# 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 DP 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 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 or it may moving continuously while navigating.

Other examples include Inventory Control problems (Arrow et al., 1958) for continuous resources such as petroleum products where 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). *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.*

*To further clarify we provide two example of hybrid state* Inventory Control *with discrete or continuous 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. The order action $a_j$ takes two values of $(0, 200)$ where the first indicates no ordering and the second assumes maximum amount of ordering which is 200. 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 $j$.*

Continuous Action Inventory Control *(*CAIC*): In a more general setting to this problem, 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:*

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

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

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

*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 using $-\infty$, in this case having the capacity lower than zero at any time and having capacity higher than that of the total C.*

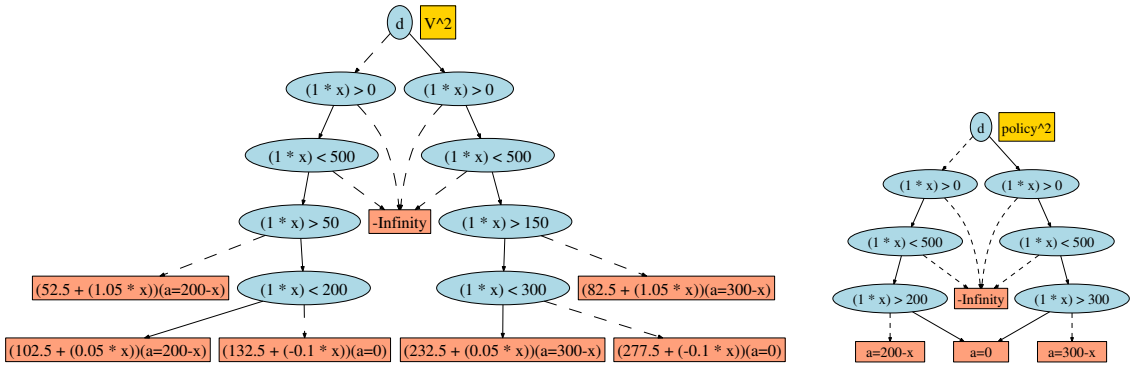



Figure 1: Optimal value function $V^2(x)$ for the CAIC problem represented as an extended algebraic decision diagram (XADD). Here the solid lines represent the *true* branch for the decision and the dashed lines the *false* branch. To evaluate $V^2(x)$ for any state $x$, one simply traverses the diagram 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); Simplified diagram for the optimal policy for the second iteration $\pi^2$ consistent with Scarf's policy (Right).

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 CAIC problem [1] the optimal value function for the second horizon is defined in Figure 1 (left):

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

The policy obtained from this piecewise and linear value function and $V^2$ itself are shown in Figure 1 using an extended algebraic decision diagram (XADD) representation which allows efficient implementation of the *case calculus* for arbitrary functions. According to 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. According to this we can rewrite Scarf's policy where each slice

1. For purposes of concise exposition and explanation of the optimal value function and policy, this example uses continuous univariate state and action; the empirical results will later discuss a range of HMDPs with multivariate hybrid state and action.

of the state space matches with this general rule:

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

While this simple example illustrates the power of using continuous variables, for a multivariate problem it is the very *first solution* to exactly solving problems such as the DAIC and CAIC. We propose novel ideas to work around some of the expressiveness limitations of previous approaches, significantly generalizing 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 $\mathcal{T}(s, s')$ and receives a reward $\mathcal{R}(s, a)$. 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

The formal definition of a hybrid MDP (HMDP) is presented in the following:

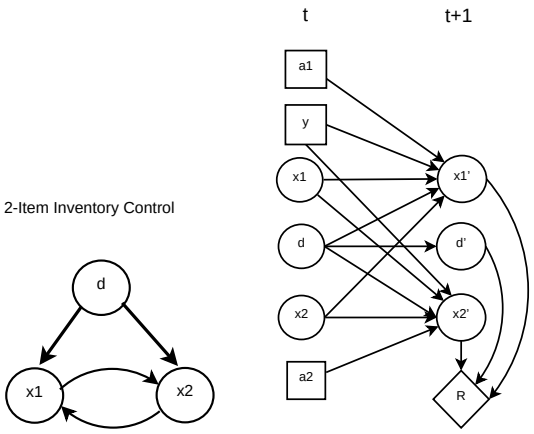
- States are represented by vectors of variables $(\vec{b}, \vec{x}) = (b_1, \ldots, b_n, x_1, \ldots, x_m)$. We assume that each $b_i \in \{0, 1\}$ $(1 \leq i \leq m)$ is boolean and each $x_j \in \mathbb{R}$ $(1 \leq j \leq n)$ is continuous.
- A finite set of $p$ actions $A = \{a_1(\vec{y}_1), \ldots, a_p(\vec{y}_p)\}$, 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.
- The 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 $\vec{y}$;
- Reward function $\mathcal{R}(\vec{b}, \vec{x}, \vec{b}', \vec{x}', a, \vec{y})$, which specifies the immediate reward obtained by taking action $a(\vec{y})$ in state $(\vec{b}, \vec{x})$;
- Discount factor $\gamma,\ 0 \leq \gamma \leq 1$. [2]

A policy $\pi$ 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 [3] $\Pi^* = (\pi^{*,1}, \ldots, \pi^{*,H})$ that maximizes the expected sum of discounted rewards over a horizon $h \in H; H \geq 0$:

$$V^{\Pi^*}(\vec{x}) = E_{\Pi^*}\left[\sum_{h=0}^{H} \gamma^h \cdot r^h \Big| \vec{b}_0, \vec{x}_0\right]. \tag{4}$$

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

2. 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.
3. 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 $\epsilon$-optimal policy for arbitrary $\epsilon$ can be computed by SDP in finite time.




$$P(d' = true \mid \vec{x}, d, \vec{a}, y) = \begin{cases} d : 0.7 \\ \neg d : 0.3 \end{cases}$$

$$P(x'_1 \mid d, d', \vec{x}, \vec{a}, y) = \delta\left(x'_1 - \begin{cases} d : x_1 + a_1 - 150 \\ \neg d : x_1 + a_1 - 50 \end{cases}\right)$$

$$R(x_1, x'_1, d, d', a_1, y) = \begin{cases} d \wedge x_1 \geq 150 : 150 - 0.1 * a_1 - 0.05 * x_1 \\ d \wedge x_1 < 150 : \ x_1 - 0.1 * a_1 - 0.05 * x_1 \\ \neg d \wedge x_1 \geq 50 : 50 - 0.1 * a_1 - 0.05 * x_1 \\ \neg d \wedge x_1 < 50 : \ x_1 - 0.1 * a_1 - 0.05 * x_1 \\ x_1 < 0 \vee (x_1 + x_2) \geq 500 : \quad -\infty \end{cases}$$

Figure 2: *(left)* Network topology between state variables in the 2-item continuous action Inventory Control (CAIC) problem; *(middle)* DBN structure representing the transition and reward function; *(right)* transition probabilities and reward function in terms of CPF and PLE for $x_1$.

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 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 a 2-item 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 (1). 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 the following properties:

- PLEs can only condition on the action, current state, and previous state variables
- PLEs are deterministic meaning that to be represented by probabilities they must be encoded using Dirac $\delta[\cdot]$ functions (example forthcoming)
- 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 (1) can be expressed in PLE format such as the right figure in Figure 2:

$$P(x_1'|d, d', \vec{x}, \vec{a}, y) = \delta\left(x_1' - \begin{cases} d: & x_i + a_i - 150 \\ \neg d: & x_i + a_i - 50 \end{cases}\right)$$

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. [4] We note that this representation effectively allows modeling of continuous variable transitions as a mixture of $\delta$ 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 DA-HMDPs representation is more general than (Feng et al., 2004; Li & Littman, 2005; Meuleau et al., 2009) in that we do not restrict the equations to be linear, but rather allow it to specify *arbitrary* functions (e.g., nonlinear).

The reward function in DA-HMDPs is defined as *arbitrary* function of the current state for each action $a \in A$. While empirical 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) as in equation (2); or

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

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

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:

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

$$Q_a^h(\vec{b},\vec{x},\vec{y}) = \sum_{\vec{b'}} \int \left( \prod_{i=1}^{n} P(b_i'|\vec{b},\vec{x},\vec{b'},\vec{x'},a,\vec{y}) \prod_{j=1}^{m} P(x_j'|\vec{b},\vec{b'},\vec{x},\vec{x'},a,\vec{y}) \right) \left[ R(\vec{b},\vec{b'},\vec{x},\vec{x'},a,\vec{y}) + \gamma \cdot V^{h-1}(\vec{b'},\vec{x'}) d\vec{x'} \right] \tag{5}$$

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:

$$V^h(\vec{b},\vec{x}) = \max_{a \in A} \max_{\vec{y} \in \mathbb{R}^{|\vec{y}|}} \left\{ Q_a^h(\vec{b},\vec{x},\vec{y}) \right\} \tag{6}$$

For discrete actions, maximization over the continuous parameter $\vec{y}$ is omitted. The $max_{\vec{y}}$ operator defined in the next section 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.

## 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. Specifically SDP provides symbolic abstraction of all functions which allows building distinct logical partitions on the state space to represent the policy and value function. 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. Using the *mathematical* definitions of the previous section, we show how to *compute* equations (5) and (6) symbolically.

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

### 3.1 Case Representation and Operations

The symbolic case representation presented in this section, extends the SDP framework in (Boutilier et al., 2001), which addresses piecewise constant functions, to general piecewise

functions. This allows an expressive representation of functions in the following case form:

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

Here, $\phi_i$ is a logical formulae over the hybrid state $(\vec{b}, \vec{x}), \vec{b} \in \mathbb{B}^m, \vec{x} \in \mathbb{R}^n$ and is defined by arbitrary logical combinations $(\wedge, \vee, \neg)$ of (1) boolean variables in $\vec{b}$ and (2) inequality relations $(\geq, >, \leq, <)$ over continuous variables $\vec{x}$. The following example is a simple function in this case form:

$$f_1 = \begin{cases} b \wedge (x_1 + x_2 > 5) : & x_1 + 3 \\ \neg b \wedge (x_1 + x_2 \leq 5) : & x_1 + x_2 \end{cases}$$

Each $\phi_i$ is disjoint from the other $\phi_j$ $(j \neq i)$. However the $\phi_i$ may not exhaustively cover the state space. Hence $f$ may only be a partial function and undefined for some variable assignments. In general $f_i$ can be defined as an arbitrary function. However due to the restrictions mentioned later, for CA-HMDPs, we assume $f_i$ to be either linear or quadratic on $\vec{x}$. Similarly we assume $\phi_i$ is defined over linear inequalities.

In the next section, the main case operations required to perform SDP are presented. Note that if $f$ is continuous then all SDP operations preserve this continuous property.

### Scalar multiplication and Negation

Unary operations include the scalar multiplication $c \cdot f$ (for a constant $c \in \mathbb{R}$) and negation $-f$ on case statements. These unary operations are applied to each case partition $f_i$ $(1 \leq i \leq k)$ as follows:

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

### Binary operations

For the binary operations of cross-sum $\oplus$, cross-product $\otimes$ or cross-minus $\ominus$ on two case statements, the cross-product of the logical partitions of each case statement $\phi_i$ and $\psi_j$ is used to perform the corresponding operation:

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

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

$$\begin{cases} \phi_1: & f_1 \\ \phi_2: & f_2 \end{cases} \ominus \begin{cases} \psi_1: & g_1 \\ \psi_2: & g_2 \end{cases} = \begin{cases} \phi_1 \wedge \psi_1: & f_1 - g_1 \\ \phi_1 \wedge \psi_2: & f_1 - g_2 \\ \phi_2 \wedge \psi_1: & f_2 - g_1 \\ \phi_2 \wedge \psi_2: & f_2 - g_2 \end{cases}$$

Some partitions resulting from these operators may be inconsistent, that is $\phi_i \wedge \psi_j \models \bot$. In this case such a partition is discarded since it is irrelevant to the function value.

