. Here $(V_n)'$ swaps the state variables X in the diagram V_n with next state variables X', each summation computes the expectation over one X'_i and marginalizes (and removes) X'_i from the ADD. Therefore, the ADD V_{n+1} is the result of one backward DP update for all states.

Below we also use a decision diagram representation of policies. Following (Raghavan *et al.*, 2013) the policy is represented as a BDD over state and action variables and evaluates to 1 on the policy action for a given state. This can be calculated using diagram operations as $\pi_{n+1} = \overline{\max_{\mathbf{A}} Q - Q}$, where $\overline{D}$ is the complementing operator (Section 2.1). Since $\max_A Q - Q \geq 0$ everywhere and > 0 on paths that include suboptimal actions this identifies the greedy policy with respect to Q. Below we slightly abuse notation and denote this operation as $\pi_{n+1} = \arg\max_{\mathbf{A}} Q$.

Unlike other approaches to factored MDPs based on function approximation (Guestrin *et al.*, 2001; Koller & Parr, 2000), the symbolic algorithms do not require engineered basis functions but rely on compactly representing value functions and policies. FAR and Opportunistic Policy Iteration (Raghavan *et al.*, 2013), a memory-efficient symbolic variant of Modified Policy Iteration (Puterman & Shin, 1978) using FAR, are currently the most effective symbolic algorithms for factored MDPs. As mentioned above, in many cases these methods fail due to the memory exhaustion caused by progressively larger ADDs V_n as n increases. Symbolic Online Planning aims to reduce the memory usage by restricting to the state space reachable from the current world state.

2.3 Symbolic Online Planning

In order to facilitate the presentation of online symbolic methods we next consider an update that explicitly controls which states are updated. Let X be a BDD representing some set of states, and let $\underline{X}$ be the corresponding constraint ADD mapping states in X to 1, and states not in X to $-\infty$. The operator $\mathcal{B}^*(V, X)$ performs an exact update (a Bellman backup) for the values of states in X and copies the values from V for other states using $\oplus$. This operator can be implemented via the ADD expression:

$$\mathcal{B}^*(V, X) \triangleq V \oplus_X [\max_A (R + \gamma E_{X'_1} \ldots E_{X'_l}(\underline{X} \times V'))] \quad (2)$$

where multiplication by $\underline{X}$ is not necessary for correctness but it helps control space. The product of V' with $\underline{X}$ fixes the value of states that are not in the set X to $-\infty$. Therefore, the sum and product operations also result in $-\infty$ without increasing the size of ADDs for these states. The constraint $\underline{X}$ can be pushed inside the summations due to the distributive property of ADD operations (Raghavan *et al.*, 2013). Note that sRTDP uses an operator equivalent to $\mathcal{B}^*(V, X)$ via a more memory intensive ADD expression.

We can now explain more general algorithms. Let X denote a set of states or "a generalized state" and s denote an atomic state or "flat state". FAR (Equation 1) uses the backup of Equation 2 with $X = 1$, which means it updates the values of all states.

Real-Time Dynamic Programming (RTDP) (Barto *et al.*, 1995) is an online planning algorithm that only updates the values of reachable states. RTDP works by simulating trajectories from a starting state and setting $X = s$ for each encountered flat state s. While each update is time and space-efficient, convergence can take a long time in factored MDPs.

sRTDP (Feng *et al.*, 2002) generalizes RTDP updates, uses an update similar to Equation 2 by setting X to an arbitrary set of states. The set X is defined by an equivalence relation over states (e.g. the bisimulation relationship (Givan *et al.*, 2003)), which is in practice, heuristically chosen

to trade off the efficiency of the update with the benefit of generalization. An unwise choice of X in Equation 2 can lead to unreasonable space (or time) requirements. Despite its goal of generalization, the performance of sRTDP can be inferior to RTDP in the online setting where both space and time are limited.

Next we describe our formulations of generalized states X that lead to efficient updates both in time and space. Convergence of RTDP (and sRTDP) can be retained if X includes state s. Efficiency comparable to RTDP can be achieved if the generalized value functions and policies can be captured with about the same amount of memory. We give equivalence relationships that are more restricted than bisimulation (Li *et al.*, 2006), and lead to efficient symbolic online algorithms.

3 PDP

Our algorithms are instantiations of Trial-Based Real Time Dynamic Programming (RTDP) (Barto *et al.*, 1995; Keller & Helmert, 2013) with a particular generalized backup function and a fixed trial length. They have two parameters : a lookahead integer $H > 0$ that is the length of trajectories and a real valued ε that controls approximation error in the values of states.

In contrast to most online planners which use a tabular representation, we maintain one value ADD $V^d, d \in [0, H-1]$ per level of the lookahead tree. We chose this over a global ADD (as in sRTDP) because it allows representing non-stationary policies and value functions compactly, as well as allowing different levels of approximation per level, e.g., for increasingly coarse representations of the future . In addition, to simplify the presentation, we explicitly maintain a policy BDD π^d, for each level d.

