Exact Symbolic Dynamic Programming for Hybrid State and Action MDPs

Zahra Zamani zahra.zamani@anu.edu.au
Scott Sanner ssanner@nicta.com.au
The Australian National University and NICTA,
Canberra, ACT 0200 Australia

Abstract

Many real-world decision-theoretic planning problems can naturally be modeled using continuous states and actions. Problems such as the multi-item Inventory problem (Arrow, Karlin, & Scarf, 1958) have long been solved in the operation research literature using discrete variable MDPs or approximating the optimal value for continuous domains. Other work similar to that of Gaussian control deals with general continuous domains but can not handle piecewise values on the state space. Here we propose a framework to find the first exact optimal piecewise solution for problems modeled using multi-variate continuous state and action variables.

We define a Symbolic Dynamic Programming (SDP) approach using the *case* calculus which provides a closed-form solution for all operations (e.g. continuous maximization and integration). Solutions are provided for discrete action Hybrid(i.e. discrete and continuous) state MDPs (HMDPs) using polynomial transitions with discrete noise and arbitrary reward functions . Solutions to continuous action HMDPs with piecewise linear (or quadratic) discrete noise transition and reward functions are also derived.

Apart from providing the solution to general HMDPs, our other contribution is the compact representation of XADDs - a continuous variable extension of Algebraic decision diagrams (ADDs) - with related properties and algorithms. This allows us to empirically provide efficient results for HMDPs showing the *first optimal automated solution* on various continuous domains.

1. Introduction

Many stochastic planning problems in the real-world involving resources, time, or spatial configurations naturally use continuous variables in their state representation. For example, in the Mars Rover problem (Bresina, Dearden, Meuleau, Ramkrishnan, Smith, & Washington, 2002), a rover must manage bounded continuous resources of battery power and daylight time as it plans scientific discovery tasks for a set of landmarks on a given day. A rover can also have continuous actions in navigating (moving continuously).

Other examples include Inventory Control problems (Arrow et al., 1958) for continuous resources such as petroleum products, a business must decide what quantity of each item to order subject to uncertain demand, (joint) capacity constraints, and reordering costs; and Reservoir Management problems (Lamond & Boukhtouta, 2002), where a utility must manage continuous reservoir water levels in continuous time to avoid underflow while maximizing electricity generation revenue.

Little progress has been made in the recent years in developing *exact* solutions for HMDPs with multiple continuous state variables beyond the subset of HMDPs which have

an optimal *hyper-rectangular piecewise linear value function* (Feng, Dearden, Meuleau, & Washington, 2004; Li & Littman, 2005). Further previous work on *exact* solutions to multivariate continuous state *and* action settings have been limited to the control theory literature for the case of linear-quadratic Gaussian (LQG) control (Athans, 1971). However, the transition dynamics and reward (or cost) for such problems cannot be piecewise — a crucial limitation preventing the application of such solutions to many planning and operation research (OR) problems. Consider the famous OR problem of Inventory Control in (Arrow et al., 1958):

Example (Inventory Control). *Inventory control problems – how much of an item to reorder subject to capacity constraints, demand, and optimization criteria– date back to the 1950's with Scarf's optimal solution to the* single-item capacitated inventory control *(SCIC) problem.* Multi-item joint capacitated inventory (MJCIC) control – *with upper limits on the total storage of all items– has proved to be an NP-hard problem and as a consequence, most solutions resort to some form of approximation (Bitran & Yanasse, 1982; Wu, Shi, & Duffie, 2010); indeed, we are unaware of any work which claims to find an exact closed-form non-myopic optimal policy for* all *(continuous) inventory states for MJCIC under linear reordering costs and linear holding costs.*

Consider a continuous state version of this problem with discrete actions:

Discrete Action Inventory Control (DAIC): A multi-item (K-item) inventory consists of continuous amounts of specific items x_i where $i \in [0, K]$ is the number of items and $x_i \in [0, 200]$. The customer demand is a stochastic boolean variable d for low or high demand levels. Ordering a specific item a_j comes in three discrete ranges of $(low, mid, high)$ where $\{low = 25, mid = 50, high = 75\}$ and $j \in \{1, 2, 3\}$. There are linear reorder costs and also a penalty for holding items. The transition and reward functions have to be defined for each continuous item x_i and action j.

We can also consider the more general continuous action HMDP setting to this problem:

Continuous Action Inventory Control (CAIC): Here in a given time step, the inventory can order any of the i items $a_i \in [0, 200]$ considering the stochastic customer demand.

The transition functions for the continuous state x_i and actions a_i is defined as:

$$x_i' = \begin{cases} d: & x_i + a_i - 150 \\ \neg d: & x_i + a_i - 50 \end{cases} \quad P(d' = true | d, \vec{x}, \vec{x'}) = \begin{cases} d: & 0.7 \\ \neg d: & 0.3 \end{cases} \tag{1}$$

The reward is the sum of K functions $R = \sum_{i=0}^{K} R_i$ as below:

$$R = \begin{cases} \sum_j x_j \geq C & : -\infty \\ \sum_j x_j' \geq C & : -\infty \\ \sum_j x_j \leq C & : 0 \\ \sum_j x_j' \leq C & : 0 \end{cases} + \left(\sum_{i=0}^{K} \begin{cases} d \wedge x_i \geq 150 & : 150 - 0.1 * a_i - 0.05 * x_i \\ d \wedge x_i \leq 150 & : x_i - 0.1 * a_i - 0.05 * x_i \\ \neg d \wedge x_i \geq 50 & : 50 - 0.1 * a_i - 0.05 * x_i \\ \neg d \wedge x_i \leq 50 & : x_i - 0.1 * a_i - 0.05 * x_i \end{cases} + \begin{cases} x_i \leq 0 & : -\infty \\ x_i' \leq 0 & : -\infty \\ x_i \geq 0 & : 0 \\ x_i' \geq 0 & : 0 \end{cases} \right) \tag{2}$$

where C is the total capacity for K items in the inventory. The first and last cases check the safe ranges of the capacity such that the inventory capacity of each item above zero and the sum of total capacity below C is desired. Note that illegal state values are defined

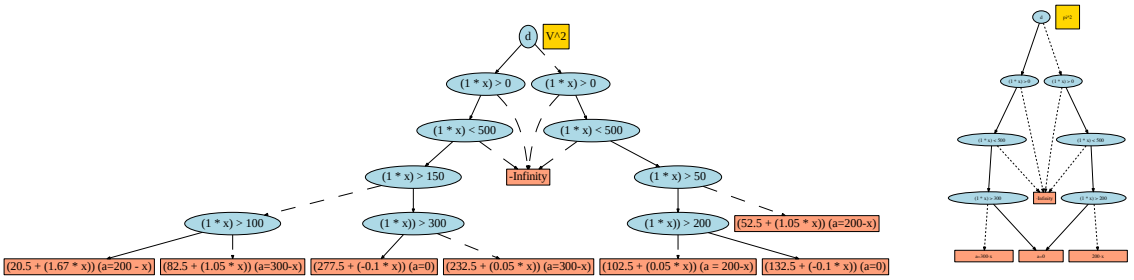

Figure 1: Optimal value function $V^2(x)$ for the CAIC problem represented as an extended algebraic decision diagram (XADD). Here the solid lines represent the *true* branch for the decision and the dashed lines the *false* branch. To evaluate $V^2(x)$ for any state x, one simply traverses the diagram in a decision-tree like fashion until a leaf is reached where the non-parenthetical expression provides the *optimal value* and the parenthetical expression provides the *optimal policy* ($a = \pi^{*,2}(x)$) to achieve value $V^2(x)$ (Left); Optimal policy for the second iteration π^2 consistent with Scarf's policy (Right).

using $-\infty$, in this case having the capacity lower than zero at any time and having capacity higher than that of the total C. If our objective is to maximize the long-term *value* V (i.e.,the sum of rewards received over an infinite horizon of actions), we show that the optimal value function can be derived in closed-form. For a single-item continuous state and action inventory CAIC problem the optimal value function is defined as below:

$$V = \begin{cases} (x < 0 \vee x > 500) & : -\infty \\ d \wedge (0 \leq x \leq 500) \wedge (x \geq 300) & : 277.5 - 0.1 * x \\ d \wedge (150 \leq x \leq 300) & : 232.5 + 0.05 * x \\ d \wedge (100 \leq x \leq 150) & : 20.5 + 1.67 * x \\ d \wedge (0 \leq x \leq 100) & : 82.5 + 1.05 * x \\ \neg d \wedge (200 \leq x \leq 500) & : 132.5 - 0.1 * x \\ \neg d \wedge (50 \leq x \leq 200) & : 102.5 + 0.05 * x \\ \neg d \wedge (0 \leq x \leq 50) & : 52.5 + 1.05 * x \end{cases} \tag{3}$$

This value function is piecewise and linear and the policy obtained from this value function shown in Figure 1 (for each slice of the state space) matches with Scarf's policy for the Inventory Control problem. If the holding and storage costs are linear, the optimal policy in each horizon is always of (S, s) (Arrow et al., 1958). In general this means if $(x > s)$ the policy should be not to order any items and if $(x < s)$ then ordering $S - s - x$ items is optimal. Figure 1 represents an extended algebraic decision diagram (XADD) representation which allows efficient implementation of the *case calculus* for arbitrary functions.

According to this we rewrite Scarf's policy:

$$\pi^{*,2}(x) = \begin{cases} (x < 0 \vee x > 500) & : -\infty \\ d \wedge (300 \leq x \leq 500) & : 0 \\ d \wedge (0 \leq x \leq 300) & : 300 - x \\ d \wedge (100 \leq x \leq 150) & : 200 - x \\ \neg d \wedge (200 \leq x \leq 500) & : 0 \\ \neg d \wedge (0 \leq x \leq 200) & : 200 - x \end{cases} \tag{4}$$

While this simple example illustrates the power of using continuous variables, for a multi-variate problem it is the very first solution to exactly solving problems such as the CAIC.

We propose novel ideas to work around some of the expressiveness limitations of previous approaches and significantly generalize the range of HMDPs that can be solved exactly. To achieve this more general solution, this paper contributes a number of important advances:

- The use of case calculus allows us to perform Symbolic dynamic programming (SDP) (Boutilier, Reiter, & Price, 2001) used to solve MDPs with piecewise transitions and reward functions defined in first-order logic. We define all required operations for SDP such as $\oplus, \ominus, max, min$ as well as new operations such as the continuous maximization of an action parameter y defined as max_y and integration of discrete noisy transition.

- We perform value iteration for two different settings. In the first setting of DA-HMDP we consider continuous state variables with a discrete action set while in the second setting CA-HMDP we consider continuous states and actions. Both DA-HMDPs and CA-HMDPs are evaluated on various problem domains. The results show that DA-HMDPs applies to a wide range of transition and reward functions providing hyper-rectangular value functions. CA-HMDPs have more restriction in modeling due to the increased complexity caused by continuous actions, and limit solutions to linear and quadratic transitions and rewards but provide strong results for many problems never solved exactly before.

- While the *case* representation for the optimal CAIC solution shown in (3) is sufficient in theory to represent the optimal value functions that our HMDP solution produces, this representation is unreasonable to maintain in practice since the number of case partitions may grow exponentially on each receding horizon control step. For *discrete* factored MDPs, algebraic decision diagrams (ADDs) (Bahar, Frohm, Gaona, Hachtel, Macii, Pardo, & Somenzi, 1993) have been successfully used in exact algorithms like SPUDD (Hoey, St-Aubin, Hu, & Boutilier, 1999) to maintain compact value representations. Motivated by this work we introduce extended ADDs (XADDs) to compactly represent general piecewise functions and show how to perform efficient operations on them *including* symbolic maximization. Also we present all properties and algorithms required for XADDs.

Aided by these algorithmic and data structure advances, we empirically demonstrate that our SDP approach with XADDs can exactly solve a variety of HMDPs with discrete and continuous actions.

2. Hybrid MDPs (HMDPs)

The mathematical framework of Markov Decision Processes (MDPs) is used for modelling many stochastic sequential decision making problems (Bellman, 1957). This discrete-time stochastic control process chooses an action a available at state s. The process then transitions to the next state s' according to $T(s, s')$ and receives a reward $R(s, a)$. The transition function follows the Markov property allowing each state to only depend on its previous state. We provide novel exact solutions using the MDP framework for discrete and continuous variables in the state and action space. Hybrid State and Action MDPs (HMDPs) are introduced in the next section followed by the finite-horizon solution via dynamic programming (Li & Littman, 2005).

2.1 Factored Representation

In an HMDP, states are represented by vectors of variables $(\vec{b}, \vec{x}) = (b_1, \ldots, b_n, x_1, \ldots, x_m)$. We assume that each $b_i \in \{0, 1\}$ $(1 \leq i \leq n)$ is boolean and each $x_j \in \mathbb{R}$ $(1 \leq j \leq m)$ is continuous. We also assume a finite set of p actions $A = \{a_1(\vec{y}_1), \ldots, a_p(\vec{y}_p)\}$, where each action $a_k(\vec{y}_k)$ $(1 \leq k \leq p)$ with parameter $\vec{y}_k \in \mathbb{R}^{|\vec{y}_k|}$ denotes continuous parameters for action a_k and if $|\vec{y}_k| = 0$ then action a_k has no parameters and is a discrete action.

Each HMDP model requires the following definitions:

(i) a state transition model $P(\vec{b}', \vec{x}' | \vec{b}, \vec{x}, a, \vec{y})$, which specifies the probability of the next state $(\vec{b}', \vec{x}')$ conditioned on a subset of the previous and next state and action a with its possible parameters $\vec{y}$;

(ii) a reward function $R(\vec{b}, \vec{x}, \vec{b}', \vec{x}', a, \vec{y})$, which specifies the immediate reward obtained by taking action $a(\vec{y})$ in state $(\vec{b}, \vec{x})$;

(iii) a discount factor γ, $0 \leq \gamma \leq 1$ to determine the weights of rewards in each time step. [1]

A policy π specifies the action $a(\vec{y}) = \pi(\vec{b}, \vec{x})$ to take in each state $(\vec{b}, \vec{x})$. Our goal is to find an optimal sequence of finite horizon-dependent policies [2] $\Pi^* = (\pi^{*,1}, \ldots, \pi^{*,H})$ that maximizes the expected sum of discounted rewards over a horizon $h \in H; H \geq 0$:

$$V^{\Pi^*}(\vec{x}) = E_{\Pi^*}\left[\sum_{h=0}^{H} \gamma^h \cdot r^h \middle| \vec{b}_0, \vec{x}_0\right]. \tag{5}$$

Here r^h is the reward obtained at horizon h following Π^* where we assume starting state $(\vec{b}_0, \vec{x}_0)$ at $h = 0$.

