

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 univariate 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 may move continuously while navigating and 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

Another example is the INVENTORY CONTROL problem (Arrow et al., 1958) with continuous resources such as petroleum products where a business manager must decide what quantity of each item to order subject to uncertain demand, (joint) capacity constraints, and reordering costs. The RESERVOIR MANAGEMENT problem (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:

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

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

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

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 .

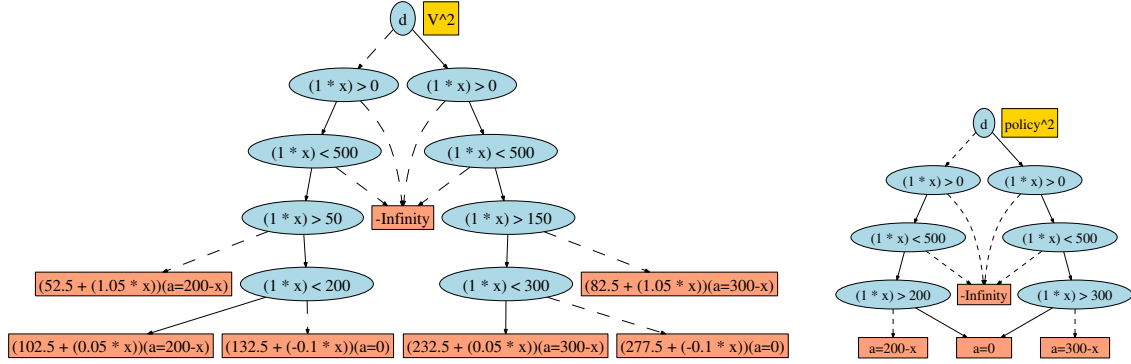


Figure 1: Optimal value function $V^2(x)$ for the CAIC problem represented as an extended algebraic decision diagram (XADD). Here the solid lines represent the *true* branch for the decision and the dashed lines the *false* branch. To evaluate $V^2(x)$ for any state x , one simply traverses the diagram 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 π^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 problem¹ 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 multi-variate 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 $(\vec{b}, \vec{x}) = (b_1, \dots, b_n, x_1, \dots, x_m)$. 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 $A = \{a_1(\vec{y}_1), \dots, a_p(\vec{y}_p)\}$, 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 γ , $0 \leq \gamma \leq 1$.²

A policy π specifies the action $a(\vec{y}) = \pi(\vec{b}, \vec{x})$ to take in each state (\vec{b}, \vec{x}) . Our goal is to find an optimal sequence of finite horizon-dependent policies³ $\Pi^* = (\pi^{*,1}, \dots, \pi^{*,H})$ that maximizes the expected sum of discounted rewards over a horizon $h \in H$; $H \geq 0$:

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

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

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 ϵ -optimal policy for arbitrary ϵ can be computed by SDP in finite time.

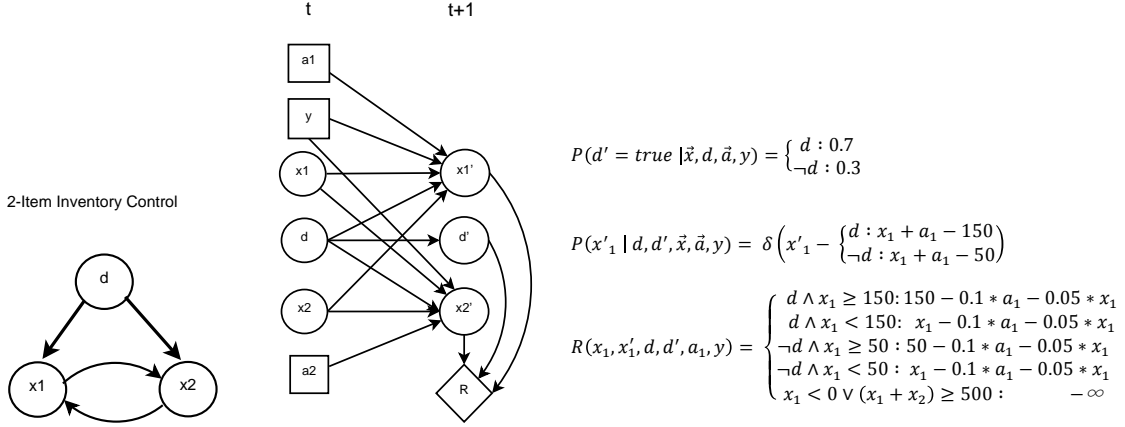


Figure 2: (*left*) Network topology between state variables in the 2-item continuous action INVENTORY CONTROL (CAIC) problem; (*middle*) DBN structure representing the transition and reward function; (*right*) transition probabilities and reward function in terms of CPF and PLE for x_1 .