As explained in Section 3.1 for CA-HMDPs, the functions $f_i$ and $g_i$ are restricted to be linear or quadratic in the continuous parameters. While $\oplus$ and $\ominus$ preserve this property (i.e. remain closed-form), for $\otimes$ the result could exceed the second order polynomial if either $f_i$ or $g_i$ are quadratic. In such cases we may only assume the other function is piecewise constant. Conveniently as we show in the SDP algorithm presented next that the cross-product only occurs for the multiplication of piecewise constants in piecewise linear or quadratic functions. Therefore it is safe to claim that all the binary operations defined above are closed-form.

### Symbolic maximization

For SDP, we also need to perform maximization over actions for (6) which is fairly straightforward to define:

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

For CA-HMDPs, if all $f_i$ and $g_i$ are linear, the casemax result is clearly still linear. If the $f_i$ or $g_i$ are quadratic with univariate variables (i.e. no bilinear terms such as $x_1 x_2$), the expressions $f_i > g_i$ or $f_i \leq g_i$ will be at most univariate quadratic. Any such constraint can then be *linearized* into a combination of at most two linear inequalities (unless tautologous or inconsistent) by completing the square (e.g., $-x^2 + 20x - 96 > 0 \equiv [x - 10]^2 \leq 4 \equiv [x > 8] \wedge [x \leq 12]$). Hence the result of this casemax operator will be representable in the restricted case format assumed in this paper (i.e. linear inequalities in decisions).

The key observation here is the case statements are closed under the binary operation of continuous case maximization (or minimization). Additionally while it may seem that this operation leads to a blowup in the number of case partitions, the XADDs presented in the next section can exploit the internal structure of the continuous case maximization to represent it more compactly.

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 restriction $f|_\phi$ we want to restrict a function $f$ to apply only in cases that satisfy some formula $\phi$. By appending $\phi$ to each case partition we make sure that all partitions must satisfy $\phi$:

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

Since $f|_\phi$ only applies when $\phi$ holds and is undefined otherwise, $f|_\phi$ is partial unless $\phi \equiv \top$.

### Substitution

Symbolic substitution takes a set $\sigma$ of variables and their substitutions and applies it to function $f$. Consider $\sigma = \{x_1'/(x_1+x_2), x_2'/x_1^2\exp(x_2)\}$ , the left-hand side of / represents the substitution variable and the right-hand side of / represents the expression that should be substituted in its place. The substitution of a non-case function $f_i$ with $\sigma$ is $f_i\sigma$. If $f_i = x_1' + x_2'$ then using the $\sigma$ defined above $f_i\sigma = x_1 + x_2 + x_1^2\exp(x_2)$.

Substitution into case partitions $\phi_j$ is performed by applying $\sigma$ to each inequality operand. Consider the following example with the $\sigma$ definition of above:

$$f = \begin{cases} x_1' \leq \exp(x_2') : & x_1' + x_2' \\ x_1' \geq \exp(x_2') : & x_1'^2 - 3x_2' \end{cases}$$

$$f\sigma = \begin{cases} x_1 + x_2 \leq \exp(x_1^2\exp(x_2)) : & x_1 + x_2 + x_1^2\exp(x_2) \\ x_1 + x_2 \geq \exp(x_1^2\exp(x_2)) : & (x_1+x_2)^2 - 3x_1^2\exp(x_2) \end{cases}$$

Note that if $f$ has mutually exclusive partitions $\phi_i$ $(1 \leq i \leq k)$ then $f\sigma$ must also have mutually exclusive partitions. This is followed from the logical consequence that if $\phi_1 \wedge \phi_2 \models \bot$ then $\phi_1\sigma \wedge \phi_2\sigma \models \bot$. In general the substitution on case statements of function $f$ is defined below:

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

### Continuous Integration of the $\delta$-function

Continuous integration evaluates the integral marginalization $\int_{x_j=-\infty}^{\infty} f(x_j)\delta[x_j - h(\vec{z})]$ over the continuous variable $x_j$ in a function $f$ where $h(\vec{z})$ can be defined as a case statement and $\vec{z}$ does not contain $x_j$. . This integration triggers the substitution $\sigma = \{x_j/h(\vec{z})\}$ on $f$, that is

$$\int_{x_j=-\infty}^{\infty} f(x_j)\delta\left[x_j - \begin{cases} \phi_1 : & h_1 \\ \vdots & \vdots \\ \phi_k : & h_k \end{cases}\right] dx_j = \begin{cases} \phi_1 : & f\{x_j/h_1\} \\ \vdots & \vdots \\ \phi_k : & f\{x_j/h_k\} \end{cases}. \tag{7}$$

As an example consider the example $\delta\left[x_1' - (2x_1 + x_2)\right]$. Using the provided function definition, the continuous integration is defined according to the following:

$$f(x_1', x_1, x_2) = \begin{cases} x_1' \geq 5: & x_1' + x_2 \\ x_1' < 5: & x_1 \end{cases}$$

$$\int_{x_1'} f(x_1', x_1, x_2)\delta\left[x_1' - (2x_1 + x_2)\right] dx_1' = \begin{cases} 2x_1 + x_2 - 5 \geq 0: & 2x_1 + 2x_2 \\ 2x_1 + x_2 - 5 < 0: & x_1 \end{cases}$$

For a generic $\delta$ function with multiple case partitions, each of the $f$ partitions above are replaced into the multiple partitions of $\delta$. Such non-nested case statements are then reduced back down to a nested case statement using a simple approach:

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

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: [5]

$$\max_y f(\vec{b}, \vec{x}, y) = \max_y \text{casemax}_i \begin{cases} \phi_i(\vec{b}, \vec{x}, y): f_i(\vec{b}, \vec{x}, y) \\ \neg\phi_i(\vec{b}, \vec{x}, y): -\infty \end{cases}$$

$$= \text{casemax}_i \boxed{\max_y \begin{cases} \phi_i(\vec{b}, \vec{x}, y): f_i(\vec{b}, \vec{x}, y) \\ \neg\phi_i(\vec{b}, \vec{x}, y): -\infty \end{cases}} \tag{8}$$

The first equality is a consequence of the mutual disjointness of the partitions in $f$. Also because $\max_y$ and $\text{casemax}_i$ are commutative and may be reordered, we can compute $\max_y$ for *each case partition individually*. The case statement in the first equality is to ensure that all illegal values are mapped to $-\infty$. For computing the result of a given function $F_1$ with $-infty$, we have $\text{casemax}(F_1, -\infty) = F_1$. Thus this case statement is reduced to $\phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y)$ for computing the maximum.

Having handled the $-\infty$ case partition, we complete this section by showing how to symbolically compute a single partition $\max_y \phi_i(\vec{b}, \vec{x}, y) : f_i(\vec{b}, \vec{x}, y)$ represented in the second equality of Equation 8.

In $\phi_i$, we observe that each conjoined constraint serves one of three purposes:

- *upper bound (UB) on y*: can be written as $y < \cdots$ or $y \leq \cdots$

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

- *lower bound (LB) on y*: it can be written as $y > \cdots$ or $y \geq \cdots$ [6]
- *independent of y (Ind)*: the constraints do not contain $y$ and can be safely factored outside of the $\max_y$.

Since 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 $\vec{x}$.

Given the *potential* maximal 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 on all of the results. We further clarify this operation in the next section using the Inventory Control example.

### 3.2 Symbolic Dynamic Programming (SDP)

In this section the symbolic value iteration algorithm (SVI) for HMDPs is presented. 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

6. 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.

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

```
1  begin
2      V^0 := 0, h := 0
3      while h < H do
4          h := h + 1
5          foreach a(y⃗) ∈ A do
6              Q_a^h(y⃗) := Regress(V^{h-1}, a, y⃗)
7              //Continuous action parameter
8              if | y⃗ |> 0 then
9                  Q_a^h(y⃗) := max_{y⃗} Q_a^h(y⃗)
10                 π^{*,h} := arg max_a Q_a^h(y⃗)
11             else
12                 π^{*,h} := arg max_a Q_a^h(y⃗)
13
14         V^h := casemax Q_a^h(y⃗)
15         if V^h = V^{h-1} then
16             break  // Terminate if early convergence
17
18     return (V^h, π^{*,h})
19 end
```

horizon $h$ in the form of a case statement presented in Algorithm 1. We use the CAIC example from the introduction to help clarify each step of this algorithm.

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 it is not action dependent) is trivially in the form of a case statement.

For $h > 0$ and for each action in line 5 we perform lines 6–12, starting with a call to 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, the (`Regress`) algorithm is can be performed using the following steps:

1. *Prime the Value Function*: $V^h$ will be the "next state" in value iteration thus the substitution operation of $\sigma = \{b_1/b'_1, \ldots, b_n/b'_n, x_1/x'_1, \ldots, x_m/x'_m\}$ is used to obtain $V'^h = V^h\sigma$ in line 2 of Algorithm 2. Starting with the first iteration, for the CAIC example this step does not apply to $h - 1 = 0$ since $V^0 = 0$.

2. *Add Reward Function*: If the reward function $R$ contains any primed state variable $b'$ or $x'$, lines 3–4 of Algorithm 2 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 Algorithm 2 in lines 14–15. The reward function of CAIC contains primed $x'$ thus the Q-value is defined as below:

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

```
1  begin
2      Q = Prime(V)   // All v_i → v'_i (≡ all b_i → b'_i and all x_i → x'_i)
3      if v' in R then
4          Q := R(b⃗, b⃗', x⃗, x⃗', a, y⃗) ⊕ (γ · Q)
5
6      foreach v' in Q do
7          if v' = x'_j then
8              //Continuous marginal integration
9              Q := ∫ Q ⊗ P(x'_j | b⃗, b⃗', x⃗, x⃗', a, y⃗) d_{x'_j}
10         if v' = b'_i then
11             // Discrete marginal summation
12             Q := [Q ⊗ P(b'_i | b⃗, b⃗', x⃗, x⃗', a, y⃗)] |_{b'_i=1} ⊕ [Q ⊗ P(b'_i | b⃗, b⃗, x⃗, x⃗', a, y⃗)] |_{b'_i=0}
13
14     if ¬ (v' in R) then
15         Q := R(b⃗, b⃗', x⃗, x⃗', a, y⃗) ⊕ (γ · Q)
16
17     return Q
18 end
```

$$Q = \begin{cases} (x < 0 \vee x > 500 \vee x' < 0 \vee x' < 500) & : -\infty \\ d \wedge (150 \leq x \leq 500) & : 150 - 0.1 * a - 0.05 * x \\ d \wedge (0 \leq x \leq 150) & : 0.95 * x - 0.1 * a \\ \neg d \wedge (50 \leq x \leq 500) & : 50 + -0.1 * a + -0.05 * x \\ \neg d \wedge (0 \leq x \leq 50) & : 0.95 * x + -0.1 * a \end{cases}$$

3. *Continuous Integration*: The integral marginalization over continuous variables $\int_{\vec{x}'}$ is performed in lines 7–9. According to the integration definition of the previous section, this operation is performed repeatedly in sequence *for each* $x'_j$ $(1 \leq j \leq m)$ and for every action $a$:

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

Following the CAIC example, the continuous integration of $x$ results in the following:

$$Q = \begin{cases} x < 0 \vee x > 500 & : -\infty \\ d \wedge (x \geq 150) \wedge (150 \leq (x+a) \leq 650) & : 150 - 0.1 * a - 0.05 * x \\ d \wedge (x \geq 150) \wedge ((x+a \geq 650) \vee (x+a \leq 150)) & : -\infty \\ d \wedge (x \leq 150) \wedge (150 \leq (x+a) \leq 650) & : 0.95 * x - 0.1 * a \\ d \wedge (x \leq 150) \wedge ((x+a \geq 650) \vee (x+a \leq 150)) & : -\infty \\ \neg d \wedge (x \geq 50) \wedge (50 \leq (x+a) \leq 550) & : 50 - 0.1 * a - 0.05 * x \\ \neg d \wedge (x \geq 50) \wedge ((x+a \geq 550) \vee (x+a \leq 50)) & : -\infty \\ \neg d \wedge (x \leq 50) \wedge (50 \leq (x+a) \leq 550) & : 0.95 * x - 0.1 * a \\ \neg d \wedge (x \leq 50) \wedge ((x+a \geq 550) \vee (x+a \leq 50)) & : -\infty \end{cases} \quad (9)$$

4. *Discrete Marginalization*: Given the partial regression $Q^{h+1}$ for each action $a$ we next evaluate the discrete marginalization $\sum_{\vec{b'}}$ in (5) which is shown in lines 10–12 of Algorithm 2. Each variable $b'_i$ $(1 \leq i \leq n)$ is summed out independently using the restriction operation:

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

Note that both $Q_a^{h+1}$ and $P(b'_i | \cdots)$ can be represented as case statements. In CAIC discrete marginalization of the boolean state variable $d$ is not performed since there is no primed version of this variable $(d')$ in the current Q-function of (9).