Such HMDPs are naturally factored (Boutilier, Dean, & Hanks, 1999) in terms of state variables $(\vec{b}, \vec{x}, \vec{y})$ where potentially $\vec{y} = 0$. The transition structure can be exploited in the form of a dynamic Bayes net (DBN) (Dean & Kanazawa, 1989) where the conditional probabilities $P(b_i' | \cdots)$ and $P(x_j' | \cdots)$ for each next state variable can condition on the

1. If time is explicitly included as one of the continuous state variables, $\gamma = 1$ is typically used, unless discounting by horizon (different from the state variable time) is still intended.
2. We assume a finite horizon H in this paper, however in cases where our SDP algorithm converges in finite time, the resulting value function and corresponding policy are optimal for $H = \infty$. For finitely bounded value with $\gamma = 1$, the forthcoming SDP algorithm may terminate in finite time, but is not guaranteed to do so; for $\gamma < 1$, an ϵ-optimal policy for arbitrary ϵ can be computed by SDP in finite time.

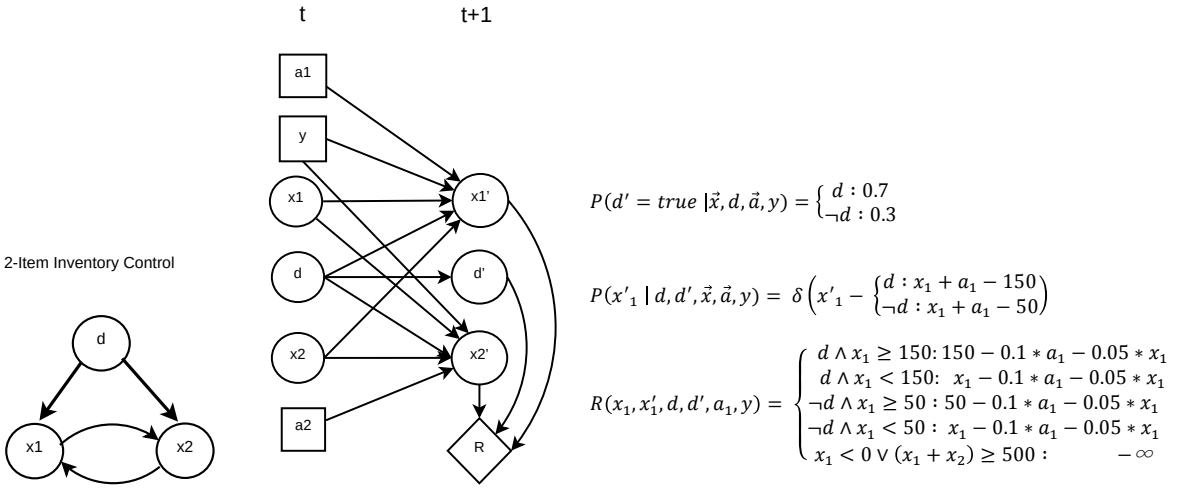

Figure 2: Network topology between state variables in the 2-item continuous action Inventory Control (CAIC) problem (Left); Dynamic bayes network (DBN) structure representing the transition and reward function (Middle); transition probabilities and reward function in terms of CPF and PLE for x_1 (Right).

action, current and next state. We can also have *synchronic arcs* (variables that condition on each other in the same time slice) within the binary $\vec{b}$ or continuous variables $\vec{x}$ and from $\vec{b}$ to $\vec{x}$.Hence we can factorize the joint transition model as

$$P(\vec{b'}, \vec{x'} | \vec{b}, \vec{x}, a, \vec{y}) = \prod_{i=1}^{n} P(b'_i | \vec{b}, \vec{x}, \vec{b'}, \vec{x'}, a, \vec{y}) \prod_{j=1}^{m} P(x'_j | \vec{b}, \vec{b'}, \vec{x}, \vec{x'}, a, \vec{y}).$$

where $P(b'_i | \vec{b}, \vec{x}, \vec{b'}, \vec{x'}, a, \vec{y})$ may condition on a subset of $\vec{b}$ and $\vec{x}$ in the current and next state and likewise $P(x'_j | \vec{b}, \vec{b'}, \vec{x}, \vec{x'}, a, \vec{y})$ may condition on a subset of $\vec{b}$, $\vec{b'}$, $\vec{x}$ and $\vec{x'}$. Figure 2 presents the DBN for our CAIC example according to this definition.

We call the conditional probabilities $P(b'_i | \vec{b}, \vec{x}, \vec{b'}, \vec{x'}, a, \vec{y})$ for *binary* variables b_i ($1 \leq i \leq n$) conditional probability functions (CPFs) — not tabular enumerations — because in general these functions can condition on both discrete and continuous state as in the right-hand side of (**??**). For the *continuous* variables x_j ($1 \leq j \leq m$), we represent the CPFs $P(x'_j | \vec{b}, \vec{b'}, \vec{x}, \vec{x'}, a, \vec{y})$ with *piecewise linear equations* (PLEs) satisfying three properties:

(i) PLEs can only condition on the action, current state, and previous state variables

(ii) PLEs are deterministic meaning that to be represented by probabilities they must be encoded using Dirac $\delta[\cdot]$ functions (example forthcoming)

(iii) PLEs are piecewise linear, where the piecewise conditions may be arbitrary logical combinations of $\vec{b}$, $\vec{b'}$ and linear inequalities over $\vec{x}$ and $\vec{x'}$.

The transition function example provided in the left-hand side of (**??**) can be expressed in PLE format such as the right figure in Figure 2. The use of the $\delta[\cdot]$ function ensures that the PLEs are conditional probability functions that integrates to 1 over x'_j; In more intuitive terms, one can see that this $\delta[\cdot]$ is a simple way to encode the PLE transition $x' = \{\ldots$ in the form of $P(x'_j | \vec{b}, \vec{b'}, \vec{x}, \vec{x'}, a, \vec{y})$.

While it will be clear that our restrictions do not permit general stochastic transition noise (e.g., Gaussian noise as in LQG control), they do permit discrete noise in the sense that $P(x'_j|\vec{b},\vec{b'},\vec{x},\vec{x'},a,\vec{y})$ may condition on $\vec{b'}$, which are stochastically sampled according to their CPFs. [3] We note that this representation effectively allows modeling of continuous variable transitions as a mixture of δ functions, which has been used frequently in previous exact continuous state MDP solutions (Feng et al., 2004; Meuleau, Benazera, Brafman, Hansen, & Mausam, 2009). Furthermore, we note that our representation is more general in DA-HMDPs than (Feng et al., 2004; Li & Littman, 2005; Meuleau et al., 2009) in that we do not restrict the equation to be linear, but rather allow it to specify *arbitrary* functions (e.g., nonlinear). The reward function can also be defined as *arbitrary* function of the current state for each action $a \in A$. While our DA-HMDP examples throughout the paper will demonstrate the full expressiveness of our symbolic dynamic programming approach, we note that there are computational advantages to be had when the reward and transition case conditions and functions can be restricted to linear polynomials.

Due to the same restrictions for CA-HMDPs the reward function $R(\vec{b},\vec{b'},\vec{x},\vec{x'},a,\vec{y})$ is defined as either of the following:

(i) a general piecewise linear function (boolean or linear conditions and linear values) such as (2)

(ii) a piecewise quadratic function of univariate state and a linear function of univariate action parameters:

$$R(x,x',d,d',a) = \begin{cases} \neg d \wedge x \geq -2 \wedge x \leq 2: & 4 - x^2 \\ d \vee x < -2 \vee x > 2: & 0 \end{cases} \tag{6}$$

These transition and reward constraints will ensure that all derived functions in the solution of HMDPs adhere to the reward constraints.

2.2 Solution methods

Now we provide a continuous state generalization of *value iteration* (Bellman, 1957), which is a dynamic programming algorithm for constructing optimal policies. It proceeds by constructing a series of h-stage-to-go value functions $V^h(\vec{b},\vec{x})$. Initializing $V^0(\vec{b},\vec{x}) = 0$ we define the quality $Q_a^h(\vec{b},\vec{x},\vec{y})$ of taking action $a(\vec{y})$ in state $(\vec{b},\vec{x})$ and acting so as to obtain $V^{h-1}(\vec{b},\vec{x})$ thereafter as the following:

$$Q_a^h(\vec{b},\vec{x},\vec{y}) = \sum_{\vec{b'}} \int \left(\prod_{i=1}^{n} P(b'_i|\vec{b},\vec{x},\vec{b'},\vec{x'},a,\vec{y}) \prod_{j=1}^{m} P(x'_j|\vec{b},\vec{b'},\vec{x},\vec{x'},a,\vec{y}) \right) \tag{7}$$
$$\left[R(\vec{b},\vec{b'},\vec{x},\vec{x'},a,\vec{y}) + \gamma \cdot V^{h-1}(\vec{b'},\vec{x'}) d\vec{x'} \right]$$

3. Continuous stochastic noise for the transition function is an on going work which allows us to model stochasticity more generally

Given $Q_a^h(\vec{b}, \vec{x}, \vec{y})$ for each $a \in A$ where $\vec{y}$ can also be empty , we can proceed to define the h-stage-to-go value function as follows:

$$V^h(\vec{b}, \vec{x}) = \max_{a \in A} \max_{\vec{y} \in \mathbb{R}^{|\vec{y}|}} \left\{ Q_a^h(\vec{b}, \vec{x}, \vec{y}) \right\} \tag{8}$$

For discrete actions, maximization over the continuous parameter $\vec{y}$ is omitted. The $max_{\vec{y}}$ operator will be defined in the next section and is required to generalize solutions from DA-HMDPs to CA-HMDPs. If the horizon H is finite, then the optimal value function is obtained by computing $V^H(\vec{b}, \vec{x})$ and the optimal horizon-dependent policy $\pi^{*,h}$ at each stage h can be easily determined via $\pi^{*,h}(\vec{b}, \vec{x}) = \arg\max_a \arg\max_{\vec{y}} Q_a^h(\vec{b}, \vec{x}, \vec{y})$. If the horizon $H = \infty$ and the optimal policy has finitely bounded value, then value iteration can terminate at horizon h if $V^h = V^{h-1}$; then $V^\infty = V^h$ and $\pi^{*,\infty} = \pi^{*,h}$.

In DA-HMDPs, we can always compute the value function in tabular form; however, how to compute this for HMDPs with reward and transition function as previously defined is the objective of the symbolic dynamic programming algorithm that we define in the next section. From this *mathematical* definition, we show how to *compute* (7) and (8) for the previously defined HMDPs.

3. Symbolic Dynamic Programming

As it's name suggests, symbolic dynamic programming (SDP) (Boutilier et al., 2001) is simply the process of performing dynamic programming (in this case value iteration) via symbolic manipulation. While SDP as defined in (Boutilier et al., 2001) was previously only used with piecewise constant functions, we now generalize the representation to work with general piecewise functions for HMDPs in this article.

Before we define our solution, however, we must formally define our case representation and symbolic case operators.

3.1 Case Representation and Operations

Throughout this article, we will assume that all symbolic functions can be represented in *case* form as follows:

$$f = \begin{cases} \phi_1 : & f_1 \\ \vdots & \vdots \\ \phi_k : & f_k \end{cases}$$

Here the ϕ_i are logical formulae defined over the state $(\vec{b}, \vec{x})$ that can include arbitrary logical $(\wedge, \vee, \neg)$ combinations of (a) boolean variables in $\vec{b}$ and (b) inequalities $(\geq, >, \leq, <)$, equalities $(=)$, or disequalities $(\neq)$ where the left and right operands can be *any* function of one or more variables in $\vec{x}$. Each ϕ_i will be disjoint from the other ϕ_j $(j \neq i)$; however the ϕ_i may not exhaustively cover the state space, hence f may only be a *partial function* and may be undefined for some state assignments. In general we require f to be continuous (including no discontinuities at partition boundaries); operations preserve this property. The main operations required to perform SDP are provided below in the case calculus:

Scalar multiplication and Negation

Unary operations such as scalar multiplication $c \cdot f$ (for some constant $c \in \mathbb{R}$) or negation $-f$ on case statements f are presented below; the unary operation is simply applied to each f_i $(1 \leq i \leq k)$.

$$c \cdot f = \begin{cases} \phi_1 : & c \cdot f_1 \\ \vdots & \vdots \\ \phi_k : & c \cdot f_k \end{cases} \qquad -f = \begin{cases} \neg\phi_1 : & f_1 \\ \vdots & \vdots \\ \neg\phi_k : & f_k \end{cases}$$

Binary operations

Intuitively, to perform a *binary operation* on two case statements, we simply take the cross-product of the logical partitions of each case statement and perform the corresponding operation on the resulting paired partitions. Letting each ϕ_i and ψ_j denote generic first-order formulae, we can perform the "cross-sum" $\oplus$ and "cross-product" $\otimes$ of two (unnamed) cases in the following manner:

$$\begin{cases} \phi_1 : & f_1 \\ \phi_2 : & f_2 \end{cases} \oplus \begin{cases} \psi_1 : & g_1 \\ \psi_2 : & g_2 \end{cases} = \begin{cases} \phi_1 \wedge \psi_1 : & f_1 + g_1 \\ \phi_1 \wedge \psi_2 : & f_1 + g_2 \\ \phi_2 \wedge \psi_1 : & f_2 + g_1 \\ \phi_2 \wedge \psi_2 : & f_2 + g_2 \end{cases} \quad \begin{cases} \phi_1 : & f_1 \\ \phi_2 : & f_2 \end{cases} \otimes \begin{cases} \psi_1 : & g_1 \\ \psi_2 : & g_2 \end{cases} = \begin{cases} \phi_1 \wedge \psi_1 : & f_1 * g_1 \\ \phi_1 \wedge \psi_2 : & f_1 * g_2 \\ \phi_2 \wedge \psi_1 : & f_2 * g_1 \\ \phi_2 \wedge \psi_2 : & f_2 * g_2 \end{cases}$$

Likewise, we can perform $\ominus$ by subtracting partition values to obtain the result. Some partitions resulting from the application of the $\oplus$, $\ominus$, and $\otimes$ operators may be inconsistent (infeasible); we may simply discard such partitions as they are irrelevant to the function value.