Our algorithms start from an initial state and sample a trajectory $\langle s_0, a_0, \ldots, a_{H-2}, s_{H-1}\rangle$ by following the greedy policy $\pi^0, \ldots, \pi^{H-1}$. Then, the ADDs are updated in the backward direction: V^{H-1} is updated from the ADD 0, V^{H-1} is used to update V^{H-2} and so on till V^0, which includes s_0 but may be more general.

The general pseudocode for all the algorithms is shown in Figure 2. They have an update of the form $V = V \oplus_M \max_{\mathbf{A}} Q$. The algorithm requires three properties: (A) M is a *path* over state variables (hence the name Path Dynamic Programming), (B) Q is an ADD over state and action variables with updated values for a super-set of the states in M. (C) The current state s_i is included in path M.

Proposition 1. *Any instance of the PDP algorithm (Figure 2) satisfying properties B and C converges to the optimal value (and action) for* s_0.

Proof (sketch): The proof directly follows from the convergence of Trial-Based RTDP (Barto *et al.*, 1995). First, it can be shown that PDP maintains the invariants $V_i \geq V_i^*$ for all i. Second, it uses greedy action selection to sample trajectories and each trajectory always includes the state s_0. Hence if each update is equivalent to DP update on some states, the value and policy at s_0 converge to their optimal values. □

Algorithm 3.1: (ADDs $V^0, \ldots, V^{H-1}$, $\pi^0, \ldots, \pi^{H-1}$)

Initialize each $V^i \leftarrow (H - i + 1)R_{\max}$, $\pi^i \leftarrow$ NoOp.
Sample trajectory $\langle s_0, a_0, \ldots, a_{L-1}, s_L\rangle$ using π.
for $i \leftarrow L - 1$ **downto** 0
do
- **if** PDP-V then $(Q, M) \leftarrow$ Equations 4, 5
- **if** PDP-π then $(Q, M) \leftarrow$ Equations 6, 8
- **if** pPDP then $(Q, M) \leftarrow$ Equations 9, 11
- $V^i \leftarrow V^i \oplus_M \max_{\mathbf{A}} Q$
- $\pi^i \leftarrow \pi^i \oplus_M \arg\max_{\mathbf{A}} Q$

if beyond time or trajectory budget
then return $\pi^0(s_0)$

Figure 2: Pseudocode for Path Dynamic Programming (PDP). $A \oplus_X B = (1 - X)A + XB$.

3.1 PDP-V

The main idea for PDP is to restrict the update to one path in the ADD, instead of one state in RTDP, and PDP-V uses one path in the value ADD. However, this requires a careful control of the set M as shown below.

$$\mathcal{B}(V, s) = V \oplus_M \max_{\mathbf{A}} Q \tag{3}$$

$$Q = R + \sum_{X_1'} P_1 \times \ldots \sum_{X_l'} P_l \times (\underline{\Phi(V, s)} \times V') \tag{4}$$

$$M = \Phi(\max_{\mathbf{A}} Q, s) \wedge \Phi(V, s) \tag{5}$$

For a given state s and value ADD V, the values of all states that are extensions of the current path $\Phi(V, s)$ are updated in the ADD Q (Equation 4). The ADD $\max_{\mathbf{A}} Q$ has updated values for the path extensions of $\Phi(V, s)$ and $-\infty$ otherwise. But using this update with $M = \Phi(V, s)$ might lose compactness if many of the extensions of $\Phi(V, s)$ have different values. Additionally, the new path $\Phi(\max_{\mathbf{A}} Q, s)$ can be shorter than $\Phi(V, s)$ whereas only the extensions $\Phi(V, s)$ have correctly updated values. Sound generalization and compactness are both achieved by restricting the update to the path refinement of $\Phi(V, s)$ by setting M to be the intersection of the set of states that share the same path as state s before and after the update. This is the main difference between PDP and sRTDP. It guarantees that the updated V has the same number of leaves as the flat state update, an important guarantee for a symbolic method.

Proposition 2. *Let V be an ADD, s a state, $W = \mathcal{B}(V, s)$*

and M the path according to Equation 5.
(a) For all states $q \in \epsilon(M)$, $W[q] = \mathcal{B}^*(V, q)[q]$.
(b) W grows by at most one leaf node over V.

Proof (sketch): (a) follows because ADD Q is a sound update for states in $\Phi(V, s)$ (because the constraint $\Phi(V, s)$ can be pushed inside the summation as in Raghavan *et al.* (2013)). The BDD M represents a subset of states that satisfy $\Phi(V, s)$ due to Equation 5[1]. (b) is true because the path M leads to a leaf in $\max_{\mathbf{A}} Q$. For paths in $1 - M$ the values are copied from V and do not add leaves to W. □

The first part of Proposition 2 guarantees the convergence of PDP-V according to Proposition 1. In practice, it is observed that the paths in symbolic VI often remain unchanged between consecutive iterations while the values have not converged. PDP-V updates these efficiently and succinctly, gaining a speedup proportional to the number of states in the path. However, in order to find the mask M in PDP-V we have to calculate updated values for all extensions of $\Phi(V, s)$ in Equation 4, and in some cases this preparatory step exceeds memory limits. Section 4 gives an algorithm that does not have this disadvantage.