Given the Q-function from Algorithm 2, next we take the maximum over the continuous parameter $y$ of action variable $a(\vec{y})$ in line 8–9 of Algorithm 1. Note that if the action were discrete $| \vec{y} | = 0$ , lines 8–10 are not performed.

We can 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:

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

Algorithm 3 outlines a univariate maximization $\max_y \phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y)$ according to the previous section. [7] Each step of this algorithm is followed using one of the partitions of the Q-function in ( 9) in this case the first partition with the constraints and function value defined below:

$$\phi_i(x, d, a) \equiv d \wedge (0 \leq x \leq 500) \wedge (x \geq 150) \wedge ((x+a) \leq 650) \wedge ((x+a) \geq 150)$$
$$f_i(x, d, a) = 150 - 0.1 * a - 0.05 * x$$

To begin the set of lower bound $LB$ is set to $-\infty$ and upper bound $UB$ to $\infty$ so that any value larger than $-\infty$ is defined as the lower bound and any value lower than $+\infty$ is

7. From here out we assume that all case partition conditions $\phi_i$ of $f$ consist of conjunctions of non-negated linear inequalities and possibly negated boolean variables — conditions easy to enforce 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.

**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=true , Case_max = ∅
3      for φ_i ∈ f (For all partitions of f) do
4          for c_i ∈ φ_i (For all conditions c of φ_i) do
5              if c_i ≤ y then
6                  LB := casemax(LB, c_i) //Add c_i to LB, take max of all LBs
7              if c_i ≥ y then
8                  UB := casemin(UB, c_i) //Add c_i to UB, take min of all UBs
9              else
10                 IND := [IND, c_i] //Add constraint c_i to independent constraint set
11
12
13     Root :=SolveForVar(y, f_i)
14     if (Root ≠ null) then
15         CONS = (𝕀[LB] ≤ 𝕀[Root]) ∧ (𝕀[Root] ≤ 𝕀[UB])
16     else
17         CONS := (𝕀[LB] ≤ 𝕀[UB])
18     //Conditions and value of continuous max for this partition
19     Max := IND ∧ CONS : casemax(f_i {y/LB} , f_i {y/UB} , f_i {y/Root})
20     //Take maximum of this partition and all other partitions
21     Case_max := max(Case_max, Max)
22     return Case_max
23 end
```

defined for $UB$. Constraint variables $IND$ and $CONS$ are assumed to be true and the result of the casemax is set to empty.

Each constraint $c_i$ in each partition $\phi_i$ is added to one of the sets of lower bound, upper bound or independent constraint as determined in lines 5–10. In our example this is equal to $LB = (150 - x, 0)$, $UB = (650 - x, 1000000)$ and $Ind = (d, x \geq 0, x \leq 500)$ where the 0 and 1000000 are the natural lower and upper bounds on any inventory item $x$. A unique $LB$ and $UB$ is defined by taking the maximum of the lower bounds and the minimum of the upper bounds as the best bounds in the current partition and the function *SolveForVar* of line 13 takes any roots of the partition function (not applicable in the current partition):

$$LB = \text{casemax}(0, 150 - x) = \begin{cases} x > 150: & 0 \\ x \leq 150: & 150 - x \end{cases}$$

$$UB = \text{casemin}(100000, 650 - x) = \begin{cases} x > -1000000: & 650 - x \\ x \leq -1000000: & 1000000 \end{cases}$$

The boundary constraints in lines 14–17 are added to the independent constraints as the constraint of the final maximum $Max$:

$$Cons_{LB \leq UB} = [0 \leq 1000000] \wedge [150 - x \leq 1000000] \wedge [150 - x \leq 650 - x] \wedge [0 \leq 650 - x]$$

Here, two constraints are tautologies and may be removed. A casemax is performed on the substituted $LB, UB$ and the roots on the function $f_i$:

$$Max = \text{casemax}\Big(f_i\{y/Root\} = null,$$
$$f_i\{y/LB\} = \begin{cases} x > 150 : 150 - 0.1 * (0) - 0.05 * x = 150 - 0.05 * x \\ x \leq 150 : 150 - 0.1 * (150 - x) - 0.05 * x = 135.00075 + 0.05 * x \end{cases},$$
$$f_i\{y/UB\} = \begin{cases} x > -1000000 : 150 - 0.1 * (650 - x) - 0.05 * x = 84.980494 + 0.05 * x \\ x \leq -1000000 : 150 - 0.1 * (1000000) - 0.05 * x = -99850 - 0.05 * x \end{cases}\Big)$$

$\max_y$ is performed in line 19 for each partition using both independent and boundary constraints. The resulting maximum is according to the casemax operator defined in the previous section. Note that the partition of $x \leq -1000000 : -99850 - 0.05 * x$ is omitted due to inconsistency.

$$Max = \begin{cases} (x > 150) \wedge (x \leq 650) : & 150 - 0.05 * x \\ (x > 150) \wedge (x \geq 650) : & 84.980494 + 0.05 * x \\ x \leq 150 : & 135.00075 + 0.05 * x \end{cases}$$

Returning to (8), we have now specified the inner operation (shown in the $\square$) for this partition after removing all redundant and inconsistent cases.[8]

$$Max = \begin{cases} d \wedge (150 \leq x \leq 500) : & 150 - 0.05 * x \\ \text{otherwise} : & -\infty \end{cases}$$

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 21:

$$Q = \begin{cases} d \wedge (150 \leq x \leq 500) : & 150 - 0.05 * x \\ d \wedge (0 \leq x \leq 150) : & -14.99925 + 1.05 * x \\ \neg d \wedge (50 \leq x \leq 500) : & 50 - 0.05 * x \\ \neg d \wedge (0 \leq x \leq 50) : & -5 + 1.05 * x \\ \text{otherwise} : & -\infty \end{cases}$$

To obtain the policy in Figure 1, we can annotate leaf values with any $UB$, $LB$, and $Root$ substitutions performing line 10 or 12 in Algorithm 1.

As the last step of Algorithm 1, a case maximization is performed on the current Q-function. Given $Q_a^{h+1}(\vec{y})$ 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 (6) requires sequentially applying *symbolic maximization* in line 14:

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

Note that for our CAIC example the last Q-function is equal to the optimal value function since we have considered a single continuous action here. By induction, because $V^0$ is a

8. These last two results are defined by taking out all inconsistent partitions. This is done using efficient pruning techniques mentioned in the previous section.

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 (5) or (6), 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 (5) or (6). 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. However as the previous section demonstrated, the final result of applying most operations such as binary operators and maximization is a larger case statement than the initial operands. Thus in practice it can be prohibitively expensive to maintain a tractable case representation. Maintaining a compact and efficient data structure for case statements is the key to SDP solutions.

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 XADD.

Here we formally introduce this compact data structure of XADDs which can implement case statements efficiently along with pruning algorithms required to perform SDP operations.

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 ($\{0,1\}^n \rightarrow \{0,1\}$) as an ordered DAG where each node represents a variable and edges represent boolean assignment to variables. Each decision node is a boolean test variable with two branches of low and high. The edge from the decision node to its low (high) successor represents assigning 0 (1) to its respective boolean variable. To evaluate the boolean function of a certain BDD under an assignment to its variables, the BDD is traversed by starting from the root node and following one of the low or high branches at each decision node until it reaches a leaf. The boolean value at the leaf is the value returned by this function according to the given variable assignment.

ADDs extend BDDs to allow real-valued ranges in the function representation ($\{0,1\}^n \rightarrow \mathbb{R}$); they only differ from BDDs in the real-valued leaf nodes. There is a set of efficient operations for ADDs such as addition ($\oplus$), subtraction ($\ominus$), multiplication ($\otimes$), division ($\oslash$), $\min(\cdot,\cdot)$ and $\max(\cdot,\cdot)$ ((Bahar et al., 1993)). Furthermore, BDDs and ADDs provide an efficient representation of CSI defined as the following:

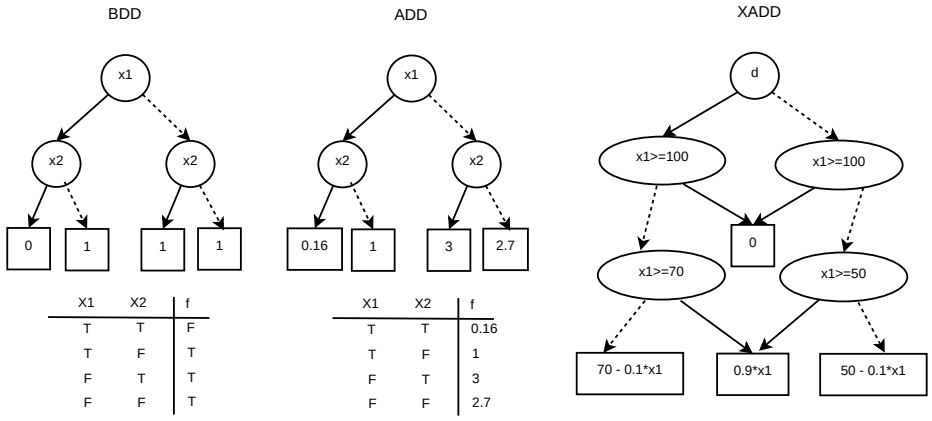



Figure 3: Comparison of three decision diagrams: *(left)* BDDs with boolean leaves and decisions representing $f(x_1, x_2) = x_1\bar{x_2} + \bar{x_1}$ as shown in the truth table; *(middle)* ADDs with boolean decision nodes and real values at the leaves; *(right)* XADDs with polynomial leaves and decision nodes.

**Definition** The set of variables $A$ and $B$ are *context-specifically independent* given the set of variables $C$ and the context $c \in C$, denoted by $A \perp B|(C = c)$, if the following holds:

$$P(A|B, C, c) = P(A|C, c) \;\; whenever \;\; P(B, C, c) > 0. \tag{10}$$

Parameterized ADD (PADD) is an extension of ADD that allows a compact representation of functions $\{0, 1\}^n \to \mathbb{E}$, where $\mathbb{E}$ is the space of expressions parameterized by $\vec{p}$. For example, the ADD of Figure 3 (middle) can be extended to a PADD to allow leaves such as $0.16p_1$ or $2.7 + p_1$. PADDs are used as an efficient representation of factored MDPs with imprecise transitions ((Delgado, Sanner, & de Barros, 2010)).

XADDs extend ADDs to allow representing functions with both boolean and continuous variables, i.e. $\{0, 1\}^m \times \mathbb{R}^n \to \mathbb{R}$. Here decisions can be boolean variable tests or continuous inequalities and leaves can be defined as arbitrary continuous expressions. Similar to its predecessors XADDs are evaluated from root to leaf. We next formally define such XADDs with operations and algorithms required to support all case operations of SDP. Pruning algorithms are then introduced to make this representation even more efficient. We note that the formal definitions of our XADD are similar to that of PADD.

### 4.1 Formal Definition and Operations

An XADD is a function represented by a DAG on $\mathbb{R}$ with the domain $(\vec{b}, \vec{x})$ where $b_i \in \{0, 1\}$ $(1 \leq i \leq m)$ is boolean and $x_j \in \left[x_j^{min}, x_j^{max}\right]$ is continuous with $1 \leq j \leq n; x_j^{min}, x_j^{max} \in \mathbb{R}; x_j^{min} < x_j^{max}$. According to Section 3.1, all functions can be represented using the case