Symbolic maximization

For SDP, we'll also need to perform which is fairly straightforward to define:

$$\text{casemax}\left(\begin{cases} \phi_1 : & f_1 \\ \phi_2 : & f_2 \end{cases}, \begin{cases} \psi_1 : & g_1 \\ \psi_2 : & g_2 \end{cases} \right) = \begin{cases} \phi_1 \wedge \psi_1 \wedge f_1 > g_1 : & f_1 \\ \phi_1 \wedge \psi_1 \wedge f_1 \leq g_1 : & g_1 \\ \phi_1 \wedge \psi_2 \wedge f_1 > g_2 : & f_1 \\ \phi_1 \wedge \psi_2 \wedge f_1 \leq g_2 : & g_2 \\ \phi_2 \wedge \psi_1 \wedge f_2 > g_1 : & f_2 \\ \phi_2 \wedge \psi_1 \wedge f_2 \leq g_1 : & g_1 \\ \phi_2 \wedge \psi_2 \wedge f_2 > g_2 : & f_2 \\ \phi_2 \wedge \psi_2 \wedge f_2 \leq g_2 : & g_2 \end{cases}$$

One can verify that the resulting case statement is still within the case language defined previously. At first glance this may seem like a cheat and little is gained by this symbolic sleight of hand. However, simply having a case partition representation that is closed

under maximization will facilitate the closed-form regression step that we need for SDP. Furthermore, the XADD that we introduce later will be able to exploit the internal decision structure of this maximization to represent it much more compactly.

Restriction

In the next operation of *restriction* we want to restrict a function f to apply only in cases that satisfy some formula ϕ, which we write as $f|_\phi$. This can be done by simply appending ϕ to each case partition as follows:

$$f = \begin{cases} \phi_1 : & f_1 \\ \vdots & \vdots \\ \phi_k : & f_k \end{cases} \qquad f|_\phi = \begin{cases} \phi_1 \wedge \phi : & f_1 \\ \vdots & \vdots \\ \phi_k \wedge \phi : & f_k \end{cases}$$

Clearly $f|_\phi$ only applies when ϕ holds and is undefined otherwise, hence $f|_\phi$ is a partial function unless $\phi \equiv \top$.

Substitution

Symbolic substitution simply takes a set σ of variables and their substitutions, e.g., $\sigma = \{x_1'/(x_1+x_2), x_2'/x_1^2\exp(x_2)\}$ where the LHS of the / represents the substitution variable and the RHS of the / represents the expression that should be substituted in its place. No variable occurring in any RHS expression of σ can also occur in any LHS expression of σ. We write the substitution of a non-case function f_i with σ as $f_i\sigma$; as an example, for the σ defined previously and $f_i = x_1' + x_2'$ then $f_i\sigma = x_1 + x_2 + x_1^2\exp(x_2)$ as would be expected. We can also substitute into case partitions ϕ_j by applying σ to each inequality operand; as an example, if $\phi_j \equiv x_1' \leq \exp(x_2')$ then $\phi_j\sigma \equiv x_1 + x_2 \leq \exp(x_1^2\exp(x_2))$. Having now defined substitution of σ for non-case functions f_i and case partitions ϕ_j we can define it for case statements in general:

$$f = \begin{cases} \phi_1 : & f_1 \\ \vdots & \vdots \\ \phi_k : & f_k \end{cases} \qquad f\sigma = \begin{cases} \phi_1\sigma : & f_1\sigma \\ \vdots & \vdots \\ \phi_k\sigma : & f_k\sigma \end{cases}$$

One property of substitution is that if f has mutually exclusive partitions ϕ_i $(1 \leq i \leq k)$ then $f\sigma$ must also have mutually exclusive partitions — this follows from the logical consequence that if $\phi_1 \wedge \phi_2 \models \bot$ then $\phi_1\sigma \wedge \phi_2\sigma \models \bot$.

Continuous Integration of the δ-function

Continuous Integration evaluates the integral marginalization $\int_{\vec{x}}$ over the continuous variables in a function f. One of the *key novel insights of SDP* in the context of HMDPs is that the integration $\int_{x_j'} \delta[x_j' - g(\vec{x})] f dx_j'$ simply *triggers the substitution* $\sigma = \{x_j'/g(\vec{x})\}$ on f, that is

$$\int_x \delta[x - g(\vec{x})] f dx' \;=\; f\{x/g(\vec{x})\}. \tag{9}$$

To perform 9 on a more general representation, we obtain:

$$= \begin{cases} \phi_1: & f\{x = g_1\} \\ \vdots & \vdots \\ \phi_k: & f\{x = g_k\} \end{cases}$$

Here we note that because f is *already* a case statement, we can simply replace the single partition ϕ_i with the multiple partitions of $f\{x/g_i\}|_{\phi_i}$.This reduces the *nested* case statement back down to a non-nested case statement as in the following example:

$$\begin{cases} \phi_1: & \begin{cases} \psi_1: & f_{11} \\ \psi_2: & f_{12} \end{cases} \\ \phi_2: & \begin{cases} \psi_1: & f_{21} \\ \psi_2: & f_{22} \end{cases} \end{cases} = \begin{cases} \phi_1 \wedge \psi_1: & f_{11} \\ \phi_1 \wedge \psi_2: & f_{12} \\ \phi_2 \wedge \psi_1: & f_{21} \\ \phi_2 \wedge \psi_2: & f_{22} \end{cases}$$

Continuous Maximization

Continuous Maximization of a variable y is defined as $g(\vec{b}, \vec{x}) := \max_{\vec{y}} f(\vec{b}, \vec{x}, \vec{y})$ where we crucially note that *the* maximizing $\vec{y}$ is a function $g(\vec{b}, \vec{x})$, hence requiring *symbolic* constrained optimization. We can rewrite $f(\vec{b}, \vec{x}, y)$ via the following equalities: [4]

$$\begin{aligned} \max_y f(\vec{b}, \vec{x}, y) &= \max_y \text{casemax}_i \, \phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y) \\ &= \text{casemax}_i \boxed{\max_y \phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y)} \end{aligned} \tag{10}$$

Because the ϕ_i are mutually disjoint and exhaustive, $f(\vec{b}, \vec{x}, y) = \text{casemax}_i \, \phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y)$. Then because $\max_y$ and casemax_i are commutative and may be reordered, we can compute $\max_y$ for *each case partition individually*. Thus to complete this section we need only show how to symbolically compute a single partition $\max_y \phi_i(\vec{b}, \vec{x}, y) : f_i(\vec{b}, \vec{x}, y)$.

In ϕ_i, we observe that each conjoined constraint serves one of three purposes:

(i) *upper bound on* y: it can be written as $y < \cdots$ or $y \leq \cdots$.

(ii) *lower bound on* y: it can be written as $y > \cdots$ or $y \geq \cdots$. [5]

(iii) *independent of* y: the constraints do not contain y and can be safely factored outside of the $\max_y$.

Because there are multiple symbolic upper and lower bounds on y, in general we will need to apply the casemax (casemin) operator to determine the highest lower bound LB (lowest upper bound UB).

We also know that $\max_y \phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y)$ for a continuous function f_i must occur at the critical points of the function — either the upper or lower bounds (UB and LB) of y, or the *Root* (i.e., zero) of $\frac{\partial}{\partial y} f_i$ w.r.t. y. Each of UB, LB, and *Root* is a symbolic function of $\vec{b}$ and $\vec{x}$.

4. The second line ensures that all illegal values are mapped to $-\infty$

5. For purposes of evaluating a case function f at an upper or lower bound, it does not matter whether a bound is inclusive ($\leq$ or $\geq$) or exclusive ($<$ or $>$) since f is required to be continuous and hence evaluating at the limit of the inclusive bound will match the evaluation for the exclusive bound.

Given the *potential* maxima points of $y = UB$, $y = LB$, and $y = Root$ of $\frac{\partial}{\partial y} f_i(\vec{b}, \vec{x}, y)$ w.r.t. constraints $\phi_i(\vec{b}, \vec{x}, y)$ — which are all symbolic functions — we must symbolically evaluate which yields the maximizing value Max for this case partition:

$$Max = \begin{cases} \exists Root: & \text{casemax}(f_i\{y/Root\}, f_i\{y/UB\}, f_i\{y/LB\}) \\ \text{else:} & \text{casemax}(f_i\{y/UB\}, f_i\{y/LB\}) \end{cases}$$

Here $\text{casemax}(f, g, h) = \text{casemax}(f, \text{casemax}(g, h))$. The substitution operator $\{y/f\}$ replaces y with case statement f, defined previously.

At this point, we have almost completed the computation of the $\max_y \phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y)$ except for one issue: the incorporation of the independent (Ind) constraints (factored out previously) and additional constraints that arise from the symbolic nature of the UB, LB, and $Root$.

Specifically for the latter, we need to ensure that indeed $LB \leq Root \leq UB$ (or if no root exists, then $LB \leq UB$) by building a set of constraints $Cons$ that ensure these conditions hold; to do this, it suffices to ensure that for each possible expression e used to construct LB that $e \leq Root$ and similarly for the $Root$ and UB. Now we express the final result as a single case partition:

$$\max_y \phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y) \quad = \quad \{Cons \wedge Ind : Max$$

Hence, to complete the maximization for an entire case statement f, we need only apply the above procedure to each case partition of f and then perform a symbolic casemax all of these results.

3.2 Symbolic Dynamic Programming (SDP)

In this section we present the Symbolic value iteration algorithm (SVI) for HMDPs.

Our objective is to take a DA-HMDP or CA-HMDP as defined in Section 2.1, apply value iteration as defined in Section 2.2, and produce the final value optimal function V^h at horizon h in the form of a case statement. Algorithm 1 presents this briefly.

For the base case of $h = 0$ in line 2, we note that setting $V^0(\vec{b}, \vec{x}) = 0$ (or to the reward case statement, if not action dependent) is trivially in the form of a case statement.

Next, for $h > 0$ and for each action in line 5 we must perform lines 6–12. Starting with the application of Algorithm 2. Note that we have omitted parameters $\vec{b}$ and $\vec{x}$ from V and Q to avoid notational clutter. Fortunately, given our previously defined operations, SDP is straightforward and can be divided into five steps:

1. *Prime the Value Function*: Since V^h will become the "next state" in value iteration, we setup a substitution $\sigma = \{b_1/b'_1, \ldots, b_n/b'_n, x_1/x'_1, \ldots, x_m/x'_m\}$ and obtain $V'^h = V^h\sigma$ in line 2 of Algorithm 2.

2. *Add Reward Function*: If the reward function R contains any primed state variable b' or x', lines 3–4 is executed to add this reward function to the previous discounted Q-value. If R had no primed variables, then it is added to the Q-value at the end of this algorithm in lines 14–15.

Algorithm 1: VI(HMDP, H) $\longrightarrow (V^h, \pi^{*,h})$

begin
  $V^0 := 0, h := 0$
  while $h < H$ **do**
    $h := h + 1$
    foreach $a(\vec{y}) \in A$ **do**
      $Q_a^h(\vec{y})$:= Regress$(V^{h-1}, a, \vec{y})$
      //Continuous action parameter
      if $|\vec{y}| > 0$ **then**
        $Q_a^h(\vec{y}) := \max_{\vec{y}} Q_a^h(\vec{y})$
        $\pi^{*,h} := \arg\max_a Q_a^h(\vec{y})$
      else
        $\pi^{*,h} := \arg\max_a Q_a^h(\vec{y})$
    $V^h := \text{casemax}\, Q_a^h(\vec{y})$
    if $V^h = V^{h-1}$ **then**
      break *// Terminate if early convergence*
  return $(V^h, \pi^{*,h})$
end

Algorithm 2: Regress$(V, a, \vec{y}) \longrightarrow Q$

begin
  $Q =$ Prime(V) *// All $b_i \to b_i'$ and all $x_i \to x_i'$*
  if v' *in* R **then**
    $Q := R(\vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y}) \oplus (\gamma \cdot Q)$
  foreach v' *in* Q **do**
    if $v' = x_j'$ **then**
      //Continuous marginal integration
      $Q := \int Q \otimes P(x_j' | \vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y})\, d_{x_j'}$
    if $v' = b_i'$ **then**
      // Discrete marginal summation
      $Q := \left[Q \otimes P(b_i' | \vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y}) \right] |_{b_i'=1} \oplus \left[Q \otimes P(b_i' | \vec{b}, \vec{b}, \vec{x}, \vec{x}', a, \vec{y}) \right] |_{b_i'=0}$
  if $\neg$ *(v' in R)* **then**
    $Q := R(\vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y}) \oplus (\gamma \cdot Q)$
  return Q
end

3. *Continuous Integration*: As defined in line 7–9 once we have our primed value function V'^h in case statement format defined over next state variables $(\vec{b}', \vec{x}')$, we evaluate the integral marginalization $\int_{\vec{x}'}$ over the continuous variables in (7). Because the lower and upper integration bounds are respectively $-\infty$ and ∞ and we have disallowed synchronic arcs between variables in $\vec{x}'$ in the transition DBN, we can marginalize out each x'_j independently, and in any order. According to 9 we have the following:

$$\int_{x'_j} \delta[x'_j - g(\vec{x})] V'^h dx'_j \;=\; V'^h\{x'_j/g(\vec{x})\}$$

This operation is performed repeatedly in sequence *for each* x'_j $(1 \le j \le m)$ for every action a. The only additional complication is that the form of $P(x'_j|\vec{b}, \vec{x}, \vec{b'}, \vec{x'}, a, \vec{y})$ is a *conditional* equation such as the right-hand of Figure 2, and represented generically as follows:

$$P(x'_j|\vec{b}, \vec{x}, \vec{b'}, \vec{x'}, a, \vec{y}) = \delta \left[x'_j = \begin{cases} \phi_1 : & f_1 \\ \vdots & \vdots \\ \phi_k : & f_k \end{cases} \right] \tag{11}$$

In effect, we can read (11) as a *conditional substitution*, i.e., in each of the different *previous state* conditions ϕ_i $(1 \le i \le k)$, we obtain a different substitution for x'_j appearing in V'^h (i.e., $\sigma = \{x'_j/f_i\}$).