3.2 PDP-π

PDP-π similarly restricts the update to one path. However, is appeals to the notion of policy irrelevance (Jong, 2005; Li *et al.*, 2006; Hostetler *et al.*, 2014), that captures states having the same optimal action. Recall that PDP maintains a policy representation in addition to the value ADDs. PDP-π updates states that share a path in π before and after a DP update to the policy. The memory efficiency of path refinement is retained with respect to the policy representation.

PDP-π starts with a trivial policy (e.g. NoOp) and refines the policy for generalized states visited by trajectories. In this way, PDP-π behaves more like a policy search method. It is well known that in some cases paths in the policy BDD remain unchanged during iterations of symbolic VI even though the values keep changing. PDP-π captures these succinctly (Section 5).

Let $\pi(s)$ be the policy action, i.e., a complete assignment to action variables for state s according to BDD π, $\pi(s) = \arg\max_{\mathbf{A}} \pi_{\downarrow s}$. In case of a tie, some action variables are set to false (including the case when they are unspecified by $\pi_{\downarrow s}$). Let $\Phi_\pi(s)$ be the path over state variables according to the greedy action in π, $\Phi_\pi(s) = \Phi(\pi, s \wedge \pi(s))$. The update is similar to PDP-V except it uses Φ_π and $\arg\max$ instead.

$$Q = (R + \sum_{X_1'} P_1 \times \ldots \sum_{X_l'} P_l \times (\underline{\Phi_\pi(s)} \times V')) \quad (6)$$

$$\mu = \arg\max_{\mathbf{A}} Q \quad (7)$$

$$M = \Phi_\mu(s) \wedge \Phi_\pi(s) \quad (8)$$

The ADD Q contains updated values for the states in $\Phi_\pi(s)$ rather than $\Phi(V, s)$ as in PDP-V. The mask M uses $\mu = \arg\max_{\mathbf{A}} Q$ to denote the greedy policy BDD extracted from Q. Finally, the policy BDD is updated as $\pi = \pi \oplus_M \arg\max_{\mathbf{A}} Q$. Clearly, the property in Proposition 2 (a) holds here as well and therefore by Proposition 1 the algorithm PDP-π converges.

4 pPDP

The idea in pPDP is to repeatedly prune the intermediate ADDs of Equation 4 so that the ADD Q has space complexity no larger than the DP update of a flat state. We use the pruning operator proposed in (Raghavan *et al.*, 2013) to control the size of the ADD. Briefly, the pruning operator denoted by $\mathcal{P}(D, C)$ for an ADD D and a constraint C represented as a BDD returns an ADD which is no larger than D. The result of pruning removes some of the paths from D that violate the constraint C but not all.

Lemma 1. *(Raghavan* et al.*, 2013). Let $G = \mathcal{P}(D, \pi)$ then*
(1) Every path in G is a sub-path of a path in D.
(2) If a path p in G does not lead to $-\infty$, then for all extensions $y \in \epsilon(p), G(y) = D(y)$.
(3) If a path p in G does lead to $-\infty$, then for all extensions $y \in \epsilon(p)$ either $\pi(y) = -\infty$ or $D(y) = -\infty$.

Part (1) gives a strong memory guarantee that G is no larger than D. Pruning accomplishes this by leaving some paths in G unchanged if only some (not all) of their extensions violate the constraint. pPDP uses the flat state $C = s$ as the constraint. Let $\mathcal{P}_s(D)$ denote $\mathcal{P}(D, s)$ for any ADD D and a flat state s.

$$Q = \mathcal{P}_s(R + \mathcal{P}_s(\sum_{X_1'} P_1 \times \ldots \mathcal{P}_s(\sum_{X_l'} P_l \times V'))) \quad (9)$$

As the expectation is computed over X_i', state and action variables are introduced into the paths of V'. The paths that do not cover the current state are pruned and point to $-\infty$. Hence it is efficient to compute the ADD Q in memory.

Proposition 3. *(1) Q contains $O(2^m)$ paths over state and action variables that do not lead to $-\infty$.*
(2) Every path p that does not lead to $-\infty$ is a DP update for the Q-value of all states and actions in p.

Proof (Sketch) : (1) is due to using the state s as the constraint. Any path that has assignments to state variables

[1]Note that the proposition does not hold for the path $\Phi(W, s)$ (rather than M) due to the $\oplus$ operator which might merge an updated path with a path that was not updated.

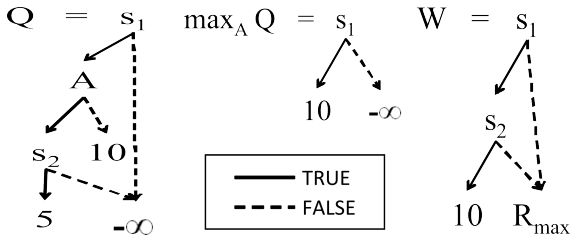

Figure 3: Example illustrating the mask M in pPDP. V is set to $R_{\max}$ initially. Q is computed using Equation 9 for the state $s_1 = 1, s_2 = 1$. The ADD $\max_{\mathbf{A}} Q$ gives an incorrect value for $s_1 = 1, s_2 = 0$, compared to W, the update using PDP-V (Equation 3).

differing from s is pruned and leads to $-\infty$. The paths that do not lead to $-\infty$ have different assignments to action variables for a maximum of $O(2^m)$ paths. (2) is due to part two of Lemma 1 because the pruning operator does not alter the paths that are consistent with state s. $\square$

The pruning operator removes portions of the diagram in subtle ways and therefore we have to be careful in choosing the mask M. Consider using the path $M = \Phi(\max_{\mathbf{A}} Q, s)$ which at first appears to be a natural choice. Unfortunately, this path does not give sound generalization because of the maximization.