notation, therefore a case statement can be used to represent an XADD:

$$f = \begin{cases} \phi_1(\vec{b}, \vec{x}) : & f_1(\vec{x}) \\ \vdots & \vdots \\ \phi_k(\vec{b}, \vec{x}) : & f_k(\vec{x}) \end{cases} \tag{11}$$

here the $f_i$ are arbitrary expressions over $\vec{x}$ and $\phi_i(\vec{b}, \vec{x})$ are logical formulae over $(\vec{b}, \vec{x})$ and defined by arbitrary logical combinations $(\wedge, \vee, \neg)$ of (a) boolean variables in $\vec{b}$ and (b) inequality relations $(\geq, >, \leq, <)$ over continuous variables $\vec{x}$.

In an XADD structure, each $f_i$ represents a leaf node and the $\phi_i(\vec{b}, \vec{x})$ represent all decision node constraints leading to the leaf $f_i$. Also similar to decision trees, the DAG representation has a fixed ordering of decision tests from the root to a leaf where the low (l) or high (h) branch of each decision node determines the next node in the XADD.

Similar to case statements that allow arbitrary functions, the above formulation represents a general XADD with arbitrary decision nodes and leaf expressions. However in this paper, we restrict our functions to linear decisions and linear leaf expressions. This allows us to prune inconsistent linear constraints using an LP-solver, resulting in a more compact XADD. Next we provide the mathematical definition of such *linear XADDs* in the following:

**Definition** (**Linear Leaf**): A linear XADD leaf can be canonically defined using the set of continuous variables $\vec{x}$ and the set of constants $c_i$ as the following:

$$\mathit{linearLeaf} := c_0 + \sum_i c_i x_i, 1 \leq i \leq n.$$

**Definition** (**Linear XADD**): Formally an XADD with linear decision and linear leaves is defined using the following BNF grammer:

$$\begin{array}{lll} F & ::= & \mathit{linearLeaf} \mid \text{if}(\mathit{dec}) \text{ then } F_h \text{ else } F_l \\ \mathit{dec} & ::= & (\mathit{linearLeaf} \leq 0) \mid (\mathit{linearLeaf} \geq 0) \mid b \\ \mathit{linearLeaf} & ::= & c_0 + \sum_i c_i x_i \\ b & ::= & 0 | 1. \end{array}$$

An XADD node $F$ is either a leaf node *linearLeaf* with a linear expression value or a decision node *dec* with two branches $F_h$ and $F_l$; both of the non-terminal type $F$.

The decision node *dec* can be a boolean decision test $b \in \{0, 1\}$ or a linear inequality over continuous variables $\vec{x}$. Figure 3 (right) represents an XADD consisting of leaf nodes such as node $0.9x$ or decision nodes such as nodes $b$ and $x \geq 100$. For our experiments, we also use univariate quadratic XADDs, that is XADDs with linear piecewise constraints (since the univariate quadratic constraints can be linearized) and univariate quadratic values. Thus the leaf node can also be in the form of $c_0 + c_1 x_j^2 + \sum_i c_i x_i (2 \leq i \leq n)$. All other definitions of a linear XADD can be applied to this XADD.

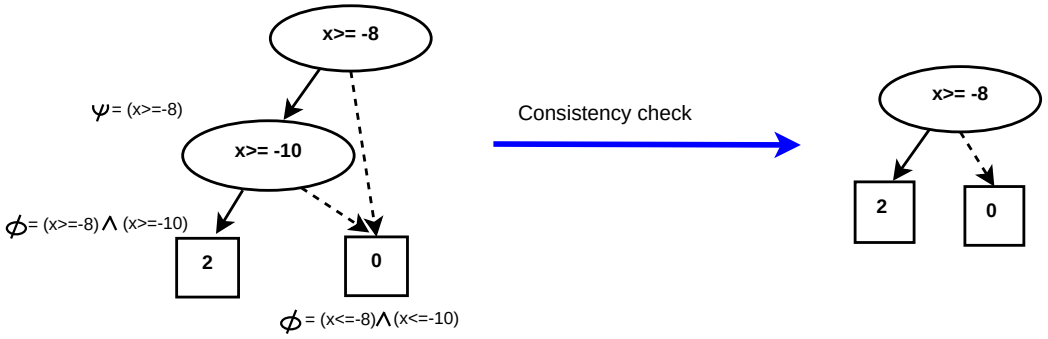



Figure 4: *(left)* Path definitions on each node in the XADD with an inconsistent path $(x \geq -8) \wedge (x \geq -10 \models \bot)$; *(right)* Pruned XADD after removing unreachable node $x \geq -10$.

For such an XADD, if branch $F_h$ is taken, $dec$ is true and if $F_l$ is taken the negation of the decision node, i.e. $\neg dec$, is set to true. Formally we evaluate the XADD function according to variable assignments to $dec$ as defined below: [9]

**Definition** (**XADD evaluation**): An XADD returns a real value given variable assignments $\rho \in \{\{0,1\}^m, \mathbb{R}^n\}$ to $(\vec{b}, \vec{x})$. Formally if $Val(F, \rho)$ is the value of XADD $F$ under the variable assignment $\rho$ then the full evaluation of $F$ is defined recursively as:

$$f = Val(F, \rho) = \begin{cases} linearLeaf, & \text{if } F = linearLeaf \\ Val(F_h, \rho), & \text{if } F = dec \wedge \ \ \rho(dec) = true \\ Val(F_l, \rho), & \text{if } F = dec \wedge \ \ \rho(dec) = false \end{cases}$$

Here $F = dec$ denotes that node $F$ is a decision node with the decision provided in $dec$. This recursive definition reflects the structural evaluation of $F$ starting at its root node and following the branch at each decision node corresponding to the decisions taken in $dec$. This process is continued until a leaf node is reached, which then returns $Val(F, \rho)$. As an example, in Figure 3 (right) an assignment of $\rho = \{1, 90\}$ to $(b, x)$ yields $Val(F, \rho) = 8.1$.

### 4.1.1 XADD Minimization

One issue with applying operations on the XADD (such operations are later introduced in Subsection 4.3) is the potential growth in the number of XADD nodes. Therefore after applying any operation on the XADD, it should be checked for any sources of infeasibility so that inconsistent branches are removed and not expanded in later stages. Furthermore, there may be redundant structures in the XADD producing extra nodes. Using pruning algorithms which are presented in the next section, we search for infeasibilities and redundancies in the XADD. Before that, we need to provide a set of definitions required to explain such algorithms.

9. Note we assume continuous linear inequality. Thus 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 $\leq(\geq)$.

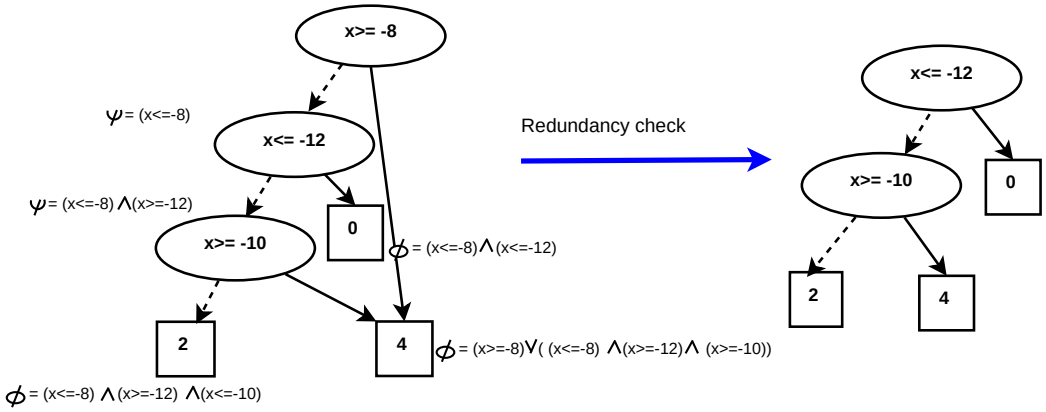



Figure 5: *(left)* Path definitions on each node in the XADD with the redundant nodes of $x >= -8$; since it is redundant in the paths leading to leaf node 4, that is: $(x \geq -8) \vee ((x \leq -8) \wedge (x \geq -12) \wedge (x \geq -10)) \equiv (x \geq -12) \wedge (x \geq -10)$;*(right)* The pruned XADD without the redundant node.

**Definition** (**Path**): A path in an XADD $F$ is defined as the conjunction of decision tests and their value from root to node $i$, that is $P_j^i = \{(dec_k, \rho(dec_k)\}$ where $j$ denotes the path number. The path leading to node $F_i$ is defined by the conjunction of all decision nodes $F_k$ (and their assignments) occurring before $F_i$ in XADD $F$. The formula for such a path $P_1^i$ is represented below:

$$\psi_j^i := \bigwedge_{k=1}^{i-1} \begin{cases} \rho(dec_k) = true: & dec_k \\ \rho(dec_k) = false: & \neg dec_k \end{cases}. \tag{12}$$

Recall the leaf node $F_i$ has the value $f_i$ in the case representation of Equation 11. We can now formally define a *case partition* $\langle \phi_i(\vec{b}, \vec{x}) : f_i(\vec{x}) \rangle$ using the path definition in 12:

**Definition** (**Case Partition**): The disjunction of all paths $\psi^i$ from root to leaf node $F_i$, denoted by $\phi^i$, is defined below:

$$\phi_j := \bigvee_{j=1}^{d} \psi_j^i \tag{13}$$

where $1 \leq j \leq d$ is the number of paths $\psi_j^i$ leading to leaf node $F_i$.

Figures 4 and 5 show two XADDs with all $\psi$ and $\phi$ paths before applying pruning techniques. According to the definitions of 12 and 13, we can now define an inconsistent path as the following:

**Definition** (**Inconsistent path**): A path $\psi_i$ in XADD $F$ is inconsistent if $\psi_i \models \bot$. An unreachable node is the final decision node on this inconsistent path by any assignment to $(\vec{b}, \vec{x})$. In the next section we describe how to prune such nodes from the XADD. As an example, Figure 4 (left) has an inconsistent path on which $x \geq -10$ violates the previous constraint of $x \geq -8$, that is: $(x \geq -8) \wedge (x \geq -10 \models \bot$. The XADD never reaches beyond $x \geq -10$, making it an unreachable node.

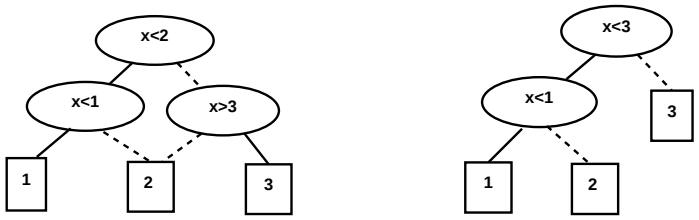



Figure 6: *(left)* A counterexample for an XADD that is not canonical after applying consistency and redundancy checking; *(right)* The true value of the true XADD which can not be derived from the left diagram.

Removing redundant nodes or branches from the XADD is a requirement for obtaining a compact XADD representation. Redundant low and high branches are easy to detect for any node and can be removed from an XADD using Algorithm 4 presented in the next section. A redundant node occurs in an XADD where two nodes give the same function definition and removing one does not change any of the partitions of that function. As an example consider the left diagram of Figure 5. Here removing node $x >= -8$ will not effect the function of final XADD as presented on the right side. The following provides a definition of redundancy in XADD nodes:

**Definition** (**Redundant node**): A node $F_r$ in $\phi_j^i$ is redundant if removing its decision constraint $dec_r$ in any of the paths containing $F_r$ does not change this case partition. Upon removing a decision constraint either the *true* or *false* branch is taken. Therefore a boolean assignment to $dec_r$ in $\phi_j^i$ (i.e. that is the assignment of $[\rho(dec_r) \rightarrow true]$ or $[\rho(dec_r) \rightarrow false]$) must equal the set of paths minus node $F_r$.

Consider Figure 5 (left) at the leaf node with value 4. The set of all paths for this node is equal to $\phi_4 = (x \geq -8) \vee ((x \leq -8) \wedge (x \geq -12) \wedge (x \geq -10))$ which is equal to $(x \geq -12) \wedge (x \geq -10)$; thus removing the redundant node of decision $(x \geq -8)$ results in the right-most figure.

It seems that using the two mentioned pruning techniques of inconsistency and redundancy checking removes all sources of infeasibility and therefore proof of canonicity is straightforward. Indeed for BDDs and ADDs proof of canonicity can be defined using the reduced diagrams ((Bryant, 1986)). As we defined reduced XADDs are derived from the result of the applying redundant branch pruning. Furthermore two pruning techniques of inconsistent path checking and redundant node pruning are applied to the final result of all XADDs. However unlike BDDs and ADDs a canonical XADD may not be produced after such pruning approaches. As a counterexample consider the simple XADD in the top left diagram of Figure 6 where the values as below:

$$\begin{cases} x \leq 1: & 1 \\ x \geq 3: & 3 \\ 1 < x < 3: & 2 \end{cases}$$

As the values suggest, there is no need to branch on $x < 2$ in this function, i.e. it can be removed from the tree. According to the implication checks in both pruning techniques, this

**Algorithm 4**: GetNode($dec, F_h, F_l$) $\longrightarrow \langle F_r \rangle$