To perform the full continuous integration, if we initialize $\tilde{Q}_a^{h+1} := V'^h$ for each action $a \in A$, and repeat the above integrals for all x'_j, updating $\tilde{Q}_a^{h+1}$ each time, then after elimination of all x'_j $(1 \le j \le m)$, we will have the partial regression of V'^h for the continuous variables for each action a denoted by $\tilde{Q}_a^{h+1}$.

4. *Discrete Marginalization*: Now that we have our partial regression $\tilde{Q}_a^{h+1}$ for each action a, we proceed to derive the full backup Q_a^{h+1} from $\tilde{Q}_a^{h+1}$ by evaluating the discrete marginalization $\sum_{\vec{b}'}$ in (7) which is shown in lines 10–12. Because we previously disallowed synchronic arcs between the variables in $\vec{b}'$ in the transition DBN, we can sum out each variable b'_i $(1 \le i \le n)$ independently. Hence, initializing $Q_a^{h+1} := \tilde{Q}_a^{h+1}$ we perform the discrete regression by applying the following iterative process *for each* b'_i in any order for each action a:

$$Q_a^{h+1} := \left[Q_a^{h+1} \otimes P(b'_i|\vec{b}, \vec{x}, \vec{b'}, \vec{x'}, a, \vec{y}) \right] |_{b'_i=1} \oplus \left[Q_a^{h+1} \otimes P(b'_i|\vec{b}, \vec{x}, \vec{b'}, \vec{x'}, a, \vec{y}) \right] |_{b'_i=0}. \tag{12}$$

This requires a variant of the earlier restriction operator $|_v$ that actually *sets* the variable v to the given value if present. Note that both Q_a^{h+1} and $P(b'_i|\vec{b}, \vec{x}, \vec{b'}, \vec{x'}, a, \vec{y})$ can be represented as case statements (discrete CPTs *are* case statements), and each operation produces a case statement. Thus, once this process is complete, we have marginalized over all $\vec{b}'$ and Q_a^{h+1} is the symbolic representation of the intended Q-function.

Algorithm 3: `Continuous Maximization`$(y,\ f(\vec{b},\vec{x},y)) \longrightarrow (max_y f(\vec{b},\vec{x},y))$

```
1  begin
2      LB = UB = IND = Cons = ∅ , Case_max = ∅
3      for φ_i ∈ f ( for all partitions of f perform maximization) do
4          for c_i ∈ φ_i (For all conditions c of φ_i) do
5              if c_i ≤ y then
6                  LB := [LB, c_i] //Add constraint c_i to lower bound set
7              if c_i ≥ y then
8                  UB := [UB, c_i] //Add constraint c_i to upper bound set
9              else
10                 IND := [IND, c_i] //Add constraint c_i to independent constraint set
11
12
13         LB = casemax(LB_i, LB_{i+1}) //Take maximum of all lower bounds
14         UB = casemin(UB_i, UB_{i+1}) //Take minimum of all upper bounds
15         Root := (∂/∂y f_i = 0)
16         if (Root ≠ null) then
17             Cons = (𝕀[LB] ≤ 𝕀[Root]) ∧ (𝕀[Root] ≤ 𝕀[UB])
18         else
19             Cons = (𝕀[LB] ≤ 𝕀[UB])
20         //Conditions and value of continuous max for this partition
21         Max = IND ∧ Cons : casemax(f_i{y/LB}, f_i{y/UB}, f_i{y/Root})
22         //Take maximum of this partition and all other partitions
23         Case_max = max(Case_max, Max)
24
25     return Case_max
26 end
```

5. *Continuous action Maximization*: This maximization is over an action variable $a(\vec{y})$ in line 8–9 of Algorithm 1 where $|\ \vec{y}\ | > 0$, requires a continuous maximization. Here we take the maximum over parameter y of action variable $a(\vec{y})$. If the action is discrete $|\ \vec{y}\ | = 0$, lines 8–10 are not performed. Exploiting the commutativity of max, we can first rewrite any multivariate $\max_{\vec{y}}$ as a sequence of univariate max operations $\max_{y_1} \cdots \max_{y_{|\vec{y}|}}$; hence it suffices to provide just the *univariate* $\max_y$ solution:

$$\max_{\vec{y}} = \max_{y_1} \cdots \max_{y_{|\vec{y}|}} \Rightarrow g(\vec{b},\vec{x}) := \max_{y}\ f(\vec{b},\vec{x},y).$$

According to the properties on the Continuous Maximization operation defined in the previous section, we compute a univariate maximization $\max_y \phi_i(\vec{b},\vec{x},y) f_i(\vec{b},\vec{x},y)$ using Algorithm 3. [6]

6. Note also that from here out we assume that all case partition conditions ϕ_i of f consist of conjunctions of non-negated linear inequalities and possibly negated boolean variables — conditions easy to enforce

For each of the constraints c_i in each partition ϕ_i the lower and upper bounds and independent constraints are determined in 4–10. A unique LB and UB and the root of the function are computed in lines 13–15 by taking the maximum of the lower bounds and the minimum of the upper bounds as the best bounds in the current partition and line 15 takes any roots of the leaf function. The boundary constraints in lines 16–19 are added to the independent constraints as the constraint of the final maximum *Max* and a casemax is performed on the LB,UB and the roots as the function of *Max*. Taking this $\max_y$ is performed in line 21 for each partition. Returning to (10), we find that we have now specified the inner operation (shown in the $\Box$). Hence, to complete the maximization for an entire case statement f, we need only apply the above procedure to each case partition of f and then casemax all of these results in line 23. To obtain the policy in Figure 1, we need only annotate leaf values with any *UB*, *LB*, and *Root* substitutions. Continuous maximization is further explained in the next section using the appropriate data structure.

6. *Maximization*: Now that we have $Q_a^{h+1}(\vec{y})$ in case format for each action $a \in \{a_1(\vec{y}_1), \ldots, a_p(\vec{y}_p)\}$, obtaining V^{h+1} in case format as defined in (8) requires sequentially applying *symbolic maximization* as defined previously:

$$V^{h+1} = \max(Q_{a_1}^{h+1}(\vec{y}), \max(\ldots, \max(Q_{a_{p-1}}(\vec{y})^{h+1}, Q_{a_p}^{h+1}(\vec{y}))))$$

Line 10 and 12 in Algorithm 1 computes the optimal policy on the Q-function for the two action cases and line 14 performs symbolic maximization. By induction, because V^0 is a case statement and applying SDP to V^h in case statement form produces V^{h+1} in case statement form, we have achieved our intended objective with SDP. On the issue of correctness, we note that each operation above simply implements one of the dynamic programming operations in (7) or (8), so correctness simply follows from verifying (a) that each case operation produces the correct result and that (b) each case operation is applied in the correct sequence as defined in (7) or (8).

On a final note, we observe that SDP holds for *any* symbolic case statements; we have not restricted ourselves to rectangular piecewise functions, piecewise linear functions, or even piecewise polynomial functions. As the SDP solution is purely symbolic, SDP applies to *any* HMDP using bounded symbolic function that can be written in case format! Of course, that is the theory, next we meet practice.

4. Extended Algebric Decision Diagrams (XADDs)

In the previous section all operations required to perform SDP algorithms were covered. The case statements represent arbitrary piecewise functions allowing general solutions to continuous problems. In practice, it can be prohibitively expensive to maintain a case statement representation of a value function with explicit partitions. Motivated by the SPUDD (Hoey et al., 1999) algorithm which maintains compact value function representations for finite discrete factored MDPs using algebraic decision diagrams (ADDs) (Bahar et al., 1993), we extend this formalism to handle continuous variables in a data structure we refer to as the

since negation inverts inequalities, e.g., $\neg[x < 2] \equiv [x \geq 2]$ and disjunctions can be split across multiple non-disjunctive, disjoint case partitions.

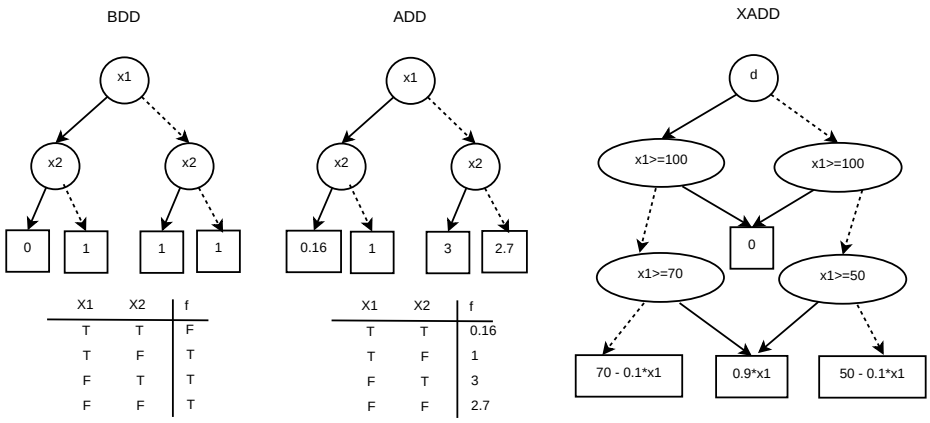

Figure 3: Comparison of the three decision diagrams: Binary decision diagrams (BDDs) with boolean leaves and decisions (Left) representing $f(x_1, x_2) = x_1\bar{x_2} + \bar{x_1}$ as shown in the truth table; Algebraic decision diagrams (ADDs) with boolean decision nodes and real values at the leaves (Middle) represented by the truth table; Extended algebraic decision diagrams (XADDs) with polynomial leaves and decision nodes (Right) demonstrating the reward function (2) for a 12-item stochastic CAIC problem.

XADD. Here we introduce this compact data structure of XADDs which can implement case statements efficiently. Figure 1 of the introduction section demonstrates the value function for the Inventory Control problem as an XADD representation. While XADDs are extended from ADDs, ADDs are extended from Binary decision diagrams (BDDs), allowing first-order logic instead of boolean logic. Figure 3 demonstrates examples of the three decision diagrams of BDD, ADD and XADD as a comparison to show their expressiveness.

A *binary decision diagram* (BDD) (Bryant, 1986) can represent propositional formulas or boolean functions $\{0,1\}^n \to \{0,1\}$ as an ordered *directed acyclic graph* (DAG) where each node represents a random variable and edges represent direct correlations between the variables. Each decision node is a boolean test variable with two successor nodes of false/true. The edge from the decision node to a false (true) child represents assigning 0 (1) in boolean logic. To evaluate the boolean function which is represented by a certain BDD, each of the variables are assigned a false/true value by following the corresponding branches until reaching a leaf. The boolean value at the leaf is the value returned by this function according to the given variable assignment.

Extending BDDs to *algebric decision diagrams* (ADDs) allows a real-value range in the function representation $\{0,1\}^n \to \mathbb{R}$. ADDs further provide an efficient representation of *context-specific independent* (Boutilier, Friedman, Goldszmidt, & Koller, 1996) functions (CSI) where node X is independent of nodes W and V given the context u where node $(U = true)$. Arithmetic operations can be performed on these functions returning a function value at the leaves; examples include addition ($\oplus$), subtraction ($\ominus$), multiplication ($\otimes$), division ($\oslash$), $\min(\cdot,\cdot)$ and $\max(\cdot,\cdot)$ (Bahar et al., 1993).

Parameterized ADDs (PADDs) are an extension of ADDs that allow for a compact representation of functions from $\{0,1\}^n \to \mathbb{E}$, where $\mathbb{E}$ is the space of expressions parameterized by $\vec{p}$. Formal definitions of our XADD are similar to that of PADD (Delgado, Sanner, & de Barros, 2010).

Extended ADDs (XADDs) allow representing continuous variables in a decision diagram in the function representation of $\mathbb{R}^{n+m} \to \mathbb{R}$ over case statements. Each leaf in an XADD represents a multi-variate arbitrary function from the real-value domain and each decision node can be an equality, dis-equality or inequality on the multi-variate domain which is more expressive than the ADD boolean decisions. The branches are true/false depending on the value of each decision node. This compact representation will not require truth tables like ADDs or BDDs as it is more expressive in allowing infinitely many real values for each decision.

We next formally define the XADD operations and algorithms required to support all case operations of SDP as well as pruning algorithms to make this representation even more efficient.