To illustrate this, consider the hypothetical diagram Q shown on the left of Figure 3 where the state s assigns $s_1 = 1$ and $s_2 = 1$ and paths disagreeing with this assignment have been replaced with $-\infty$. The diagram $\max_{\mathbf{A}} Q$ is shown on the right and gives a value of 10 for state $s_1 = 1$ and $s_2 = 0$. This is incorrect since the true value of Q from the path $s_1 = 1$, $a = 1$, $s_2 = 0$ can be larger than 10. In $\max_{\mathbf{A}} Q$ the true value of the states on this path is ignored and assumed to be $-\infty$, and therefore the value calculated for the partial assignment $s_1 = 1$ is not correct for all extensions.

To guarantee correctness we require that all the values in the sub-diagram below the node to be different than $-\infty$. Therefore, states whose values are valid in $\max_{\mathbf{A}} Q$ are those where for all actions $\mathbf{A}$, $(Q \neq -\infty)$, denoted by the BDD $\forall_{\mathbf{A}}(Q \neq -\infty)$. In this example, $(Q \neq -\infty)$ when $\{s_1 = 1, A = 0\} \vee \{s_1 = 1, A = 1, s_2 = 1\}$, and quantification yields the mask $M = \{s_1 = 1, s_2 = 1\}$. Note that the BDDs $(Q \neq -\infty)$ and $\forall_{\mathbf{A}}(Q \neq -\infty)$ cannot be zero because all actions are updated in state s. Therefore, in the worst case, the mask M is equal to the state s and the step degenerates to a flat RTDP update. Formally, the operator used in pPDP is

$$\hat{\mathcal{B}}(V, s) = V \oplus_M \mathcal{P}_s(\max_{\mathbf{A}} Q) \quad (10)$$

$$M = \Phi(\forall_{\mathbf{A}}(Q \neq -\infty), s) \quad (11)$$

Proposition 4. *Given an ADD V and state s, let $W = \hat{\mathcal{B}}(V, s)$ as in Equation 10, and let M be the path from 11. Then, $\forall q \in \epsilon(M)$, $W[q] = \mathcal{B}^*(V, q)[q]$.*

The proof follows from the soundness of pruning (Lemma 1) and the fact that all path extensions of M lead to a value not equal to $-\infty$ in Q. Therefore, by Proposition 1 pPDP converges as well. In cases where PDP exceeds memory limits pPDP can capture some of the sound generalizations, precisely those that can be captured without increasing the size of intermediate Q ADD. The only overhead in pPDP is the time required for the pruning operations which is negligible.

5 EXPERIMENTS

We now evaluate the empirical impact of our proposed generalization operators within the family of RTDP-syle algorithms. To do this we compare our algorithms PDP-V, PDP-π, and pPDP to the following baselines: 1) **RTDP(Table)** is a simple table-based implementation of RTDP with state values initialized to $R_{\max}$. 2) **RTDP(ADD)** is like RTDP(Table) (i.e. no state generalization), except that each state backup is done symbolically using the FAR operator. This can be more efficient for factored actions compared to enumerating actions. 3) **sRTDP** (Feng *et al.*, 2002), where our implementation uses FAR for updates in order to exploit factored actions. 4) **LR^{2}TDP** (Kolobov *et al.*, 2012) is an extension of RTDP(Table) to solve finite horizon MDPs using iterative deepening and labeling, which was successful in recent planning competitions. 5) **FAR** (Raghavan *et al.*, 2012) as described above. FAR is limited to 500 minutes of offline planning and then the resulting policy is executed online. This algorithm is only applicable to some of the small problem instances in our experiments and is included to give an optimal baseline value when possible.

All planners were implemented in a common framework, except for LR^{2}TDP, for which we used the publicly available code. For PDP-V and pPDP, we initialized each V^i with $R_{\max}$ (scaled by i). For PDP-π, we initialized π^i to the NoOp policy. Planning domains and problems are specified in the Relational Dynamic Influence Diagram Language (RDDL) (Sanner, 2010), which we convert to an ADD-based representation. The ADD variable ordering puts $parents(X'_i)$ above X'_i, where the X'_is are ordered (ascending) by the number of parents that are action variables. Note that the parents include current state variables and action variables so that this defines an ordering over all variables. In all experiments, our symbolic operators allow an approximation error of $\varepsilon = 0.1$ with the upper bound merging strategy (St-Aubin *et al.*, 2001).