```
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 ⟨dec, F_h, F_l⟩ → id is not in NodeCache then
7  |  |  id = new unallocated id
8  |  |  insert ⟨dec, F_h, F_l⟩ → id in NodeCache
9  |  return id
10 end
```

XADD is not inconsistent and further it can not be removed using the redundant technique which replaces a redundant node with one of its branches. To remove the node, a *reordering* of the decision nodes is required which will effect the structure of the tree, changing it to a new XADD. For this reason, although consistency and redundancy checking takes care of most unwanted branches, there is no guarantee that a given XADD is minimal after applying the two pruning techniques. The right diagram of Figure 6 is the true value this XADD holds which is a different XADD. Thus a challenging open problem is the proof of XADD minimality with respect to a reordering for linear decision nodes.

In the next section we use the definitions of inconsistency and redundancy to provide efficient algorithms to prune XADDs.

### 4.2 XADD Algorithms

Before defining our pruning algorithms, we first describe how a reduced XADD can be constructed from an arbitrary ordered decision diagram represented by $\{0,1\}^m \times \mathbb{R}^n \to \mathbb{R}$.

The helper function *getNode* in Algorithm 4 is used in all future algorithms. This algorithm returns a compact representation of a single internal decision node. It checks for branch redundancy so that no node has identical children on the high and low branch. If a redundant branch is found, it simply returns one of the children nodes in lines 3–4. Otherwise this function must build a new node with a unique id. Each new id is stored in a *NodeCache* table and lines 6–8 shows that before adding a new node, the cache is checked for an existing node with the same properties.

Algorithm *ReduceXADD* allows the construction of a compact XADD representation for a given XADD. This algorithm recursively constructs a reduced XADD from the bottom up. If the root node input to the algorithm is a leaf node, the algorithm returns the canonical form of $F$ in lines 4–5. Note that for a linear expression in the leaf, the canonical form is $c_0 + \sum_i c_i x_i, 0 \leq i \leq n$, however for arbitrary functions finding a canonical form may be time consuming or intractable.

If the root node input to the algorithm is a decision node then lines 7–13 are executed. A decision node is represented as $\langle dec, F_h, F_l \rangle$, where *dec* is the variable name, and $F_h$ and $F_l$ are the id for high and low branches respectively. The algorithm recursively computes the reduced XADDs of the low and high branch. It then uses the *GetNode* function in

**Algorithm 5**: ReduceXADD($F$) $\longrightarrow \langle F_r \rangle$

```
1  begin
2      //F is root node id for an arbitrary ordered decision diagram
3      //F_r is root node id for a reduced XADD
4      if F is terminal node then
5          return canonical terminal node for F
6      //else reduce decision node dec
7      if F → F_r is not in ReduceCache then
8          F_h = ReduceXADD(F_h)
9          F_l = ReduceXADD(F_l)
10         //get a canonical internal node id
11         F_r = GetNode(dec, F_h, F_l)
12         insert F → F_r in ReduceCache
13     return F_r
14 end
```

Algorithm 4 to remove any redundant branch. Reduced XADDs rooted at $F_r$ are then stored in the *ReduceCache* 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 according to the number of nodes in $F_r$.

Given a reduced XADD with no redundant branches, we further prune inconsistent paths and redundant nodes in the following sections.

#### 4.2.1 Inconsistency Pruning Algorithm

Given an XADD $F$ with potential inconsistent paths and the set of decision constraints on a path $\psi$, the output of *PruneInconsistent* (Algorithm 6) is a reduced XADD with canonical leaves, linear decisions and no unreachable nodes. Similar to Algorithm *ReduceXADD*, this algorithm is of a recursive bottom-up nature.

Lines 3–4 returns a canonical linear expression at the leaf if $F$ is a terminal node. Since $F$ does not require path consistency checking at a boolean decision node, lines 6–9 recursively calls inconsistency pruning on the low and high branches. The final result is returned using Algorithm 4.

If the two mentioned conditions are false, then the current node of $F$ is a linear decision node. Here we check if the decision constraint of this node (*dec*) is consistent with the previous constraints on path $\psi$. Lines 11 and 13 use the *TestImplied* function which performs the following steps. The input to this algorithm is the path $\psi$ and current decision node constraint *dec*.

- Check if the implication result of $\psi$ exists in an *Implications* cache. This cache stores paths $\psi$ that are inconsistent. If there is a previous result which already contains the new decision constraint *dec* then return *true*.
- Similarly check the *non-Implications* cache for consistent path $\psi$. If there is a previous result which already contains the new decision constraint *dec* then return *false*.

**Algorithm 6**: PruneInconsistent($F$, $\psi$,*testRedundant*) $\longrightarrow \langle F_r \rangle$

```
1  begin
2      //F is the root node represented as (dec, F_l, F_r)
3      if F is terminal node then
4          return canonical terminal node for F
5      //if F is a boolean decision, no inconsistency checking possible for F
6      if F ∈ {0,1}^m then
7          low = PruneInconsistent(F_l, ψ)
8          high = PruneInconsistent(F_h, ψ)
9          return GetNode(dec, high, low)
10     //else F is a linear decision check if ψ ⊨⊥
11     if TestImplied(ψ,dec) then
12         return PruneInconsistent(F_h, ψ)
13     if TestImplied(ψ,¬dec) then
14         return PruneInconsistent(F_l, ψ)
15     //result of TestImplied was false
16     ψ ⟵ [ψ, ¬dec]
17     low = PruneInconsistent(F_l, ψ)
18     ψ ⟵ ψ \ ¬dec
19     ψ ⟵ [ψ, dec]
20     high = PruneInconsistent(F_h, ψ)
21     ψ ⟵ ψ \ dec
22     if testRedundant then
23         //check if high branch is implied in the low branch
24         ψ ⟵ [ψ, dec]
25         isRedundant =IsRedundant(ψ,low, high)
26         ψ ⟵ ψ \ dec
27         if isRedundant then
28             return low
29         //is low branch implied in the high branch for the negation of dec
30         ψ ⟵ [ψ, ¬dec]
31         isRedundant =IsRedundant(ψ,high, low)
32         ψ ⟵ ψ \ ¬dec
33         if isRedundant then
34             return high
35
36
37     return F_r = GetNode(dec, high, low)
38 end
```

- If there is no cache hit, then the negation of *dec* is added to the path: $\psi \longleftarrow [\psi, \neg dec]$. This allows us to check for the infeasibility property.

- The LP-solver is called on the set of decision constraints in $\psi$. The result of the LP-solver determines infeasibility with respect to the new decision added to $\psi$, which is implementing inconsistency checking.
- The path $\psi$ is stored in its related cache according to the result of the LP-solver (i.e. *true* or *false*). If the result is inconsistent then $\psi$ is added to *Implications*, else it is added to *non-Implications*.
- The negation of *dec* is removed from the path: $\psi \longleftarrow \psi \setminus \neg dec$.
- The result of the LP-solver is returned as the output of this function.

Line 11 checks for the high branch of $F$ that is adding the true assignment of *dec*. If *true* is returned from *TestImplied*, then the high branch is inconsistent. As a result this algorithm returns the child at the high branch as the result of this pruning in line 12. Similarly in line 13 the low branch is checked with $\neg dec$ and if *true* is returned then the low branch is inconsistent and line 14 is returned.

In case the current decision node is not on an inconsistent path, both low and high branches need to be checked for inconsistency. Lines 17–19 add the decision constraint for the low branch $\neg dec$ to $\psi$ and call *PruneInconsistent* for $F_l$ and removes this constraint after this call. Lines 20–22 perform the same for the high branch ($F_h$).

At this stage the current subtree with the computed low and high branches does not contain any inconsistent paths. Thus if the boolean indicator to check for redundancy *testRedundant* is set to *false* then this algorithm returns a reduced node computed using *GetNode* in line 37. However as we explain in the next section, for full pruning (i.e. *testRedundant* = *true*) all redundant nodes must be pruned.

#### 4.2.2 Redundancy Pruning Algorithm

Lines 22–34 of Algorithm 6 removes any redundant nodes from the subtree rooted at $\langle dec, low, high \rangle$ where *low* and *high* are the subtrees after inconsistent pruning. To check for redundancy, the decision constraint of the current subtree is added to $\psi$ and a call is made to Algorithm 7. Intuitively *IsRedundant* checks to see whether a *subtree* can replace another subtree *goal* without changing the XADD function.

Initially lines 3–4 check if *subtree* and *goal* are terminal nodes and equal, in this case the *goal* is redundant and can be replaced by *subtree*, therefore the algorithm returns *true*. If the nodes are non-equal terminal nodes then *goal* is not redundant and line 28 returns *false*. Next lines 6–9 check if both nodes are decision nodes but the *goal* node appears before the *subtree* in the variable ordering of the XADD, in which case the *goal* subtree can not be redundant.

If neither of the above cases occur, the algorithm must check whether $subtree \models goal$ using the *TestImplied* function. Lines 11–15 call *TestImplied* for the decision constraint of the root node of *subtree*. If *TestImplied* returns *true* for the false value of the decision node (i.e. $subtree_{dec}$), then the algorithm returns the redundancy check for the low child of the *subtree*. For the true value of $subtree_{dec}$ a recursive call to *IsRedundant* is performed with the high child. This process ensures that the entire subtree is traversed and all children of the subtree also imply the *goal*.

**Algorithm 7**: IsRedundant($\psi$, *subtree*, *goal*) $\longrightarrow \langle F_r \rangle$