4.1 Formal Definition and Operations

An XADD allows polynomials at the leaves and decisions instead of a single real-value. According to the set of continuous variables in an XADD $\theta = \{x_1, x_2, ..., x_n\}$ and the set of constants $c_i (0 \leq i \leq p)$ each leaf can be canonically defined as:

$$c_0 + \sum_i c_i \prod_j \theta ij$$

where $1 \leq j \leq n$. Each decision node is an inequality of some polynomial function. Formally an XADD is defined using a BNF grammer:

$$\begin{array}{lll} F & ::= & Poly \mid \text{if}(F^{var}) \text{ then } F_h \text{ else } F_l \\ F^{var} & ::= & (Poly \leq 0) \mid (Poly \geq 0) \mid B \\ Poly & ::= & c_0 + \sum_i c_i \prod_j \theta_{ij} \\ B & ::= & 0 | 1 \end{array}$$

An XADD node F can either be a leaf $Poly$ with a polynomial value or a decision node F^{var} with two branches F_h and F_l which are both of the non-terminal type F. The decision node F^{var} associated with a single variable var can be a polynomial inequality or a boolean decision $B = \{b_1, b_2, ..., b_m\}$ where each boolean variable $b_k \in \{0, 1\}$. If F_h is taken the value of the decision node F^{var} is true and if F_l is taken the negation of the decision node $\neg F^{var}$ is set to true.[7]

The value returned by a function f represented as an XADD (F) containing (a subset of) the discrete and continuous variables $\{b_1, \cdots, b_m, x_1, \cdots, x_n\}$ with variable assignments $\rho \in \{\{true, false\}^m, \mathbb{R}^n\}$ can be defined recursively by:

$$Val(F, \rho) = \begin{cases} \text{if } F = Poly: & Poly \\ \text{if } F = Poly \rho(F^{var}) = true: & Val(F_h, \rho) \\ \text{if } F = Poly \rho(F^{var}) = false: & Val(F_l, \rho) \end{cases}$$

This recursive definition of $Val(F, \rho)$ reflects the structural evaluation of F by starting at its root node and following the branch at each decision node corresponding to the decisions

7. Note we assume continuous functions; if a function has the same values on a boundary point (equality), we allow only one of the $\leq, \geq$ at the boundary point. This continuous property allows us to replace $<$ ($>$) with $\leq (\geq)$.

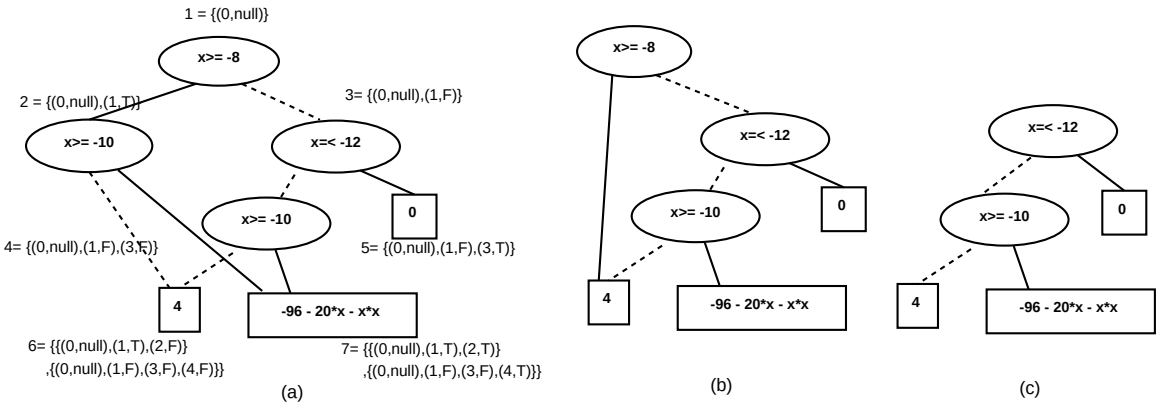

Figure 4: (a) Path and formula definitions on each node in the XADD. A path is defined as a sequence of the tuple (F^{var}, dec) and a formula is a set of paths. (b) Pruning inconsistent nodes of (a). (c) Pruning redundant nodes of (b)

taken in F^{var} — continuing until a leaf node is reached, which is then returned as $Val(F, \rho)$. The diagram on the right of Figure 3 demonstrates the polynomial leaves and the decision node inequalities which branch to true/false depending on the decision value.

To define the pruning algorithms, we provide some definitions listed below:

Definition (**Function representation**): A multi-variate function of booleans and real values $f : \mathbb{B}^m \times \mathbb{R}^n \to \mathbb{R}$ denoted by $f(\vec{b}, \vec{x})$ represents an XADD (f_{XADD}) defined on the class of piecewise formulas (case statements).

Definition (**Path**): A path p in f_{XADD} is a sequence of the pair $\rho = (F^{var}, dec)$ where each node F^{var} has a unique id and each decision assignment $dec \in \{true, false\}$ represents the (true or false) branch that node F^{var} has followed. Note that the root node $\rho_0 = (0, null)$ has a null decision assignment. A path is generally defined as a finite subset (in a sequence) of all possible pairs in f_{XADD}.

$$p_j \subset \{(F_1^{var}, dec_1), (F_2^{var}, dec_2), \cdots, (F_{n+m}^{var}, dec_{n+m})\}$$

where $1 \leq j \leq (n + m)$ is the path number and the last pair on a given path p_k is defined as the *end-node* : $\eta(p_k)$ of that path.

Definition (**Formula**): The set of all paths for a node F^{var} is defined as all paths $\{p_1 \cdots p_k\}$ $(1 \leq k \leq (n + m))$ such that the end-node of these paths are equal to node F^{var} that is: $\eta(p_k) = F^{var}$. A formula on this node ψ_{var} is defined as this finite set of paths $\psi_{var} = \{p_k | \eta(p_k) = F^{var}, 1 \leq k \leq (n + m)\}$.

To define any node F^{var} in f_{XADD} logically we have the following:

$$\psi_{var} = \bigvee_{p_j \in p_{var}} \Big(\bigwedge_{\rho_i \in p_j} \rho_i | \eta(p_j) = F^{var} \Big) \quad , \rho_i = \begin{cases} dec_i = true: & F_i^{var} \\ dec_i = false: & \neg F_i^{var} \end{cases}$$

Figure 4 (a) shows a simple XADD with all paths and formulas determined on each node.

Definition (**Node ordering**): A node F_i^{var} in f_{XADD} is defined before node F_j^{var} if it appears before this node in a path p_s containing both nodes $\rho_i \in p_s, \rho_j \in p_s$ and has an ordering such that $i < j$. F_j^{var} can also be named the child of node F_i^{var}. A parent node F_k^{var} is defined for node F_i^{var} if node F_i^{var} appears after node F_k^{var} in a path p_s containing both nodes $\rho_i \in p_s, \rho_k \in p_s$ and has an ordering such that $k < i$. In Figure 4 (a) node 1 is the parent of node 3 and nodes $4, 5$ are the children of node C.

Definition (**Inconsistent node**): A node F_i^{var} in f_{XADD} is inconsistent if it violates any of the constraints in its parent decision node F_k^{var} in any of the path defined in the formula ψ_i over this node. In mathematical terms this is equal to the following:

$$\exists p_j \in \psi_i, (\rho_i \in p_j) : \exists \rho_k \in p_j, k < i, \phi_i \Rightarrow \phi_k = true \longleftrightarrow F_i^{var} = inconsistent. \tag{13}$$

where ϕ_i is the logical constraints of node F_i^{var} as defined in case statements. Figure 4 (b) prunes (a) of the inconsistent node 2.

Definition (**Redundant node**): A node F_i^{var} in f_{XADD} is redundant if its constraints can be addressed using any of the constraints in its child decision node F_j^{var} in any of the path defined in the formula ψ_j over this node. In mathematical terms this is equal to the following:

$$\exists p_k \in \psi_j, (\rho_j \in p_k) : \exists \rho_i \in p_k, i < j, (\phi_j \Rightarrow \phi_i) \vee (\neg\phi_j \Rightarrow \phi_i) = true \longleftrightarrow F_i^{var} = redundant. \tag{14}$$

Figure 4 (c) prunes (b) of the redundant node 1.

For any function from $\mathbb{R}^{n+m} \rightarrow \mathbb{R}$, we next describe how a reduced XADD can be constructed from an arbitrary ordered decision diagram. All algorithms that we will dene in the following sections rely on the helper function *getNode* in Algorithm 5 , which returns a more compact representation of a single internal decision node.

The algorithm *ReduceXADD* allows the construction of a compact XADD representation from an arbitrary ordered decision diagram with polynomial leaves and polynomial inequalities as the decision nodes. Algorithm 4 is defined according to the following definition:

Definition: A function graph G is reduced if it contains no vertex v with $low(v) = high(v)$, nor does it contains distinct vertices v and v such that the subgraphs rooted by v and v are isomorphic.

This algorithm recursively constructs a reduced XADD from the bottom up. Internal nodes are represented as $\langle F^{var}, F_h, F_l \rangle$, where F^{var} is the variable name, and F_h and F_l are the true and false branch node ids, respectively. Reduced nodes are stored in the *ReduceCache* table. Using the function *GetNode* (Algorithm5) any redundant decision tests are removed. This function stores a unique id for each node in the *NodeCache* table.

ReduceCache ensures that each node is visited once and a unique reduced node is generated in the final diagram. Thus *ReduceXADD* has linear running time and space according to the size of the input graph.

Algorithm 4: ReduceXADD(F)

input : F (root node id for an arbitrary ordered decision diagram)
output: F_r (root node id for reduced XADD)

```
1  begin
2    //if terminal node, return canonical terminal node
3    if F is terminal node then
4      return canonical terminal node for polynomial of F;
5    //use recursion to reduce sub diagrams
6    if F → F_r is not in ReduceCache then
7      F_h = ReduceXADD(F_h);
8      F_l = ReduceXADD(F_l);
9      //get a canonical internal node id
10     F_r = GetNode(F^var, F_h, F_l);
11     insert F → F_r in ReduceCache;
12   return F_r;
13 end
```

Algorithm 5: GetNode($\langle var, F_h, F_l \rangle$)

input : $\langle var, F_h, F_l \rangle$ (variable and true and false branches node ids for internal node)
output: F_r (canonical internal node id)

```
1  begin
2    //redundant branches
3    if F_l = F_h then
4      return F_l;
5    //check if the node exists previously
6    if ⟨var, F_h, F_l⟩ → id is not in NodeCache then
7      id = new unallocated id;
8      insert ⟨var, F_h, F_l⟩ → id in NodeCache;
9    return id;
10 end
```

We next present two successive algorithms, Algorithm 6 for removing inconsistent nodes and Algorithm 7 for removing redundant nodes. Given any XADD with potential inconsistent nodes, the output of *algPrune* (Algorithm 6) is a reduced XADD with canonical leaves, linear decisions and no inconsistent nodes. In lines 7 and 11 we test the current decision node with all previous decisions for implications in *TestImplied*. This process is performed using the definition of constraints for the LP-solver. Once a node is returned inconsistent (returns true or false from *TestImplied*) the high or low branch is returned without referring to the current node in the final XADD.

While any (parent(A) $\Rightarrow$ child(B)) relation is returned for consistency checking, at the same time this function stores other implications in the a related cache (*hmImplications*).

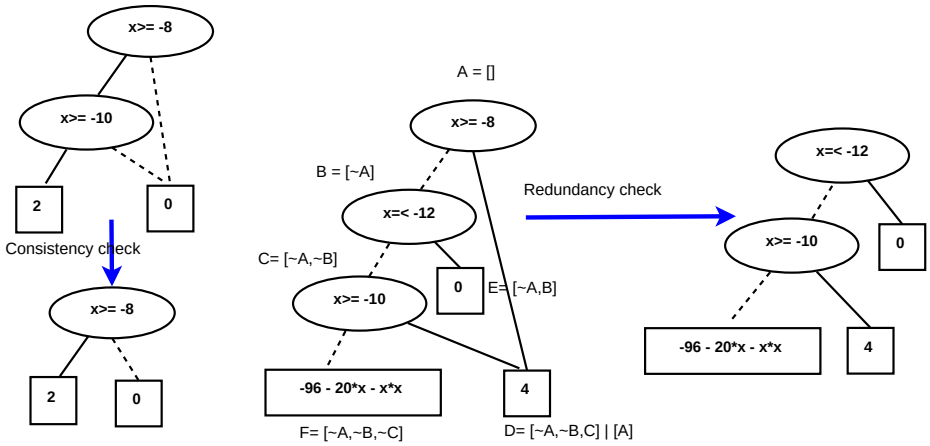

Figure 5: Using pruning algorithms for inconsistency and redundancy. Top-left figure is reduced to the bottom-left figure using the LP-Solver which recognizes the parent-child relation. The middle figure is reduced to the right-most figure using the SAT-Solver. The paths of each node in the middle graph are demonstrated. These paths along with the child-parent implications in the KB can reduce the redundant node A.

Since $A \Rightarrow B$ is already tested, we cache the following child to parent implications: (i) $B \Rightarrow A$ (2)$\neg B \Rightarrow A$ ($\neg B \Rightarrow \neg A$ is already defined by $A \Rightarrow B$).We add these to the knowledge-base (KB) of the SAT-Solver before performing the next recursive algorithm. Lines 14–15 define the decision labels and their values $(true, false)$ for the current node and add them to the set of labels and decisions which are input to the algorithm. The final result is returned from the *GetNode* algorithm. As an example consider the two cases on the left side of the Figure 5. In the upper diagram, $x \geq -8 \Rightarrow x \geq -10$ is reduced to $x \geq -8$ since it covers all the state space in the lower diagram.

Next we propose an equivalence testing approach for redundancy pruning using a SAT-solver. Consider the middle example in Figure 5 where the child node(C) implies the parent node(A). We define the recursive algorithm required to prune all child-parent implications from an XADD using this example. We introduce Algorithm 7 to remove redundant nodes. The input to this algorithm is an XADD where each node F^{var} is marked with its formula ψ_{var} (set of paths) as in the middle diagram of Figure 5.