Our experiments below are on 5 problem instances of varying sizes from three domains of the 2011 and 2014 International Probabilistic Planning Competitions (IPPCs). A memory limit of 4G is imposed to restrict the size of the

ADDs, beyond which the planner can no longer proceed which we denote as "EML"(Exceeded Memory Limit). A planner is evaluated on a problem by running 30 trials of horizon 40 and averaging the total reward across the trials. We report the averages and 95% confidence intervals for each problem. Planners use a specified time limit per decision and we give results for different time limits. The value functions and policies are reinitialized after each decision.

Academic Advising Problem: The Academic Advising domain (Guerin *et al.*, 2012), from IPPC 2014, is a stochastic cost minimization problem that models the process of selecting the courses for a student in order to complete degree requirements, where the courses have complex prerequisite structure. The state space encodes which courses have been taken and whether they were passed or not. The actions at each step correspond to selecting one or more courses to take next. We consider two variants of the domain, a non-concurrent variant, which only allows a single course to be selected at each decision point, and a concurrent version, which allows multiple courses to be selected. The dynamics encode the probability that a course is passed if taken. Missing requirements and retaking of courses are penalized.

Figure 4[2] shows the performance vs. time for the different planners on IPPC 2014 problem instances for the non-concurrent and concurrent variants. All algorithms use a planning lookahead horizon of 16 steps. In the non-concurrent variant and shortest time limit (top left panel), we see that sRTDP fails to scale beyond the two smallest problems, and that FAR is able to solve these two problems as well. In larger instances both of these methods EML. In contrast, PDP-V, PDP-π and pPDP are able to give good performance across problem sizes. Moreover, for the smallest instances where FAR is able to compute an optimal policy, these algorithms yield near optimal performance. This result demonstrates the importance of using update operators that attempt to trade-off generalization and memory usage.

The flat search methods RTDP(Table), RTDP(ADD), and LR^{2}TDP do not perform well. For the largest three instances, these methods have a return no better than that of a NoOp policy. This shows the importance of generalization across states in order to achieve good performance in reasonable time.

Comparing performance across time limits (increasing time from left to right) we see the following. The flat search methods are not able to improve by much as the time limit is increased. PDP-V, PDP-π and pPDP also do not improve significantly with more time.Importantly they are able to avoid EML as more trajectories are sampled with larger time limits. PDP-V shows the most improvement on the largest instance as time increases. This shows that, in this domain, the use of generalization by our methods is the dominating factor in improving performance, and is even more effective than increasing the time limit.

Figure 4 (bottom) shows results for the same problem instances, but with concurrency (of 5 for the first instance and 2 for others). Here, both sRTDP and FAR (not shown) EML even for the smallest instances. The flat search methods degrade quickly as the problem instances become larger. Our proposed methods (with one exception) outperform the competitors, especially for the larger instances. The exception is PDP-π on the largest instance, which results in EML after 18 seconds of planning due to the size of the intermediate ADDs in Equation 6. If we increase the time further (not shown here), PDP-V also does EML. On the-other-hand pPDP does not result in EML due to its guarantees on bounding the diagram size (including intermediate diagrams), possibly at the expense of less aggressive generalization.

SysAdmin Problem : SysAdmin (Guestrin *et al.*, 2001) models a computer network with n computers. Computers can fail with some probability, which requires a reboot action to correct. Neighbors of a failed computer have a higher probability of failing. The reward is based on the number of running computers with a cost associated with a reboot action. Unlike the academic advising domain, the number of reachable states in this domain is practically the entire state space. To allow sRTDP to produce non-EML results we consider networks of 10 computers connected in a star network. Following Raghavan *et al.* (2012), we consider problems that vary the maximum number of computers that can be rebooted per decision ($1, 3, 5, 7$, or 10), which gives a progressively growing factored action space.

Figure 5 gives results for three different time-limit settings. Due to the highly random nature of the problem, we used a short lookahead of four steps for all algorithms. The curve above the bar graphs shows the performance of the optimal policy found by FAR.

sRTDP exhibits interesting behavior in this domain. It performs worse than using no state generalization (i.e. RTDP(ADD)) in the first four instances and then optimally for the largest instance. The increased complexity of the sRTDP backup causes poor performance in the smaller action spaces—sRTDP samples fewer trajectories than our algorithms. On the largest instance as the value ADD becomes more compact (more states have similar values), sRTDP is able to exploit generalization. This shows the difficulty of predicting apriori how much space and time the sRTDP generalization operator may require. In this domain, PDP-V and pPDP scale similarly to RTDP(ADD), because the reward function involves counting the number of computers, which makes the path formula $\Phi(R, s)$ the same as s. This means that these algorithms achieve very little state generalization in this domain. However, they

[2]Charts best viewed in color.