```
1  begin
2      //both branches are terminal nodes, redundant if high=low
3      if subtree = goal then
4          return true
5      if subtree is non-terminal node then
6          if goal is non-terminal node then
7              //node is not redundant if the goal occurs before subtree
8              if subtree ≥ goal then
9                  return false
10
11         // check if subtree ⊨ goal
12         if TestImplied(ψ,¬subtree_dec) then
13             return IsRedundant(ψ,subtree_l ,goal)
14         if TestImplied(ψ,subtree_dec) then
15             return IsRedundant(ψ,subtree_h ,goal)
16
17         //result of TestImplied was false
18         ψ ⟵ [ψ, ¬subtree_dec]
19         isImplied = IsRedundant(ψ,subtree_l ,goal)
20         ψ ⟵ ψ \ ¬subtree_dec
21         //if the low branch does not imply goal, the node is not redundant
22         if ¬isImplied then
23             return false
24         ψ ⟵ [ψ, subtree_dec]
25         isImplied = IsRedundant(ψ,subtree_h ,goal)
26         ψ ⟵ ψ \ subtree_dec
27         return isImplied
28     return false
29 end
```

Similar to Algorithm 6, if the result of *TestImplied* is false, both low and high branches need to be checked for redundancy. Lines 18–20 adds the decision constraint for the low branch $\neg subtree_{dec}$ to $\psi$ and calls *IsRedundant* for $subtree_l$ and removes this constraint after this call. If any of the children along the way do not imply *goal*, then *goal* can not be considered redundant as line 22–23 illustrates. Lines 24–27 perform the same for the high branch $subtree_h$ and returns the result of *IsRedundant*.

Returning back to Algorithm 6, line 26 removes the current decision constraint from $\psi$ and checks for redundant nodes returned from Algorithm 7. If the algorithm has returned *true* then the low branch is sufficient to represent this subtree, as the node in the high branch is redundant. Figure 5(right) shows that the node $x \geq -8$ in the left figure is redundant and can be removed from the diagram.

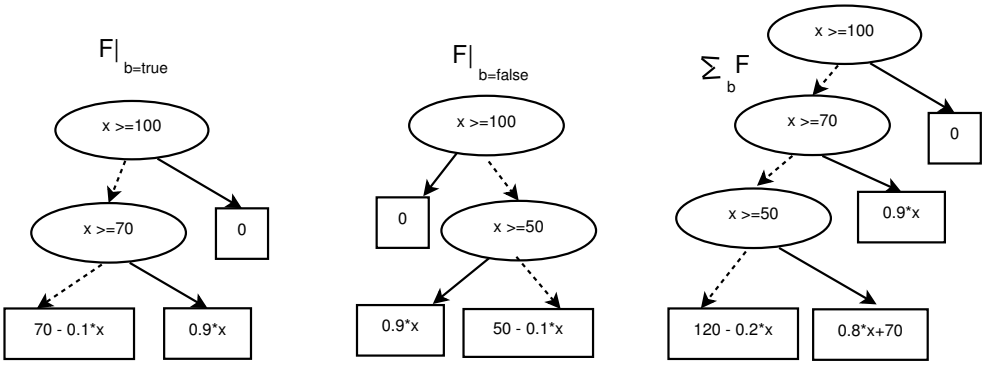



Figure 7: *(left)* Example of restriction operation $F|_{b=true}$ and $F|_{b=false}$ on the XADD of Figure 3(right) for the binary variable $b$. *(right)* The resulting XADD after marginalization $\sum_b F$.

**Algorithm 8**: Reorder$(F) \longrightarrow \langle F_r \rangle$

```
1  begin
2      if F is terminal node then
3          return canonical terminal node of F
4      if F → F_r is not in ReorderCache then
5          F_true = Reorder(F_true) ⊗ 𝕀[dec]
6          F_false = Reorder(F_false) ⊗ 𝕀[¬dec]
7          F_r = F_true ⊕ F_false
8          insert F → F_r in ReorderCache
9      return F_r
10 end
```

A similar approach is then performed in lines 29–34 to check if the *low* branch is implied in the *high* branch given the decision constraint is *false*. In this case the *high* branch is returned instead of the node rooted at *dec*. This concludes the pruning algorithm for removing any redundant node from the XADD $F$.

Next we present the main XADD operations required for SDP algorithms.

### 4.3 XADD Operations

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

#### 4.3.1 Unary XADD Operations

According to the previous section on unary operations required in SDP, scalar multiplication $c.f$ and negation $-f$ on a function can simply be represented as an XADD. Negation of XADD $F$ is equal to performing a binary operation of $0 \ominus F$ explained in the next

**Algorithm 9**: ReduceLinearize($F$)$\longrightarrow \langle F_r \rangle$

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

section. Furthermore restriction, substitution and integration of the $\delta$ function are also unary operations that can be applied to XADDs are explained below.

Restriction of a variable $b$ in $F$ to some formula $\phi$ is performed by appending $\phi$ to each of the decision nodes (using logical $\wedge$) while leaves are not affected. For a binary variable, restriction is equal to taking the true or false branch that is $(F|_{b_i=true})$ or $(F|_{b_i=false})$. This operation can also be used for *marginalizing* boolean variables; $\sum_{b_i}$ eliminates variable $b_i$ from $F$ by computing the sum of functions restricted to *true* or *false* value of $b_i$, i.e. $F|_{b_i=true} \oplus F|_{b_i=false}$. The mentioned restriction operation uses $\oplus$ which is a binary operation handled in Algorithm 11. Figure 7 demonstrates restriction of $F|_b$ and the right diagram shows the marginalization of $\sum_b F$.

Marginalizing a continuous variable $x$ is performed using the integration of the $\delta$-function on variable $x$. Specifically in SDP we require computing $\int_x \delta[x - g(\vec{x})] f dx$ which triggers the substitution $f\{x/g(\vec{x})\}$ on $f$ as defined next. Note that the XADD representation is used for both functions $f$ and $g$.

Substitution for a given XADD $F$ is performed by applying the set of variables and their substitutions to each case statement 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 may become unordered when substituted. A reorder algorithm has to be applied to the result of the substitution operand. As Algorithm 8 shows, we recursively apply the binary operations of $\otimes$ and $\oplus$ to decision nodes for reordering the XADD after a substitution. Similar to other XADD algorithm a *ReorderCache* is used to prevent unnecessary reordering.

The other operation required for CA-HMDPs is a continuous maximization over continuous action parameters. As we see in the next section this maximization can be performed symbolically but often produces non-linear inequalities at the decision nodes. This does not effect our symbolic solution theoretically but in practice to prune the resulting XADD using an LP-solver, we require linear decisions.

The linearize algorithm (in Algorithm 9) is of a recursive nature similar to other XADD algorithms with a *LinearCache* to avoid linearizing the same node. For each node in the XADD starting at the root node, the algorithm linearizes the decision node using *GetRoots*

**Algorithm 10**: ChooseEarliestDec$(F_1, F_2) \longrightarrow \langle dec \rangle$

```
1  begin
2  |  //select the decision to branch based on the order
3  |  if F_1 is a non-terminal node then
4  |  |  if F_2 is a non-terminal node then
5  |  |  |  if dec_1 comes before dec_2 then
6  |  |  |  |  dec = dec_1
7  |  |  |  else
8  |  |  |  |  dec = dec_2
9  |  |  |
10 |  |  else
11 |  |  |  dec = dec_1
12 |  |
13 |  else
14 |  |  dec = dec_2
15 |  return dec
16 end
```

to find the roots of the non-linear function. The algorithm then returns the linear decision nodes. By iterating on the low and high branches of the decision node, all nodes of $F$ are traversed.

#### 4.3.2 Binary XADD Operations

For *all* binary operations, the function $Apply(F_1, F_2, op)$ (Algorithm 11) computes the resulting XADD. Two reduced 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 . The output result is a reduced XADD after applying the binary operation.

Briefly if the result of this algorithm is a quick computation of Table *ComputeResultXADD* it can be immediately returned. Otherwise it checks the *ApplyCache* for any previously stored apply result. If there is not a cache hit, the earliest decision in the ordering to branch is chosen according to Algorithm 10. Two recursive *Apply* calls are then made on the branches of this decision to compute $F_l$ and $F_h$. Finally *GetNode* checks for any redundancy before storing it in the cache and returning the resulting XADD.

Specifically this algorithm can be described using the following steps:

**Terminal computation**: The function *ComputeResult* in line 3 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 discrete maximization (minimization) operation (entries 11–16) , 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 reordered by Algorithm 8.

**Caching**: If the result of *ComputeResult* is empty, in the next step we check the *ApplyCache* of line 6 for any previously computed operation using this set of operands and

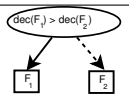
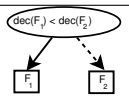
| Case number | Case operation | Return |
|---|---|---|
| 1 | $F_1\ op\ F_2; F_1 = Poly_1; F_2 = Poly_2$ | $Poly_1\ op\ Poly_2$ |
| 2 | $F_1 \oplus F_2; F_2 = 0$ | $F_1$ |
| 3 | $F_1 \oplus F_2; F_1 = 0$ | $F_2$ |
| 4 | $F_1 \ominus F_2; F_2 = 0$ | $F_1$ |
| 5 | $F_1 \otimes F_2; F_2 = 1$ | $F_1$ |
| 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 | $F_1 \oplus \infty$ | $\infty$ |
| 10 | $F_1 \otimes \infty; F_1 \neq 0$ | $\infty$ |
| 11 | $\max(F_1, +\infty)$ | $\infty$ |
| 12 | $\max(F_1, -\infty)$ | $F_1$ |
| 13 | $\min(F_1, +\infty)$ | $F_1$ |
| 14 | $\min(F_1, -\infty)$ | $-\infty$ |
| 15 | $\max(F_1, F_2)$ |   |
| 16 | $\min(F_1, F_2)$ |   |
| 17 | other | *null* |

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

operations. To increase the chance of a match, all items stored in a cache are reduced XADDs.

**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. If both operands are constant terminal nodes the function *ComputeResult* takes care of the result and for the rest of the cases one of the following conditions applies:

- $F_1$ or $F_2$ is a constant terminal node or $dec_1 \neq dec_2$: The high and low branch of the operand chosen by *ChooseEarliestDec* (here $F_2$) is applied with the other operand $F_1$. The result of these two *Apply* functions are used with the constraint decision node of $dec_2$ for a reduced result:

$$F_h = Apply(F_1, F_{2,h}, op)$$
$$F_l = Apply(F_1, F_{2,l}, op)$$
$$F_r = GetNode(dec_2, F_h, F_l)$$

- $F_1$ and $F_2$ are constant nodes and $dec_1 = dec_2$: Since the decision constraints are equal, the final result is a decision with the constraint of $dec_1 (= dec_2)$ and the high

**Algorithm 11**: Apply$(F_1, F_2, op) \longrightarrow \langle F_r \rangle$