The recursive property of the algorithm now traverses backwards from the leaf nodes up to the root node. Every time a decision node is reached (A), the true and false branches of that node are put to a satisfiability test (*Sat-Test*) to see if that branch can be eliminated from the tree due to some child-parent implication. A query is built to test the KB according to the current decision node. At node A, we test to see if the false or true branch can be removed. Considering the true branch we look for the equivalence of reaching leaf D (value 4) with the whole tree or only considering the false branch of A (which is equal to pruning out the true branch). We test the formulas $\psi_{A=F/T} \iff \psi$ where each F is the disjunct of all paths leading to this node (D). The KB here contains the child-parent implication of $C \Rightarrow \neg A$ For this example this is equal to the following satisfiability test:

$$(B \Rightarrow A) \models ((A \vee (\neg A \wedge \neg B \wedge C)) \iff (false \vee (true \wedge \neg B \wedge C)))$$

We use a SAT-solver (minisat) to entail this sentence and since the result is true, we can prune the true branch returning node B represented in the right-hand diagram of Figure

Algorithm 6: PruneInconsistent(F,*decLabel*, *decision*)

input : F (root node id for an inconsistent XADD, Decision label, Decision value)
output: F_r (root node id for a consistent XADD)

```
1  begin
2    //if terminal node, return the value
3    if F is terminal node then
4      return canonical terminal node for polynomial of F;
5    //else if internal node, find all implications consider any parent in XADD
6    //if (parent⇒ child) remove child
7    if TestImplied(decLabel,decision,F^var) then
8      F_r = PruneInconsistent(F_h,decLabel,decision);
9    //if (parent⇒ ¬child) remove child
10   if !TestImplied(decLabel,decision,−F^var) then
11     F_r = PruneInconsistent(F_l,decLabel,decision);
12   //if result of LP-solver is null
13   else
14     insert F^var → decLabel;
15     insert false → decision;
16     F_r^l = PruneInconsistent(F_l,decLabel,decision);
17     insert true → decision;
18     F_r^= PruneInconsistent(F_h,decLabel,decision);
19     insert null → decision;
20     delete F^var → decLabel;
21     F_r = GetNode(var, F_r^h, F_r^l);
22   return F_r;
23 end
```

5. A similar approach is performed for the high branch in lines 16–17. The returned value is the true or false branch if the parent node can be pruned else a new node is built using line 19 of Algorithm 5.

In principle exact SDP solutions can be obtained for arbitrary symbolic functions, we restrict XADDs to use polynomial functions only. Having proved the main properties of an XADD, next we present the operations and algorithms required for SDP for XADDs.

4.2 XADD algorithms and operations

In this section we review the symbolic operations required to perform SVI using the XADD structure. This is mainly categorized into unary and binary operations.

4.2.1 Unary XADD Operations

According to the previous section on SDP unary operations, scalar multipication $c.f$ and negation $-f$ on a function results in a function which can simply be represented as an

Algorithm 7: PruneRedundancy(X,KB,F)

input : X (root node id for a consistent XADD), KB (child-parent implications), F (formulas for each node)
output: X_r (root node id for a consistent and redundant XADD)

begin
  //current node is X, keep its path
  $Path := F(X)$;
  if *F is terminal node* **then**
    return *canonical terminal node for polynomial of F*;
  //else if internal node, add paths to low and high branch
  foreach *$path \in Path$* **do**
    //add high and low branch of internal node to current path
    insert $X_r \rightarrow path$; add $path \rightarrow F$;
    insert $X_l \rightarrow path$; add $path \rightarrow F$;
  $X_l =$ PruneRedundancy(X_l, KB, F);
  $X_h^=$ PruneRedundancy(X_h, KB, F);
  //reached the lowest decision node, perform satisfiability test
  if Sat-Test$(X, F(X_l), KB, T)$ **then**
    return X_h;
  if Sat-Test$(X, F(X_h), KB, F)$ **then**
    return X_l;

  $X_r =$ GetNode(var, X_h, X_l);
  return X_r;
end

Algorithm 8: Reorder(F)

input : F (root node for possibly unordered XADD)
output: F_r (root node for an ordered XADD)

begin
  //if terminal node, return canonical terminal node
  if *F is terminal node* **then**
    return *canonical terminal node for polynomial of F*;
  //else nodes have a *true* & *false* branch and *var* id
  if *$F \rightarrow F_r$ is not in Cache* **then**
    $F_{true} =$ Reorder$(F_{true}) \otimes \mathbb{I}[F_{var}]$;
    $F_{false} =$ Reorder$(F_{false}) \otimes \mathbb{I}[\neg F_{var}]$;
    $F_r = F_{true} \oplus F_{false}$;
    insert $F \rightarrow F_r$ in Cache;
  return F_r;
end

XADD. Apart from this restriction, substitution and marginalization of the δ function are also unary operations that can be applied to XADDs are explained below.

Restriction of a variable x_i in an XADD (F) to some formula ϕ is performed by appending ϕ to each of the decision nodes while leaves are not affected. For a binary variable restriction to a single variable x_i is equal to taking the true or false branch according to that variable ($F|_{x_i=true}$) or ($F|_{x_i=false}$). This operation can also be used to `marginalize` or `sum_out` boolean variables in a DCSA-MDP. $\sum_{x_i \in X_i}$ eliminates a variable x_i from an XADD and computed as the sum of the *true* and *false* restricted functions ($F|_{x_i=true} \oplus F|_{x_i=false}$). We omit the restriction and marginalization algorithms for XADDs since they are identical to the same operations for ADDs.

Substitution for a given function f is performed by applying σ the set of variables and their substitutions to each inequality operand such that $\phi_i\sigma : f_i\sigma$. The substitution operand effects both leaves and decision nodes and changes them according to the variable substitute.

Decisions become unordered when substituted and also when we perform maximum or minimum explained in the next section. A reorder algorithm has to be applied to the result of the substitution operand. As Algorithm 8 shows, we recursively apply the binary ADD operations of $\otimes$ and $\oplus$ to decision nodes to reorder the XADD after a substitution.

As for the integration of the δ-function on variable x we require computing $\int_x \delta[x - g(\vec{x})]fdx$. This triggers the substitution $f\{x/g(\vec{x})\}$ on f as defined above.

For the continuous maximization max_y each XADD path from root to leaf node is considered a single case partition with conjunctive constraints, and maximization is performed at each leaf subject to these constraints and all path maximums are then accumulated using the *casemax* operation for the final result. Note that in general continuous maximization is a multi-variate operation but according to the previous section, it can be decomposed into multiple univariate operations.

A univariate continuous maximization algorithm has been presented in the previous section. Starting at the root node, if the current node is a decision node, the algorithm is recursively calls on the low and high branches of this node. If the current node is a leaf node, it processes the leaf according to *Continuous Maximization*. The input to this algorithm is the leaf node and the decisions leading to this leaf node i.e., partitioning of the state-action space. The algorithm finds the maximum value according to XADDs used for the upper and lower bounds as well as the function roots. The maximum XADD resulting from these three XADDs along with applying the independent constraints define the final result. A single step of this algorithm is presented in Figure 6. The input to this step is the leaf node obtained after the regression step of Algorithm 2. Also as an input are the decisions leading to this leaf (taking the high or low branches). The XADD representation is used to represent all intermediate results as well as the final result.

In order to solve the problem of non-linear problem domains we require a continuous maximization of the continuous action variable y over a non-linear function. This maximization is performed symbolically as explained in section (2.4) by incooperating the non-linear terms at the nodes inside decision nodes. This does not effect our symbolic solution but in order to prune the resulting value function using the LP-solver explained next, we require linear decisions.

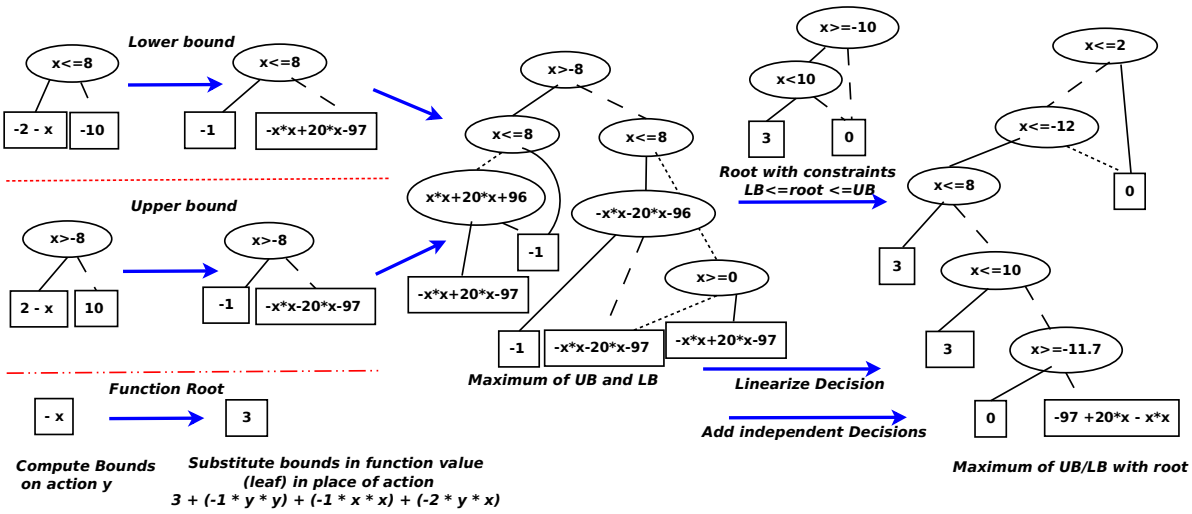

Figure 6: Representing one step of Continuous Maximization algorithm using XADDs. The upper and lower bounds of action a and the root is represented by XADDs and substituted inside the leaf node. The maximum of the upper and lower bound is then represented. The final result takes the maximum of this and the root considering the constraints and independent decisions and also linearizing the decision nodes.

The linearize algorithm (Algorithm 9) presented here is of a recursive nature similar to other XADD algorithms. For each node in the XADD starting at the root node, the algorithm linearizes the decision node and returns the two (or more) decision nodes replacing that single non-linear node. It then iterates on the low and high branches of the decision node until it returns all the leaves. The linearization process finds the roots of the non-linear function *GetRoots* in decision nodes it visits and replaces the non-linear decision of linearized decisions which are the roots of the function in the non-linear decision node.

4.2.2 Binary XADD Operations

For *all* binary operations, the function $Apply(F_1, F_2, op)$ (Algorithm 11) computes the resulting XADD.

Two canonical XADD operands F_1 and F_2 and a binary operator $op \in \{\oplus, \ominus, \otimes, \max, \min\}$ are the input to the *Apply* algorithm (Algorithm 11). The output result is a canonical XADD F_r. If the result of $Apply(F_1, F_2, op)$ is a quick computation as in line 3, it can be immediately returned. Else it checks the *ApplyCache* in line 6 for any previously stored apply result. If there is not a cache hit, the earliest variable in the ordering to branch is chosen according to *ChooseVarBranch* (Algorithm 10). Two recursive *Apply* calls are then made on the branches of this variable to compute F_l and F_h. *GetNode* checks for any redundancy in line 23 before storing it in the cache and returning the resulting XADD. We cover these steps in-depth in the following sections.

Algorithm 9: ReduceLinearize(F)

input : F (root node id for an arbitrary ordered decision diagram)
output: F_r (root node id for linearized XADD)

```
1  begin
2      //if terminal node, return canonical terminal node
3      if F is terminal node then
4          return canonical terminal node for polynomial of F;
5      if F → F_r is not in ReduceCache then
6          //use recursion to reduce sub diagrams
7          F_h = ReduceLinearize(F_h);
8          F_l = ReduceLinearize(F_l);
9          //get a linearized internal node id
10         F_r = GetRoots(F^var, F_h, F_l);
11         insert F → F_r in ReduceCache;
12     return F_r;
13 end
```

Algorithm 10: ChooseVarBranch(F_1, F_2)

input : F_1 (root node id for operand 1),
F_2 (root node id for operand 2)
output: *var* (selected variable to branch)

```
1  begin
2      //select the variable to branch based on the order criterion
3      if F_1 is a non-terminal node then
4          if F_2 is a non-terminal node then
5              if F_1^var comes before F_2^var then
6                  var = F_1^var;
7              else
8                  var = F_2^var;
9
10         else
11             var = F_1^var;
12
13     else
14         var = F_2^var;
15     return var;
16 end
```

4.2.3 Apply Algorithm for binary operations of XADDs

Terminal computation: The function *ComputeResult* determines if the result of a computation can be immediately computed without recursion. The entries denote a number of pruning optimizations that immediately return a node without recursion. For the dis-

Case number	Case operation	Return
1	F_1 *op* $F_2; F_1 = Poly_1; F_2 = Poly_2$	$Poly_1$ *op* $Poly_2$
2	$F_1 \oplus F_2; F_2 = 0$	F_1
3	$F_1 \oplus F_2; F_1 = 0$	F_2
4	$F_1 \ominus F_2; F_2 = 0$	F_1
5	$F_1 \otimes F_2; F_2 = 1$	F_1
6	$F_1 \otimes F_2; F_1 = 1$	F_2
7	$F_1 \otimes F_2; F_2 = 0$	0
8	$F_1 \otimes F_2; F_1 = 0$	0
9	$\max(F_1, F_2)$	pic
10	$\min(F_1, F_2)$	pic
11	other	*null*

Table 1: Input case and result for the method *ComputeResult* for binary operations $\oplus$, $\ominus$ and $\otimes$ for XADDs.

crete maximization (minimization) operation (entries 9 and 10) , for every two leaf nodes f and g an additional decision node $f > g$ ($f < g$) is introduced to represent the maximum(minimum). This may cause out-of-order decisions which can be solved by the reordering Algorithm 8.

Caching: If the result of *ComputeResult* is empty, in the next step we check the *ApplyCache* for any previously computed operation using this set of operands and operations. To increase the chance of a match, all items stored in a cache are made canonical.

Recursive computation: If a call to Apply is unable to immediately compute a result or reuse a previously cached computation, we must recursively compute the result. For this we have four cases based on the variable *var* chosen by *ChooseVarBranch*:

- F_1 is a non-terminal node and $F_1^{var} = var$: The high and low branch of the first operand passed to a recursive *Apply* is equal to the high and low branches of F_1.
- F_1 is non-terminal nodes and $F_1^{var} \neq var$: In this case only the low branch of the final computation of the first operand is set to F_1.
- F_2 is a non-terminal node and $F_2^{var} = var$: The high and low branch of the second operand passed to a recursive *Apply* is equal to the high and low branches of F_2.
- F_2 is non-terminal nodes and $F_2^{var} \neq var$: In this case only the high branch of the final computation of the first operand is set to F_2.