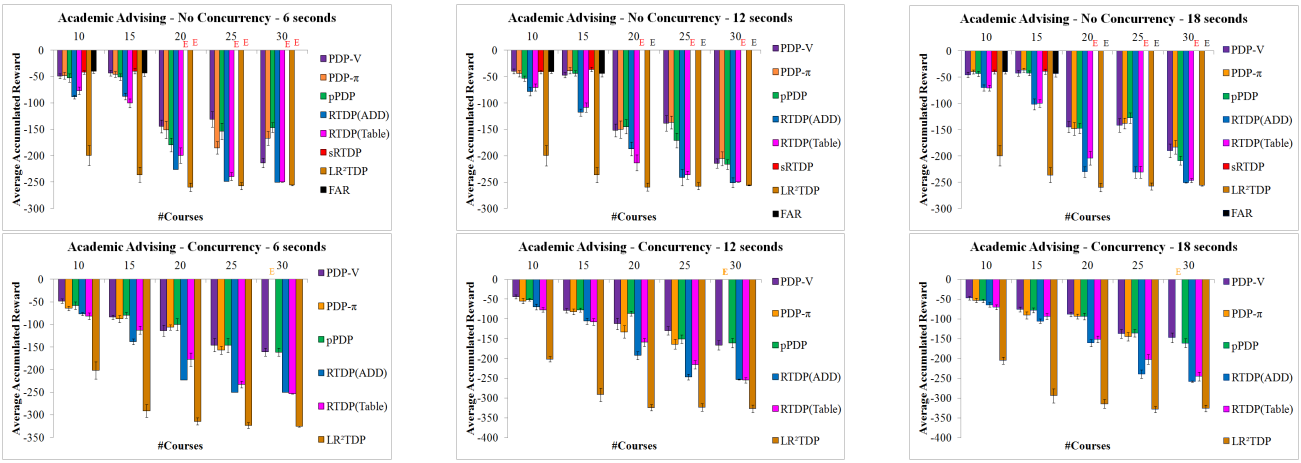

Figure 4: Academic Advising Problem : Performance vs. Time for time limits 6, 12 and 18 seconds per decision without (with) concurrency in the top (bottom) panels respectively. Error bars show 95% confidence intervals. The state space is 2^{2^x}, x is the number of courses shown on the x-axis.

do outperform RTDP(Table) due to the use of the factored FAR backup compared to a backup based on action enumeration.

In this domain, PDP-π clearly outperforms PDP-V and pPDP. In this case, policy irrelevance is able to capture abstract states more succinctly. For example, in the first instance, any state in which the computer at the center of the network is down has the same path formula : $\sim running\_c1 \Rightarrow reboot\_c1$. Note that the values of these states are not equal and depend on the status of other computers. As parallelism increases, nodes near the center get added to these rules regardless of the status of nodes farther away. Clearly, in this domain, generalization based on policy irrelevance is more appropriate than value irrelevance.

Finally we see that as the time limit increases there is a small improvement in performance for most algorithms. It is clear that the increase in performance due to larger time limits is not as significant as using the appropriate generalization mechanism, in this case policy irrelevance.

Crossing Traffic Problem: This IPPC 2011 domain models the arcade game Frogger, where the agent moves in a 2-D grid to cross a road to reach a goal location, while avoiding right-to-left moving cars that enter the road randomly from the rightmost column. The reward is -1 for each move and -40 for collision.

The boolean encoding of this domain has $|S| = O(2^{2(n^2)})$ for an $n \times n$ grid, with two bits per cell for the presence of the agent and car respectively. However, depending on the current location of the agent, many of these bits can be ignored for predicting the optimal value and action.

Figure 6 shows the results for different time limits and problem instances involving 3x3, 4x4, and 5x5 grids. There is larger variance in this domain, compared to the others, due to collisions. We see that sRTDP performs well in the first three instances and is able to improve with more time per decision. However, in instances 4 and 5 sRTDP exceeds memory limits when given more time. PDP-V and PDP-π scale to these instances and times without EML. PDP-π performs better than PDP-V initially but PDP-V is able to improve more than PDP-π with more time per decision. We see that these algorithms outperform RTDP(ADD), showing that generalization is clearly useful in this domain. Again we see that generalizing appropriately is the dominating factor toward performance compared to increasing the time limit. pPDP never performs worse than the flat search methods RTDP(Table) and RTDP(ADD), and outperforms them in some cases. Its less aggressive generalization, however, leads to overall worse performance compared to our other algorithms.

Large instances : The preprocessing of translating RDDL to propositional logic does not scale for the large instances from the IPPC. For the purpose of showing the scaling of our algorithms, we refactored the RDDL domain - by decomposing the "robot-at(x,y)" propositions into two independent propositions "robot-at(x)" and "robot-at(y)" because the actions' effects are independent along x and y dimensions. Figure 8 shows the performance of PDP, PDP-π, pPDP and sRTDP for grids of sizes 6×6 and 7×7.