```
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      //check if we previously computed the same operation
6      if ⟨F_1, F_2, op⟩ → F_r is not in ApplyCache then
7          //choose decision to branch
8          dec = ChooseEarliestDec(F_1, F_2)
9          //set up nodes for recursion
10         if F_1 is non-terminal ∧ dec = dec_1 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 ∧ dec = dec_2 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(dec, F_h, F_l)
24         //Use Algorithm 8 to reorder decisions
25         F_r = Reorder(F_r)
26         //save the result to reuse in the future
27         insert ⟨F_1, F_2, op⟩ → F_r into ApplyCache
28     return F_r
29 end
```

branch is calling *Apply* on the high branches of $F_1$ and $F_2$ and the low branch is defined as the result of the *Apply* function on the low branches:

$$F_h = Apply(F_{1,h}, F_{2,h}, op)$$
$$F_l = Apply(F_{1,l}, F_{2,l}, op)$$
$$F_r = GetNode(dec_1, F_h, F_l)$$

Finally for the high and low branch of the final computation, two recursive calls are made to *Apply* and the result is a reduced XADD returned by *GetNode* where the decisions are ordered by *Reorder*.

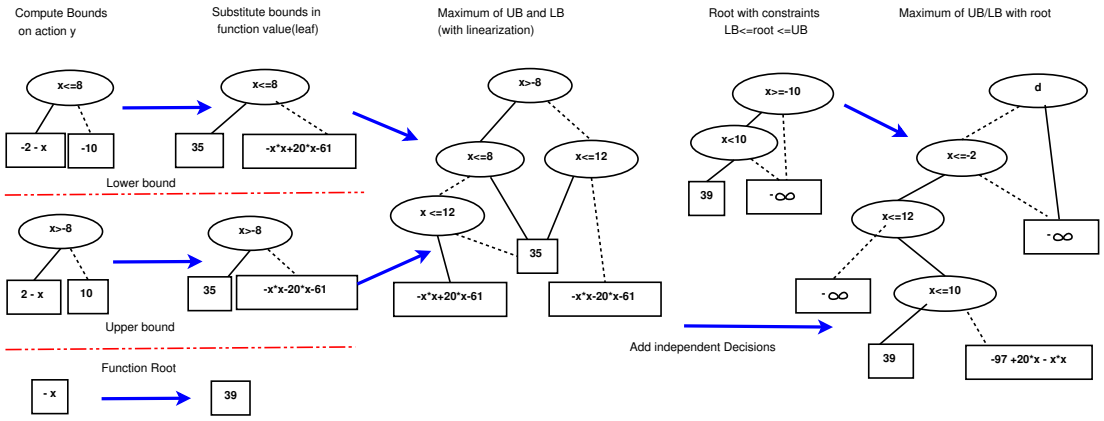



Figure 8: One step of Continuous Maximization algorithm using XADDs for partition: $\phi_i(x, d, y) \equiv \neg d \wedge (x > 2) \wedge (-10 < y < 10) \wedge (-2 < x + y < 2)$ and value $f_i(x, y) = 39 - y^2 - x^2 - 2a \times x$.

### 4.4 XADD representation for continuous maximization

The operation of continuous maximization can also be defined in this structure. Each XADD path from root to leaf node is treated as a single case partition with conjunctive constraints. $\max_y$ is performed at each leaf subject to these constraints and all path $\max_y$'s are then accumulated via the casemax operation to obtain the final result.

A single step of this algorithm is presented in Figure 8. This example is more complex compared to the CAIC partition mentioned above since it has to consider the function root XADD as well as the upper and lower bounds. The input to this illustrated step is the leaf node obtained after the regression step of Algorithm 2 and decisions leading to this leaf (high or low branches). The XADD representation is used to represent all intermediate results and also the final result.

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 with the discrete setting and one with the continuous setting.

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. [10]

10. 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

**Inventory Control** The inventory problem mentioned in the introduction is revisited to compare 1-item, 2-item and 3-item inventories for both deterministic and stochastic customer demands.

**Mars Rover** 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:

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

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

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

The maximum long-term *value* $V$ from a given state in Mars Rover is defined 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 contains non-rectangular decision boundaries like $4 - x^2 - y^2 \geq 0$. Figure 10 presents the 0-, 1-, and 2-step time horizon solution for this problem. 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 electricity generated in periods when the $drain(e)$ action drains water from $l_2$ to $l_1$ (the other action is $no\text{-}drain(e)$); we assume deterministic rainfall replenishment and present the reward function as:

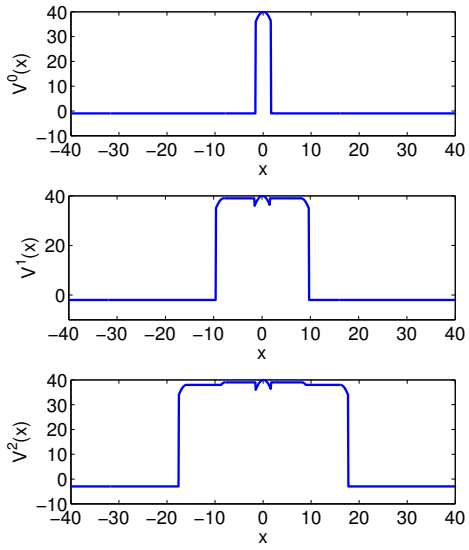



Figure 9: Optimal sum of rewards (value) $V^t(x)$ for $b = 0$ (*false*) for time horizons (i.e., decision stages remaining) $t = 0$, $t = 1$, and $t = 2$ on the Continuous Action Mars Rover problem. For $x \in [-2, 2]$, the rover automatically takes a picture and receives a reward quadratic in $x$. We 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 $x = \pm 22$.

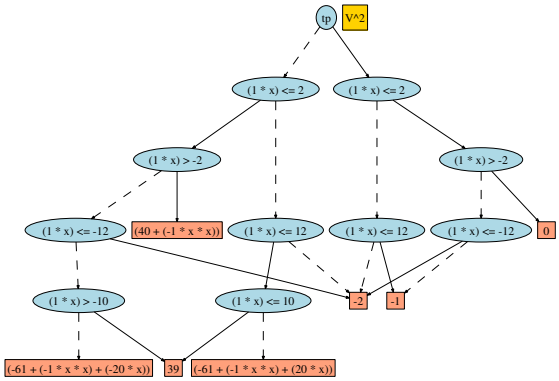



Figure 10: Optimal value function $V^2(x)$ for the Continuous Action Mars Rover problem represented as an (XADD) equal to the second diagram on the left. To evaluate $V^2(x)$ for any state $x$, one simply traverses the diagram in a decision-tree like fashion until a leaf is reached where the non-parenthetical expression provides the *optimal value* and the parenthetical expression provides the *optimal policy* ($y = \pi^{*,2}(x)$) to achieve value $V^2(x)$.

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

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 $i$ number of slices. For example the Mars Rover problem with 2 actions which are the lower and upper bounds on $a$ ($a_1 = -10, a_2 = 10$) and the transition and reward functions are defined according to these constant values. We now provide the empirical results obtained from implementing our algorithms.

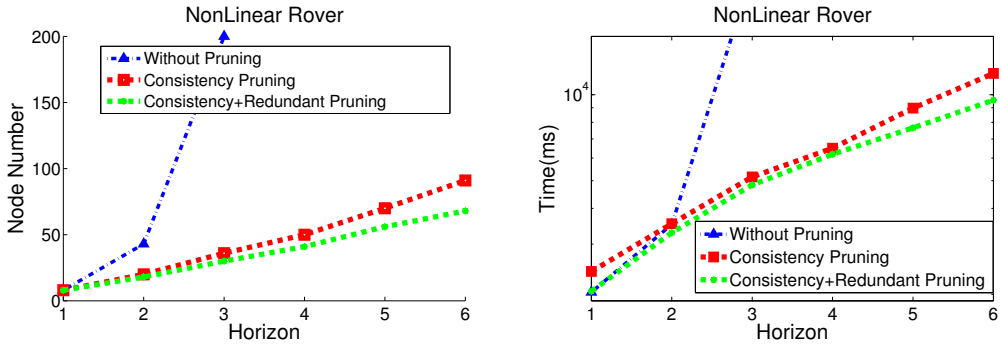



Figure 11: Space (# XADD nodes in value function) and time for different iterations (horizons) of SDP on Nonlinear CONTINUOUS ACTION MARS ROVER with 4 different results based on pruning techniques. Results are shown for the XADD with no pruning technique, with only consistency checking (using LP-solver) and with both the consistency and redundancy checking with a numerical precision heuristic.

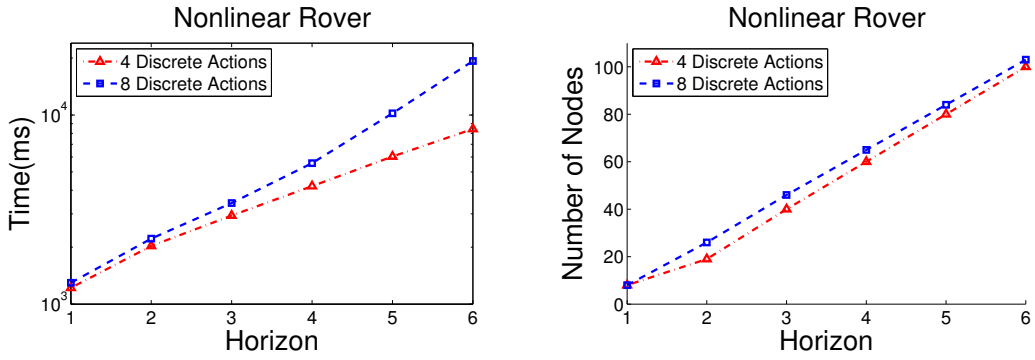



Figure 12: Space and elapsed time vs. horizon for a fixed discretization of 4 and 8 for the MARS ROVER domain. This demonstrates how the number of nodes and time increases for each horizon.

### 5.2 Results

For both the DISCRETE ACTION and CONTINUOUS 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.

For the CONTINUOUS ACTION MARS ROVER problem, we present the time and space analysis in Figure 11. Here three evaluations are performed based on the pruning algorithms of Section 4.2. We note that without the inconsistency checking of Algorithm 6, SDP can not go beyond the third iteration as it produces many inconsistent nodes. The comparison is performed with consistency pruning and redundancy checking. The full pruning experiment also uses a numerical heuristic that omits similar branches. However even with the heuristic approach very little node reduction can be gained from redundancy pruning. This suggests

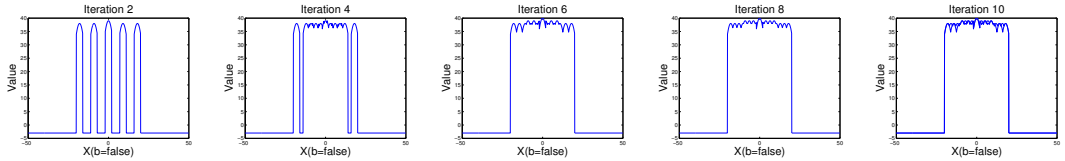



Figure 13: $V^3$ for different number of Discrete Actions in the Mars Rover problem. As the number of discrete actions increases, the value function more closely resembles the continuous action value of $V^3$ (defined in Figure 10).