Finally for the high and low branch of the final computation, two recursive calls are made to *Apply* and the result is in a canonical form returned by *GetNode*. Having defined the efficient representation of XADDs, next we show results from implementing the SVI algorithms using this structure.

5. Experimental Results

We implemented two versions of our proposed SVI algorithms using XADDs — one that does not prune nodes of the XADD and another that uses a linear programming solver to prune

Algorithm 11: Apply(F_1, F_2, op)

input : F_1 (root node id for operand 1),
F_2 (root node id for operand 2),
op (binary operator, $op \in \{\oplus, \ominus, \otimes\}$)
output: F_r (root node id for the resulting reduced XADD)

```
1  begin
2      //check if the result can be immediately computed
3      if ComputeResult(F_1, F_2, op) → F_r ≠ null then
4          return F_r;
5      //chech if we previously computed the same operation
6      if ⟨F_1, F_2, op⟩ → F_r is not in ApplyCache then
7          //choose variable to brach
8          var = ChooseVarBranch(F_1, F_2);
9          //set up nodes for recursion
10         if F_1 is non-terminal ∧ var = F_1^var then
11             F_l^v1 = F_{1,l};
12             F_h^v1 = F_{1,h};
13         else
14             F_{l,h}^v1 = F_1;
15         if F_2 is non-terminal ∧ var = F_2^var then
16             F_l^v2 = F_{2,l};
17             F_h^v2 = F_{2,h};
18         else
19             F_{l,h}^v2 = F_2;
20         //use recursion to compute true and false branches for resulting XADD
21         F_l = Apply(F_l^v1, F_l^v2, op);
22         F_h = Apply(F_h^v1, F_h^v2, op);
23         F_r = GetNode(var, F_h, F_l);
24         //save the result to reuse in the future
25         insert ⟨F_1, F_2, op⟩ → F_r into ApplyCache;
26     return F_r;
27 end
```

unreachable nodes (for problems with linear XADDs) and then performs a satisfiability check to prune redundant paths — and tested these algorithms on different problems.

For CA-HMDPs we evaluated SVI on a didactic nonlinear Mars Rover example and two problems from Operations Research (OR) Inventory Control defined in the introduction and Reservoir Management all of which are described below. For comparison purposes the DA-HMDPs example domains are discretized by their action space. [8]

8. All Java source code and a human/machine readable file format for all domains needed to reproduce the results in this paper can be found online at `http://code.google.com/p/xadd-inference`.

5.1 Domains

Mars Rover In a Continuous Action setting, a Mars Rover state consists of its continuous position x along a given route. In a given time step, the rover may move a continuous distance $y \in [-10, 10]$. The rover receives its greatest reward for taking a picture at $x = 0$, which quadratically decreases to zero at the boundaries of the range $x \in [-2, 2]$. The rover will automatically take a picture when it starts a time step within the range $x \in [-2, 2]$ and it only receives this reward once.

Using boolean variable $b \in \{0, 1\}$ to indicate if the picture has already been taken ($b = 1$), x' and b' to denote post-action state, and R to denote reward, we express the Mars Rover CA-HMDP using piecewise dynamics and reward:

$$P(b'{=}1|x, b) = \begin{cases} b \vee (x \geq -2 \wedge x \leq 2) : & 1.0 \\ \neg b \wedge (x < -2 \vee x > 2) : & 0.0 \end{cases}$$

$$P(x'|x, y) = \delta \left(x' - \begin{cases} y \geq -10 \wedge y \leq 10 : & x + y \\ y < -10 \vee y > 10 : & x \end{cases} \right)$$

$$R(x, b) = \begin{cases} \neg b \wedge x \geq -2 \wedge x \leq 2 : & 4 - x^2 \\ b \vee x < -2 \vee x > 2 : & 0 \end{cases}$$

If our objective is to maximize the long-term *value* V (i.e., the sum of rewards received over an infinite horizon of actions), then we can write the optimal value achievable from a given state in Mars Rover as a function of state variables:

$$V = \begin{cases} \neg takepic_1 \wedge takepic_2 \wedge (4 - x^2 - y^2 \geq 0) \wedge (5 - x^2 - y^2 \geq 0) : & 4 - x^2 - y^2 \\ takepic_1 \wedge \neg takepic_2 \wedge (2 - x^2 - y^2 \geq 0) \wedge (3 - x^2 - y^2 \geq 0) : & 2 - x^2 - y^2 \\ \neg takepic_1 \wedge takepic_2 \wedge (4 - x^2 - y^2 \geq 0) \wedge (5 - x^2 - y^2 \leq 0) : & -1 \\ takepic_1 \wedge \neg takepic_2 \wedge (2 - x^2 - y^2 \geq 0) \wedge (3 - x^2 - y^2 \leq 0) : & -1 \\ else & : 0 \end{cases}$$

The value function is piecewise and non-linear, and it contains non-rectangular decision boundaries like $4 - x^2 - y^2 \geq 0$. Figure 7 presents the 0-, 1-, and 2-step time horizon solutions for this problem; further, in symbolic form, we display both the 1-step time horizon value function and corresponding optimal policy in Figure 8. Here, the piecewise nature of the transition and reward function leads to piece- wise structure in the value function and policy. Yet despite the intuitive and simple nature of this result, we are unaware of prior methods that can produce such exact solutions.

Reservoir Management Reservoir management is well-studied in the OR literature (Mahootchi, 2009; Yeh, 1985). The key continuous decision is how much elapsed time e to *drain* (or *not drain*) each reservoir to maximize electricity revenue over the decision-stage horizon while avoiding reservoir overflow and underflow. Cast as a CA-HMDP, we believe SVI provides the first approach capable of deriving an exact closed-form non-myopic optimal policy for all levels.

We examine a 2-reservoir problem with respective levels $(l_1, l_2) \in [0, \infty]^2$ with reward penalties for overflow and underflow and a reward gain linear in the elapsed time e for

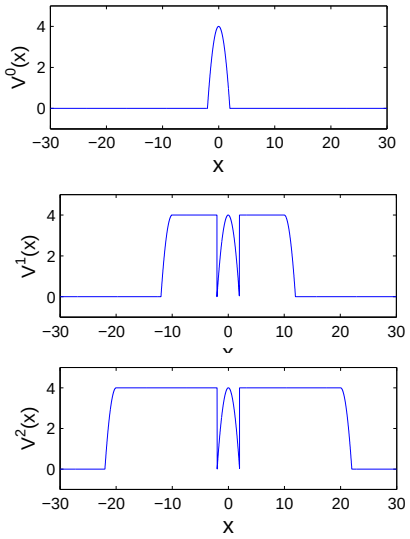

Figure 7: Optimal sum of rewards (value) $V^t(x)$ for $b = 0$ (*false*) for time horizons (i.e., decision stages remaining) $t = 0$, $t = 1$, and $t = 2$ on the Continuous Action Mars Rover problem. For $x \in [-2, 2]$, the rover automatically takes a picture and receives a reward quadratic in x. We initialized $V^0(x, b) = R(x, b)$; for $V^1(x)$, the rover achieves non-zero value up to $x = \pm 12$ and for $V^2(x)$, up to $x = \pm 22$.

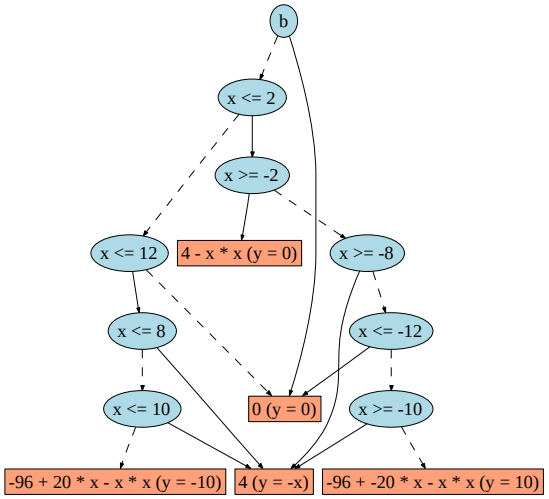

Figure 8: Optimal value function $V^1(x)$ for the Continuous Action Mars Rover problem represented as an extended algebraic decision diagram (XADD). Here the solid lines represent the *true* branch for the decision and the dashed lines the *false* branch. To evaluate $V^1(x)$ for any state x, one simply traverses the diagram in a decision-tree like fashion until a leaf is reached where the non-parenthetical expression provides the *optimal value* and the parenthetical expression provides the *optimal policy* ($y = \pi^{*,1}(x)$) to achieve value $V^1(x)$.

electricity generated in periods when the *drain*(e) action drains water from l_2 to l_1 (the other action is *no-drain*(e)); we assume deterministic rainfall replenishment and present the reward function as:

$$R = \begin{cases} ((50 - 200 * e) \le l_1 \le (4500 - 200 * e)) \wedge ((50 + 100 * e) \le l_2 \le (4500 + 100 * e)) & : e \\ ((50 + 300 * e) \le l_1 \le (4500 + 300 * e)) \wedge ((50 - 400 * e) \le l_2 \le (4500 - 400 * e)) & : 0 \\ \text{otherwise} & : -\infty \end{cases}$$

The transition function for levels of the *drain* action is defined below. Note that for the *no-drain* action, the $500 * e$ term is not involved.

$$l_1' = (400 * e + l_1 - 700 * e + 500 * e)$$
$$l_2' = (400 * e + l_2 - 500 * e)$$

Similar to the discrete version of the Inventory Control problem in the introduction, the DA-HMDP setting for these two problems defines discrete actions by partitioning the action space of each domain into 1/10 slices. For the Mars Rover problem this defines 2 actions ($a_1 = [-10, 0], a_2 = [0, 10]$), for Reservoir Management the discrete time

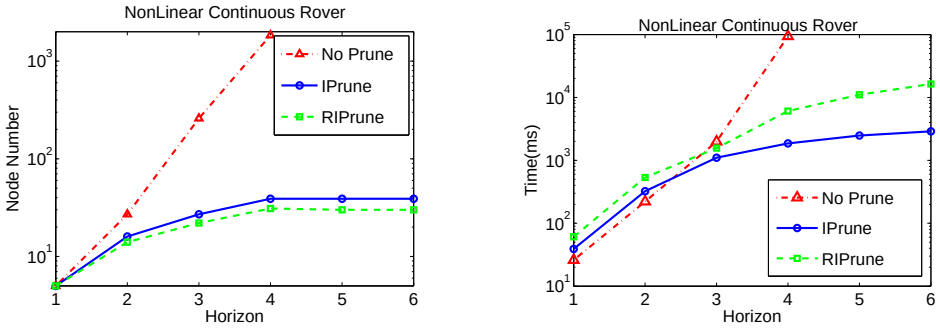

Figure 9: Space (# XADD nodes in value function) and time for different iterations (horizons) of SDP on Nonlinear CONTINUOUS ACTION MARS ROVER with 3 different results based on pruning techniques. Results are shown for the XADD with no pruning technique, with only inconsistency checking (using LP-solver) and with both the inconsistent and redundancy checking (using LP-Solver and SAT-Solver).

action set is a set of 20 elements where $e_1 = [0, 10]$ and $e_20 = [190, 200]$ and the transition and reward functions are defined according to these constant values. We now provide the empirical results obtained from implementing our algorithms.

5.2 Results

For the DISCRETE ACTION MARS ROVER domains, we have run experiments to evaluate our SDP solution in terms of time and space cost while varying the horizon and problem size.

Because the reward and transition functions for DISCRETE ACTION MARS ROVER LINEAR use piecewise linear case statements, we note the optimal value function in this domain is also piecewise linear. Hence in this domain, we use a linear constraint feasibility checker to prune unreachable paths in the XADD, we also compare solutions for MARS ROVER with and without this pruning.

We notice here that the piecewise boundaries for all three plots clearly demonstrate non-rectangular boundaries. In particular, the value function plot for the MARS ROVER demonstrates nonlinear piecewise boundaries with each piece being a nonlinear function of the state — it has the shape of stacked quadratic cones with each lower cone representing the cost of first moving from points farther away from the picture being receiving the value for taking the picture within the radius limits.

To the best of our knowledge, these results demonstrate the first exact analytical solutions for HMDPs having optimal value functions with general linear and even nonlinear piecewise boundaries.

For the CONTINUOUS ACTION MARS ROVER problem, we present the time and space analysis for the problem description defined in the introduction in Figure 9. Here three evaluations are performed based on the pruning algorithms in the previous section. We note that without the LP-Solver Algorithm 6, the algorithm can not go beyond the forth iteration as it produces many inconsistent nodes. For the redundancy check of Algorithm 7, as it can be seen in the graph, a reduction of at most 25 % is achieved. The time required to remove redundancy has increased due to the calls made to the SAT-Solver.

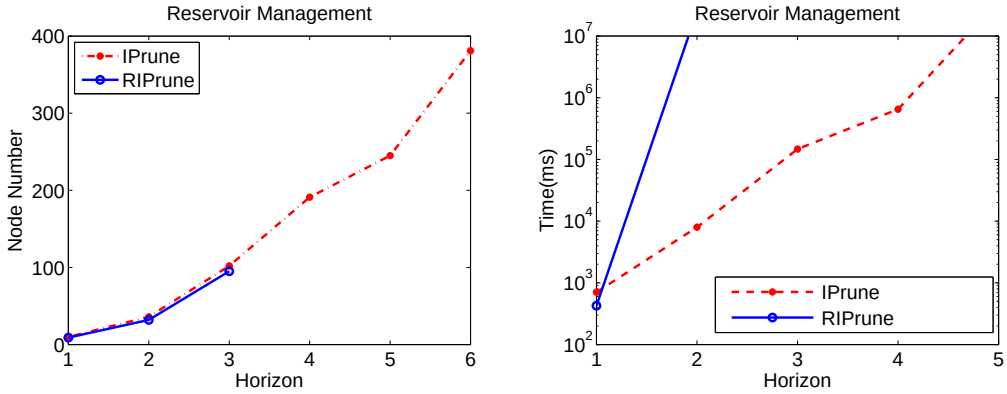

Figure 10: Reservoir Management: space and elapsed time vs. horizon. Comparing the two algorithms for inconsistency pruning (IPrune) and redundancy and inconsistency pruning (RIPrune).