The algorithms are given much more time per decision to demonstrate the comparative scaling. The charts show the percentage out of 30 trials that the agent reached the goal. We see that in the 6x6 grid (left panel) PDP-V outperforms sRTDP by a large amount. sRTDP is able to scale with time without EML and slowly converges to optimal performance. By comparison, sRTDP does not perform well in the 7 by 7 problem (right panel) whereas PDP-V is able to make progress. PDP-π performs worse (better) than sRTDP in the former (latter) instance. The flat search methods are not able to make any progress in this problem due

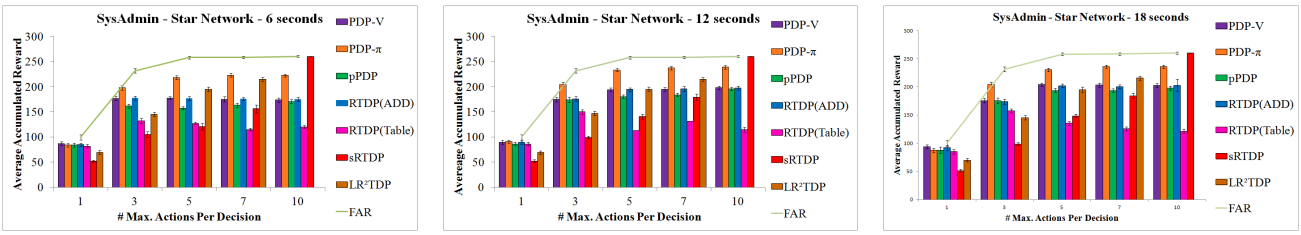

Figure 5: SysAdmin Problem vs. concurrent actions : The state space is 2^{10} and action space is $O(2^x)$, x is the maximum number of parallel actions.

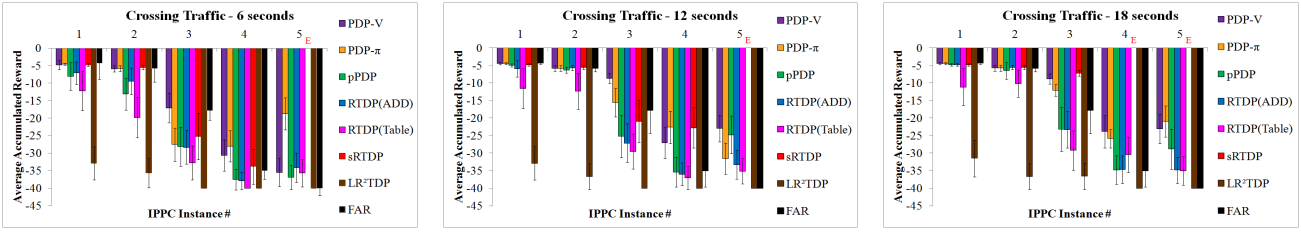

Figure 6: Crossing Traffic Problem : The odd numbered instances have 3×3, 4×4, and 5×5 grids arrival probability of 30%. The even numbered instances have the same grid sizes but with arrival probability of 60%.

Performance vs. Time for 6, 12 and 18 seconds per decision. Error bars show 95% confidence intervals.

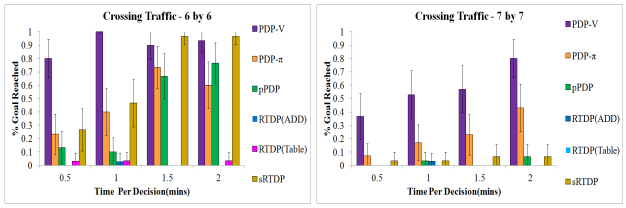

Figure 8: Crossing Traffic (large instances) : Performance vs. Time with 30, 60, 90 and 120 seconds per decision. Error bars show 95% confidence intervals.

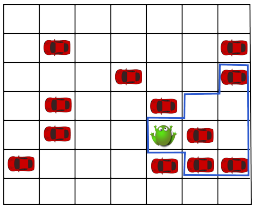

Figure 9: A generalized state discovered by PDP-V in the Crossing Traffic problem. The grid denotes a flat state s_0 and the tiles in blue denote the generalized state $\Phi(V_0, s_0)$.

to the large stochastic branching factor. pPDP is able to improve on the flat search in the first instance but falls back to the flat method in the larger problem.

Finally, we present a generalized state found by PDP-V to illustrate the effectiveness of generalization in these problems. Figure 9 shows an instance of the Crossing Traffic problem. All the cells in the grid, including the location of the agent and each car, constitute a flat state. The cells within blue denote the cells that appear in the path formula $\Phi(V_0, s_0)$ after running PDP-V from s_0. We see that PDP-V is able to ignore the assignments to many cells that are irrelevant for optimal online planning.

6 SUMMARY

We presented the first fully symbolic planning algorithms for factored MDPs that generalize simulated experience soundly and efficiently. This work is orthogonal to research on improving the anytime performance of RTDP algorithms via smart sampling (McMahan *et al.*, 2005; Walsh *et al.*, 2010) and heuristics (Kolobov *et al.*, 2012; Keller & Helmert, 2013). There were several observations from our experimental results. First, in all domains, we saw that using the appropriate form of generalization was the dominating factor towards good performance compared to increasing the time-limit for algorithms without state generalization. Second, we saw that the most appropriate form of generalization can differ across domains and sometimes problem instances within a domain. This suggests that it is fruitful to investigate mechanisms for tuning or selecting among generalization methods. Third, pPDP never exceeded memory limits, while other generalization approaches did occasionally. Further, previous versions of sRTDP, very frequently exceeded memory limits. This suggests that pPDP is perhaps the safest choice for generalization, especially for large problems and short time limits.