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 $P(b'_i | \dots)$ and $P(x'_j | \dots)$ 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:

$$P(x'_1 | d, d', \vec{x}, \vec{a}, y) = \delta \left(x'_1 - \begin{cases} d : & x_i + a_i - 150 \\ -d : & x_i + a_i - 50 \end{cases} \right)$$

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 $x' = \{\dots\}$ 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 δ functions, which has been used frequently in previous exact continuous state MDP solutions (Feng et al., 2004; Meuleau, Benazera, Brafman, Hansen, & Mausam, 2009). Furthermore, we note that our 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 expressions.

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:

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

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

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

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:

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

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 the 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 only used with piecewise constant functions, in this work we generalize the representation to solve HMDPs with general piecewise functions. 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

The case representation presented in this section allows an expressive representation of general piecewise functions. Using such case representation we can extend the SDP framework in (Boutilier et al., 2001), which addresses only piecewise constant functions. This representation .

Definition (Case representation): A piecewise function in a case partition notation is given by:

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

where, $\phi_i : f_i$ is a case partition; ϕ_i is a logical formula and f_i can be defined as an arbitrary function. The formula ϕ_i is defined by arbitrary logical operations (\wedge, \vee, \neg over: (I) boolean variables in \vec{b} and (II) inequalities ($\geq, >, \leq, <$) involving continuous variables \vec{x} . Each ϕ_i is disjoint from the other ϕ_j ($j \neq i$).

However the ϕ_i may not exhaustively cover the state space, i.e., f may only be a partial function and undefined for some variable assignments. However, in this work, we assume f_i to be either linear or univariate quadratic on \vec{x} . Similarly we assume ϕ_i is defined over linear inequalities.

The following example is a simple function in this case form:

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

Note that any function f can be represented by a case partition notation where all ϕ_i are conjunctions, i.e., if $f = \{\phi : f_1$ and $\phi = \alpha_1 \vee \alpha_2$, then:

$$f = \{\phi : f_1 = \begin{cases} \alpha_1 : & f_1 \\ \alpha_2 : & f_1, \end{cases}$$

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.

3.2 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 ϕ_i and ψ_j is used to perform the corresponding operation:

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

$$\left\{ \begin{array}{l} \phi_1 : f_1 \\ \phi_2 : f_2 \end{array} \right\} \ominus \left\{ \begin{array}{l} \psi_1 : g_1 \\ \psi_2 : g_2 \end{array} \right\} = \left\{ \begin{array}{l} \phi_1 \wedge \psi_1 : f_1 - g_1 \\ \phi_1 \wedge \psi_2 : f_1 - g_2 \\ \phi_2 \wedge \psi_1 : f_2 - g_1 \\ \phi_2 \wedge \psi_2 : f_2 - g_2 \end{array} \right.$$

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

As in Section 3.1 for CA-HMDPs, the functions f_i and g_i are restricted to be linear or univariate 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, 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:

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

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.

3.3 Unary Operations

3.3.1 SCALAR MULTIPLICATION AND NEGATION

Scalar multiplication $c \cdot f$ (for a constant $c \in \mathbb{R}$) and negation $-f$ on case statements are applied to each case partition f_i ($1 \leq i \leq k$) as follows:

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

3.3.2 RESTRICTION

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

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

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

3.3.3 SUBSTITUTION

Symbolic substitution takes a set σ of variables and their substitutions and applies it to function f . Consider $\sigma = \{x'_1/(x_1+x_2), x'_2/x_1^2 \exp(x_2)\}$ where 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 σ is $f_i\sigma$. If $f_i = x'_1 + x'_2$ then using the σ defined above we obtain $f_i\sigma = x_1 + x_2 + x_1^2 \exp(x_2)$.