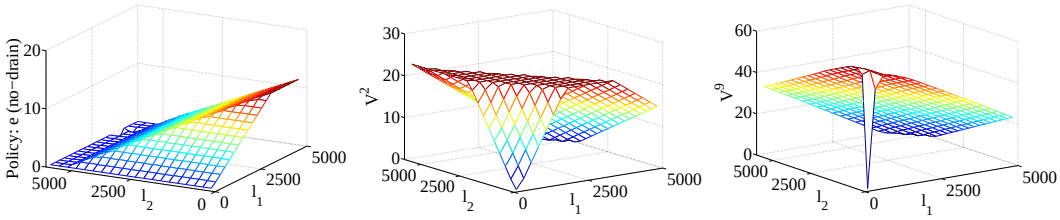

Figure 11: Reservoir Management : *(left)* Policy $\textit{no-drain}(e) = \pi^{2,*}(l_1, l_2)$ showing on the z-axis the elapsed time e that should be executed for *no-drain* conditioned on the states; *(middle)* $V^2(l_1, l_2)$; *(right)* $V^9(l_1, l_2)$.

Figure 10 compares inconsistency pruning and redundancy pruning for the reservoir management problem. In this setting, very little node reduction can be gained from redundancy pruning due to the different water levels and their correlation. The time has increased rapidly after the second iteration for the redundancy pruning due to the equivalence checking of the SAT-Solver. This suggests that in some domains using redundancy pruning in not efficient and should be omitted from the final results. We next show results for the Inventory Control problem without redundancy pruning.

In Figure 11, we show a plot of the optimal closed-form policy at $h = 2$: the solution interleaves $\textit{drain}(e)$ and $\textit{no-drain}(e)$ where even horizons are the latter; here we see that we avoid draining for the longest elapsed time e when l_2 is low (wait for rain to replenish) and l_1 is high (draining water into it could overflow it). $V^2(l_1, l_2)$ and $V^9(l_1, l_2)$ show the progression of convergence from horizon $h = 2$ to $h = 9$ — low levels of l_1 and l_2 allow the system to generate electricity for the longest total elapsed time over 9 decision stages.

In Figure 12, we provide a time and space analysis of deterministic- and stochastic-demand (resp. DD and SD) variants of the SCIC and MJCIC problem for up to three items (the same scale of problems often studied in the OR literature); for each number of items $n \in \{1, 2, 3\}$ the state (inventory levels) is $\vec{x} \in [0, \infty]^n$ and the action (reorder amounts) is $\vec{y} \in [0, \infty]^n$. Orders are made at one month intervals and we solve for a horizon up to $h = 6$ months. Here we see that linear feasibility checking/pruning in the XADD is crucial

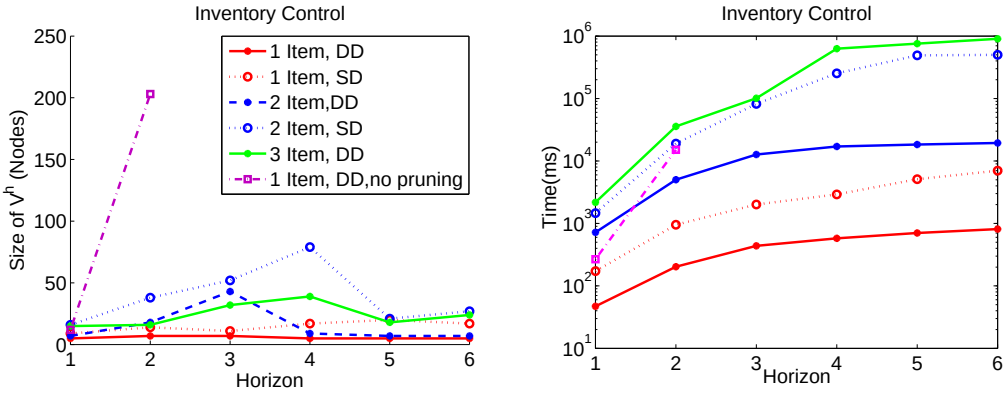

Figure 12: INVENTORY CONTROL : space and time vs. horizon.

– we cannot solve beyond $h = 2$ without it for 1 item! While solving for larger numbers of items and SD (rather than DD) both increase time and space, the solutions quickly reach quiescence indicating structural convergence.

6. Related Work

The most relevant vein of related work for DA-HMDPs is that of (Feng et al., 2004) and (Li & Littman, 2005) which can perform exact dynamic programming on HMDPs with rectangular piecewise linear reward and transition functions that are delta functions. While SDP can solve these same problems, it removes both the rectangularity and piecewise restrictions on the reward and value functions, while retaining exactness. Heuristic search approaches with formal guarantees like HAO* (Meuleau et al., 2009) are an attractive future extension of SDP; in fact HAO* currently uses the method of (Feng et al., 2004), which could be directly replaced with SDP. While (Penberthy & Weld, 1994) has considered general piecewise functions with linear boundaries (and in fact, we borrow our linear pruning approach from this paper), this work only applied to fully deterministic settings, not HMDPs.

Other work has analyzed limited HMDPS having only one continuous state variable. Clearly rectangular restrictions are meaningless with only one continuous variable, so it is not surprising that more progress has been made in this restricted setting. One continuous variable can be useful for optimal solutions to time-dependent MDPs (TMDPs) (Boyan & Littman, 2001). Or phase transitions can be used to arbitrarily approximate one-dimensional continuous distributions leading to a bounded approximation approach for arbitrary single continuous variable HMDPs (Marecki, Koenig, & Tambe, 2007). While this work cannot handle arbitrary stochastic noise in its continuous distribution, it does exactly solve HMDPs with multiple continuous state dimensions.

There are a number of general HMDP approximation approaches that use approximate linear programming (Kveton, Hauskrecht, & Guestrin, 2006) or sampling in a reinforcement learning style approach (Remi Munos, 2002). In general, while approximation methods are quite promising in practice for HMDPS, the objective of this paper was to push the boundaries of *exact* solutions; however, in some sense, we believe that more expressive exact solutions may also inform better approximations, e.g., by allowing the use of data structures with non-rectangular piecewise partitions that allow higher fidelity approximations.

As for CA-HMDPs, there has been prior work in control theory. The field of linear-quadratic Gaussian (LQG) control (Athans, 1971) which use linear dynamics with continuous actions, Gaussian noise, and quadratic reward is most closely related. However, these exact solutions do not extend to discrete and continuous systems with *piecewise* dynamics or reward. Combining this work with initial state focused techniques (Meuleau et al., 2009) and focused approximations that exploit optimal value structure (St-Aubin, Hoey, & Boutilier, 2000) or further afield (Remi Munos, 2002; Kveton et al., 2006; Marecki et al., 2007) are promising directions for future work.

7. Concluding Remarks

In this paper, we introduced a new symbolic approach to solving continuous problems in HMDPs exactly. In the case of discrete actions and continuous states, using arbitrary reward functions and expressive nonlinear transition functions far exceeds the exact solutions possible with existing HMDP solvers. As for continuous states and actions, a key contribution is that of *symbolic constrained optimization* to solve the continuous action maximization problem. We believe this is the first work to propose optimal closed-form solutions to MDPs with *multivariate* continuous state *and* actions, discrete noise, *piecewise* linear dynamics, and *piecewise* linear (or restricted *piecewise* quadratic) reward; further, we believe our experimental results are the first exact solutions to these problems to provide a closed-form optimal policy for all (continuous) states. While our method is not scalable for 100's of items, it still represents the first general exact solution methods for capacitated multi-inventory control problems. And although a linear or quadratic reward is quite limited but it has appeared useful for single continuous resource or continuous time problems such as the water reservoir problem.

In an effort to make SDP practical, we also introduced the novel XADD data structure for representing arbitrary piecewise symbolic value functions and we addressed the complications that SDP induces for XADDs, such as the need for reordering and pruning the decision nodes after some operations. All of these are substantial contributions that have contributed to a new level of expressiveness for HMDPS that can be exactly solved.

There are a number of avenues for future research. First off, it is important examine what generalizations of the transition function used in this work would still permit closed-form exact solutions. In terms of better scalability, one avenue would explore the use of initial state focused heuristic search-based value iteration like HAO* (Meuleau et al., 2009) that can be readily adapted to use SDP. Another avenue of research would be to adapt the lazy approximation approach of (Li & Littman, 2005) to approximate HMDP value functions as piecewise linear XADDs with linear boundaries that may allow for better approximations than current representations that rely on rectangular piecewise functions. Along the same lines, ideas from APRICODD (St-Aubin et al., 2000) for bounded approximation of discrete ADD value functions by merging leaves could be generalized to XADDs. Altogether the advances made by this work open up a number of potential novel research paths that we believe may help make rapid progress in the field of decision-theoretic planning with discrete and continuous state.

With the current solution for continuous states and actions, we can apply our methods to real-world data from the Inventory literature with more exact transitions and rewards. Fully

stochastic distributions are required for these problems which is a major future direction by in-cooperating a noise parameter in the models. Also we have looked into value iteration for both problems, solving the symbolic policy iteration algorithm for problems with simple policies can prove to be effective in certain domains. The other promising direction is to extend the current exact solution for non-linear functions and solving polynomial equations using computational geometric techniques.

Acknowledgements

References

Arrow, K., Karlin, S., & Scarf, H. (1958). *Studies in the mathematical theory of inventory and production*. Stanford University Press.

Athans, M. (1971). The role and use of the stochastic linear-quadratic-gaussian problem in control system design. *IEEE Transaction on Automatic Control*, *16*(6), 529–552.

Bahar, R. I., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., & Somenzi, F. (1993). Algebraic Decision Diagrams and their applications. In *IEEE /ACM International Conference on CAD*.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Bitran, G. R., & Yanasse, H. (1982). Computational complexity of the capacitated lot size problem. *Management Science*, *28*(10), 1271–81.

Boutilier, C., Dean, T., & Hanks, S. (1999). Decision-theoretic planning: Structural assumptions and computational leverage. *JAIR*, *11*, 1–94.

Boutilier, C., Friedman, N., Goldszmidt, M., & Koller, D. (1996). Context-specific Independence in Bayesian Networks. In *Proc. 12th UAI*, pp. 115–123.

Boutilier, C., Reiter, R., & Price, B. (2001). Symbolic dynamic programming for first-order MDPs. In *IJCAI-01*, pp. 690–697, Seattle.

Boyan, J., & Littman, M. (2001). Exact solutions to time-dependent MDPs. In *Advances in Neural Information Processing Systems NIPS-00*, pp. 1026–1032.

Bresina, J. L., Dearden, R., Meuleau, N., Ramkrishnan, S., Smith, D. E., & Washington, R. (2002). Planning under continuous time and resource uncertainty: A challenge for ai. In *Uncertainty in Artificial Intelligence (UAI-02)*, pp. 77–84.

Bryant, R. E. (1986). Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, *35*(8), 677–691.

Cox, D., Little, J., & O'Shea, D. (2007). *Ideals, varieties and algorithms. An introduction to computational algebraic geometry and commutative algebra*. Springer, New York.

Dean, T., & Kanazawa, K. (1989). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*(3), 142–150.

Delgado, K. V., Sanner, S., & de Barros, L. N. (2010). Efficient solutions to factored mdp with imprecise transition probabilities. *Artificial Intelligence Journal (AIJ)*, *175*, 1498–1527.

Feng, Z., Dearden, R., Meuleau, N., & Washington, R. (2004). Dynamic programming for structured continuous markov decision problems. In *Uncertainty in Artificial Intelligence (UAI-04)*, pp. 154–161.

Hoey, J., St-Aubin, R., Hu, A., & Boutilier, C. (1999). SPUDD: Stochastic planning using decision diagrams. In *UAI-99*, pp. 279–288, Stockholm.

Kveton, B., Hauskrecht, M., & Guestrin, C. (2006). Solving factored mdps with hybrid state and action variables. *Journal Artificial Intelligence Research (JAIR)*, *27*, 153–201.

Lamond, B., & Boukhtouta, A. (2002). Water reservoir applications of markov decision processes. In *International Series in Operations Research and Management Science, Springer*.

Li, L., & Littman, M. L. (2005). Lazy approximation for solving continuous finite-horizon mdps. In *National Conference on Artificial Intelligence AAAI-05*, pp. 1175–1180.

Mahootchi, M. (2009). *Storage System Management Using Reinforcement Learning Techniques and Nonlinear Models*. Ph.D. thesis, University of Waterloo,Canada.

Marecki, J., Koenig, S., & Tambe, M. (2007). A fast analytical algorithm for solving markov decision processes with real-valued resources. In *International Conference on Uncertainty in Artificial Intelligence IJCAI*, pp. 2536–2541.

Meuleau, N., Benazera, E., Brafman, R. I., Hansen, E. A., & Mausam (2009). A heuristic search approach to planning with continuous resources in stochastic domains. *Journal Artificial Intelligence Research (JAIR)*, *34*, 27–59.

Penberthy, J. S., & Weld, D. S. (1994). Temporal planning with continuous change. In *National Conference on Artificial Intelligence AAAI*, pp. 1010–1015.

Remi Munos, A. M. (2002). Variable resolution discretization in optimal control. *Machine Learning*, *49, 2–3*, 291–323.

St-Aubin, R., Hoey, J., & Boutilier, C. (2000). APRICODD: Approximate policy construction using decision diagrams. In *NIPS-2000*, pp. 1089–1095, Denver.

Wu, T., Shi, L., & Duffie, N. A. (2010). An hnp-mp approach for the capacitated multi-item lot sizing problem with setup times. *IEEE T. Automation Science and Engineering*, *7*(3), 500–511.

Yeh, W. G. (1985). Reservoir management and operations models: A state-of-the-art review. *Water Resources research*, *21,12*, 17971818.