Acknowdgements

This work was supported by NSF under grants IIS-0964705 and IIS-0964457.

References

Bahar, R Iris, Frohm, Erica A, Gaona, Charles M, Hachtel, Gary D, Macii, Enrico, Pardo, Abelardo, & Somenzi, Fabio. 1993. Algebraic Decision Diagrams And Their Applications. *In: Computer-Aided Design*.

Barto, Andrew G, Bradtke, Steven J, & Singh, Satinder P. 1995. Learning To Act Using Real-Time Dynamic Programming. *Artificial Intelligence*, **72**(1).

Bertsekas, Dimitri P., & Tsitsiklis, John N. 1996. *Neuro-Dynamic Programming*.

Boutilier, Craig, Dean, Thomas, & Hanks, Steve. 1999. Decision-Theoretic Planning: Structural Assumptions And Computational Leverage. *Journal Of Artificial Intelligence Research (JAIR)*, **11**(1).

Bryant, Randal E. 1992. Symbolic Boolean Manipulation With Ordered Binary-Decision Diagrams. *ACM Computing Surveys (CSUR)*, **24**(3).

Feng, Zhengzhu, & Hansen, Eric A. 2002. Symbolic Heuristic Search for Factored Markov Decision Processes. *In: Eighteenth National Conference on Artificial Intelligence*.

Feng, Zhengzhu, Hansen, Eric A, & Zilberstein, Shlomo. 2002. Symbolic Generalization For Online Planning. *In: Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI)*.

Givan, Robert, Dean, Thomas, & Greig, Matthew. 2003. Equivalence Notions And Model Minimization In Markov Decision Processes. *Artificial Intelligence*, **147**(1).

Guerin, Joshua T, Hanna, Josiah P, Ferland, Libby, Mattei, Nicholas, & Goldsmith, Judy. 2012. The Academic Advising Planning Domain. *WS-IPC 2012*.

Guestrin, Carlos, Koller, Daphne, & Parr, Ronald. 2001. Multiagent Planning With Factored MDPs. *Advances In Neural Information Processing Systems (NIPS)*.

Hoey, Jesse, St-Aubin, Robert, Hu, Alan, & Boutilier, Craig. 1999. SPUDD: Stochastic Planning Using Decision Diagrams. *In: Proceedings Of The Fifteenth Conference On Uncertainty In Artificial Intelligence (UAI)*.

Hostetler, Jesse, Fern, Alan, & Dietterich, Tom. 2014. State Aggregation In Monte Carlo Tree Search. *In: Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI)*.

Jong, Nicholas K. 2005. State Abstraction Discovery From Irrelevant State Variables. *In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI)*.

Keller, Thomas, & Helmert, Malte. 2013. Trial-Based Heuristic Tree Search For Finite Horizon MDPs. *In: Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS)*.

Kocsis, Levente, & Szepesvári, Csaba. 2006. Bandit Based Monte-Carlo Planning. *In: Proceedings of the 17th European Conference on Machine Learning (ECML)*.

Koller, Daphne, & Parr, Ronald. 2000. Policy Iteration For Factored MDPs. *In: Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI)*.

Kolobov, Andrey, Dai, Peng, Mausam, Mausam, & Weld, Daniel S. 2012. Reverse Iterative Deepening for Finite-Horizon MDPs with Large Branching Factors. *In: Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS)*.

Li, Lihong, Walsh, Thomas J, & Littman, Michael L. 2006. Towards a Unified Theory of State Abstraction for MDPs. *In: Proceedings of the Ninth International Symposium on Artificial Intelligence and Mathematics (ISAIM)*.

McMahan, H Brendan, Likhachev, Maxim, & Gordon, Geoffrey J. 2005. Bounded Real-Time Dynamic Programming: RTDP with Monotone Upper Bounds and Performance Guarantees. *In: Proceedings of the 22nd International Conference on Machine Learning (ICML)*.

Puterman, Martin L. 2014. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*.

Puterman, Martin L, & Shin, Moon Chirl. 1978. Modified Policy Iteration Algorithms for Discounted Markov Decision Problems. *Management Science*.

Raghavan, Aswin, Joshi, Saket, Fern, Alan, Tadepalli, Prasad, & Khardon, Roni. 2012. Planning in Factored Action Spaces with Symbolic Dynamic Programming. *In: Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*.

Raghavan, Aswin, Khardon, Roni, Fern, Alan, & Tadepalli, Prasad. 2013. Symbolic Opportunistic Policy Iteration for Factored-Action MDPs. *In: Advances in Neural Information Processing Systems (NIPS)*.

Sanner, Scott. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description. *Unpublished ms. Australian National University*.

St-Aubin, Robert, Hoey, Jesse, & Boutilier, Craig. 2001. APRICODD: Approximate Policy Construction using Decision Diagrams. *Advances in Neural Information Processing Systems (NIPS)*.

Walsh, Thomas J, Goschin, Sergiu, & Littman, Michael L. 2010. Integrating Sample-Based Planning and Model-Based Reinforcement Learning. *In: Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*.