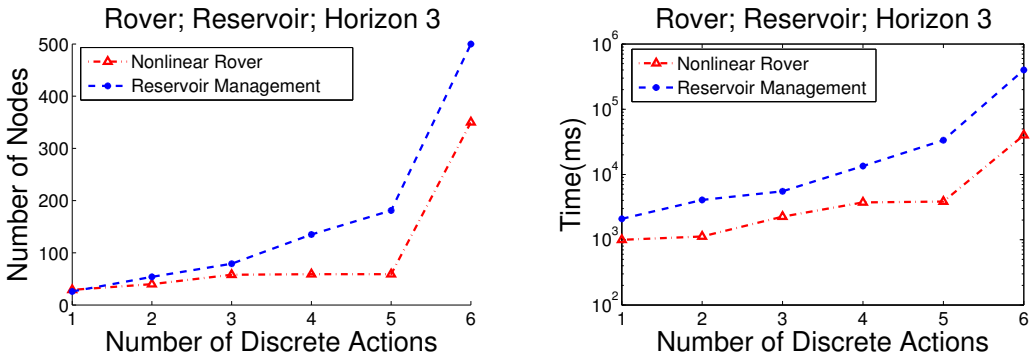



Figure 14: Space and elapsed time vs. horizon for a fixed horizon of 3 and different problem sizes for the Discrete Action Mars Rover and the Reservoir Management .

that in some domains using redundancy pruning in not efficient and should be omitted from the final results. Hence we will present results for the Reservoir Management and Inventory Control problem with only consistency pruning.

Next we present the analysis of the Discrete Action Mars Rover domains. Figure 12 shows how time and space costs of different horizons increases for a fixed action discretization of 4 and 8. Note that one of the caveats of using a discrete setting is defining the actual discrete actions. In the continuous setting an action is defined between a large high and low range (e.g $a \in [-1000000, 1000000]$) allowing the SDP algorithm to choose the best possible action among all the answers. However for a discrete setting, choosing the range to discretize the action becomes very important. As an example in the rover description, allowing actions to be far from the center (e.g. $a \in [-20, 20]$) does not result in a converged solution. Finding this range is one of the drawbacks of using a discrete setting.

On the other hand, the level of discretization within this range is also very important. To show this visually Figure 13 shows the results of the third iteration for 6 different discretizations compared to the continuous result. This figure proves the need to finely partition the action space within the predefined range. However as Figure 14 demonstrates, increasing the number of discrete action (for the fixed horizon of 3), leads to increasing time and space costs. Here results of different level of discretization is similar for the Reservoir Management problem.

Figure 15 presents the time and number of nodes for different horizons for 4 and 8 discrete actions compared to the continuous actions in the Reservoir Management domain.

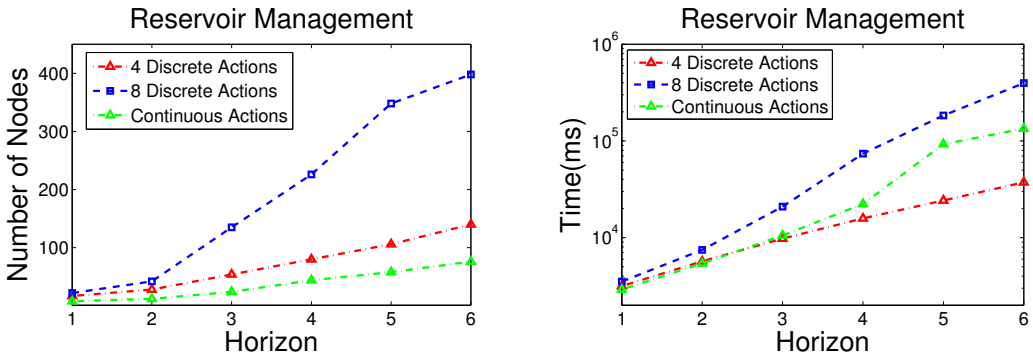



Figure 15: Space and elapsed time vs. horizon for a fixed discritization of 4 and 8 for the discrete action RESERVOIR MANAGEMENT . To compare the time and space of continuous action RESERVOIR MANAGEMENT is given.

Although in the first two iterations the time and space of the discrete action setting are similar, however for higher horizons more time and space is required in the 8 discrete action case. The number of nodes are lower for the continuous case compared to both discretizations but the time elapsed is higher than the 4-discrete actions due to the complexity of the continuous action maximization.

Figure 16 (left) demonstrates three levels of discretization in the RESERVOIR MANAGEMENT problem. The top left figure assumes 4 discrete actions, the middle figure has 7 actions and the bottom figure uses 10 discrete actions. The value of the third iteration is represented for all figures w.r.t. water levels $l_1$ and $l_2$. The figures suggest that finer grain discretization results in better results, closer to that of the continuous action value in middle right figure.

Furthermore Figure 16 (right) plots the the optimal closed-form policy at $h = 3$: the solution interleaves *drain*$(e)$ and *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). $V^3(l_1, l_2)$ and $V^6(l_1, l_2)$ show the progression of convergence from horizon $h = 3$ to $h = 6$ — low levels of $l_1$ and $l_2$ allow the system to generate electricity for the longest total elapsed time over 6 decision stages.

In Figure 17, 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. 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.

Figure 18 represents the time and space for the deterministic DISCRETE ACTION INVENTORY CONTROL for a discretization of 6 actions and different inventory items for up to $h = 6$ horizons. While the number of items affects both time and space, even for 6 discrete actions the 3-item inventory will have exponential time and space for the second horizon

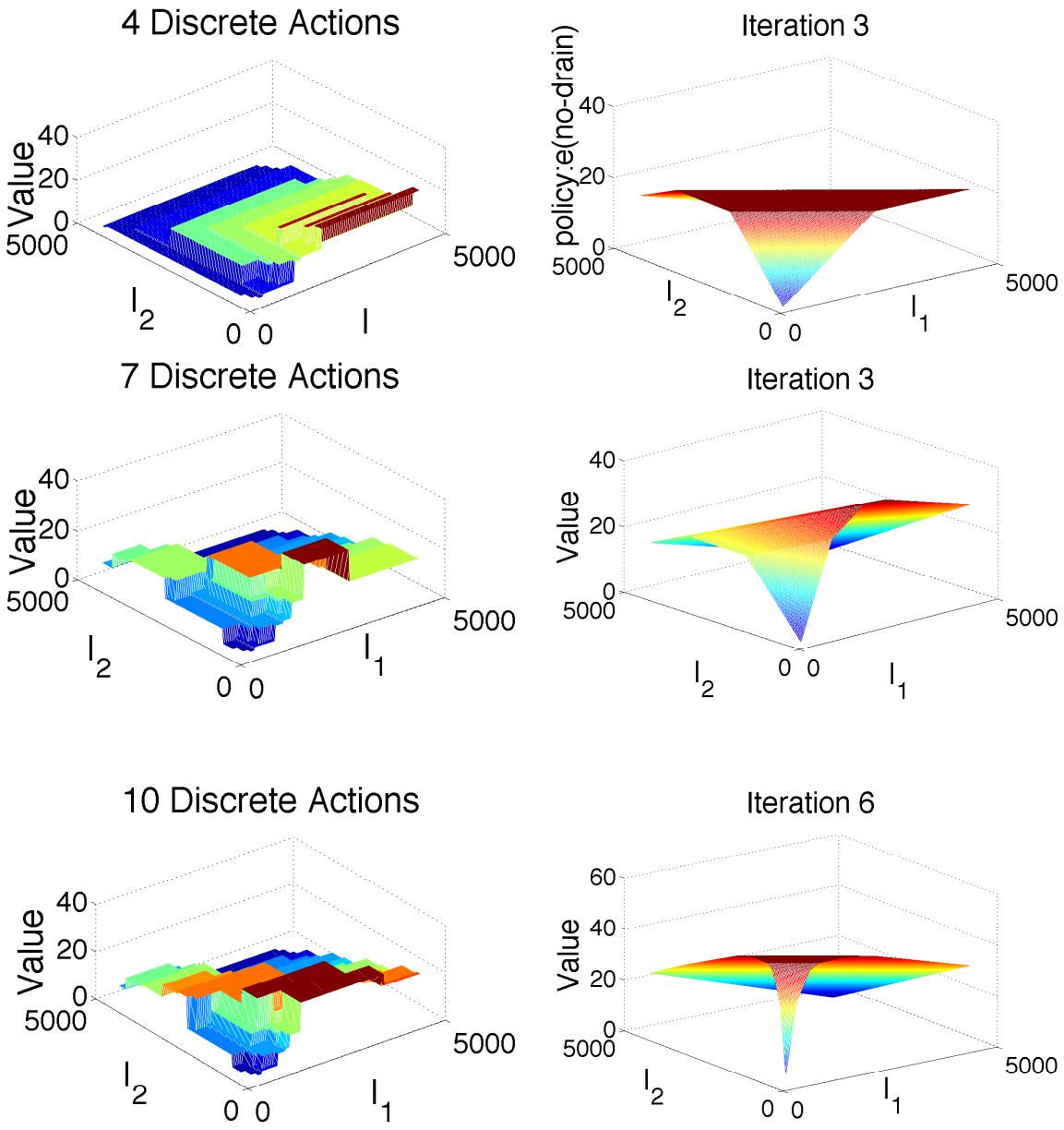



Figure 16: Results for the Discrete Action and Continuous Action of the Reservoir Management problem. *(left)* 4, 7 and 10 Discrete Actions of the Reservoir Management problem for different water levels in iteration $V^3$. Each discretization draws the value closer to that of the continuous action Reservoir Management on the right. *(right)* Policy *no-drain*$(e) = \pi^{3,*}(l_1, l_2)$ showing on the z-axis the elapsed time $e$ that should be executed for *no-drain* conditioned on the states; followed by $V^3(l_1, l_2)$ and $V^6(l_1, l_2)$.

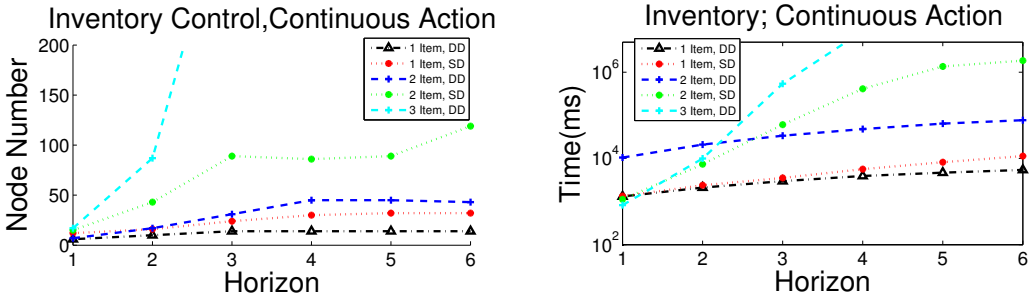



Figure 17: Continuous Action Inventory Control : space and time vs. horizon.

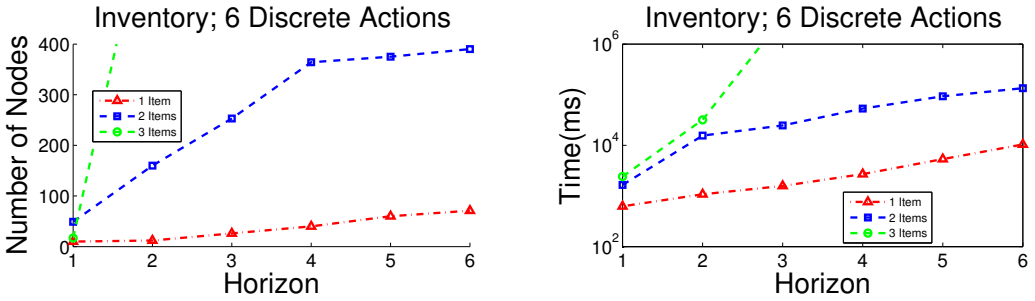



Figure 18: Discrete Action Inventory Control : Space and time vs. horizon for 6 discrete actions. Results are shown for various continuous items in the inventory.

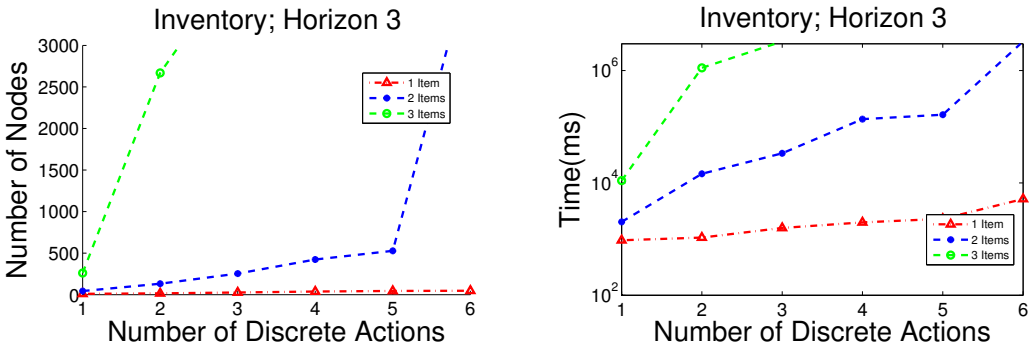



Figure 19: Space and elapsed time vs. horizon for a fixed horizon of 3 and different problem sizes for the Discrete Action Inventory Control .

onwards. The reason is behind the high dimensions, for a 3-item inventory of 6 discrete actions, a total of $6 \times 6 \times 6$ actions are required!

Figure 19 illustrates the effect of different number of action discretizations for the Inventory Control . Time and space are presented for the fixed horizon of $h = 3$ and for inventory items of $1, 2$ and 3. For a 1-item inventory, as the number of discrete actions increases, the time and space grows almost linearly. However for there is an exponential blow-up of the number of nodes for the 2-item inventory beyond 5 discrete actions while the time increases dramatically. Similar to the previous experiment, the 3-item inventory problem does not scale for more than two discrete actions due to the complexity in the action space. As a result of comparing the continuous and discrete action setting for the

2-item Inventory Control problem, the advantage of the continuous action H-MDPs is obvious.

Finally the key point evident in the results is the fact that scaling to higher dimensions requires large amounts of memory and time. This may be achievable using faster hardware, however a more efficient solution is to scale our XADD framework.

## 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., 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.

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.