Substitution into case partitions ϕ_j is performed by applying σ to each inequality operand, i.e.:

$$f\sigma = \begin{cases} \phi_1\sigma : & f_1\sigma \\ \vdots & \vdots \\ \phi_k\sigma : & f_k\sigma \end{cases} \quad (7)$$

Note that if f has mutually exclusive partitions ϕ_i ($1 \leq i \leq k$) then $f\sigma$ must also have mutually exclusive partitions. This is followed from the logical consequence that if $\phi_1 \wedge \phi_2 \models \perp$ then $\phi_1\sigma \wedge \phi_2\sigma \models \perp$.

Consider the following example with the σ given above and f given by:

$$f = \begin{cases} x'_1 \leq \exp(x'_2) : & x'_1 + x'_2 \\ x'_1 \geq \exp(x'_2) : & x_1'^2 - 3x'_2 \end{cases}$$

then $f\sigma$ is as follows:

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

3.3.4 MARGINALIZATION OF CONTINUOUS VARIABLES

Marginalization of a continuous variable x_j is performed by the integral $\int_{x_j=-\infty}^{\infty} f(x_j)P(x_j)dx_j$. As mentioned before, we assume the probabilities of continuous variables are encoded using Direc δ functions. Thus, the integral becomes $\int_{x_j=-\infty}^{\infty} f(x_j)\delta[x_j - h(\vec{z})]$, where $h(\vec{z})$ can be defined as a case statement and \vec{z} does not contain x_j . This integral triggers the substitution $\sigma = \{x_j/h(\vec{z})\}$ on f , that is

$$\int_{x_j=-\infty}^{\infty} f(x_j)\delta \left[x_j - \begin{cases} \phi_1 : & h_1 \\ \vdots & \vdots \\ \phi_k : & h_k \end{cases} \right] dx_j = \begin{cases} \phi_1 : & f\{x_j/h_1\} \\ \vdots & \vdots \\ \phi_k : & f\{x_j/h_k\} \end{cases}. \quad (8)$$

As an example consider marginalizing x'_1 in the function:

$$f(x'_1, x_1, x_2) = \begin{cases} x'_1 \geq 5 : & x'_1 + x_2 \\ x'_1 < 5 : & x_1 \end{cases}$$

with the transition probability function $\delta[x'_1 - (2x_1 + x_2)]$.

The result of marginalization is:

$$\begin{aligned} f(x'_1, x_1, x_2) &= \begin{cases} x'_1 \geq 5 : & x'_1 + x_2 \\ x'_1 < 5 : & x_1 \end{cases} \\ \int_{x'_1} f(x'_1, x_1, x_2)\delta[x'_1 - (2x_1 + x_2)] dx'_1 &= \begin{cases} 2x_1 + x_2 - 5 \geq 0 : & 2x_1 + 2x_2 \\ 2x_1 + x_2 - 5 < 0 : & x_1 \end{cases} \end{aligned}$$

For a generic δ function with multiple case partitions, each of the f partitions are replaced into the multiple partitions of δ . Such nested case statements are then reduced back down to a standard case statement using a simple approach:

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

3.3.5 CONTINUOUS MAXIMIZATION

Continuous Maximization of a function w.r.t \vec{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 of (\vec{b}, \vec{x}) , i.e., $h(\vec{b}, \vec{x}) := \arg \max_{\vec{y}} f(\vec{b}, \vec{x}, \vec{y})$, thus the maximization is a *symbolic* constrained optimization.

Exploiting the commutativity of \max , we can first rewrite any multivariate $\max_{\vec{y}}$ as a sequence of univariate \max operations $\max_{y_1} \cdots \max_{y_{|\vec{y}|}}$, i.e.,

$$\max_{\vec{y}} = \max_{y_{|\vec{y}|}}(\cdots(\max_{y_2}(\max_{y_1}));$$

hence it suffices to provide just the univariate \max_y solution: $g(\vec{b}, \vec{x}) := \max_y f(\vec{b}, \vec{x}, y)$.

Also, as a consequence of the mutual disjointness of the case partitions in f , we can rewrite $f(\vec{b}, \vec{x}, y)$ as:⁵

$$f = \begin{Bmatrix} \phi_1 : & f_1 \\ \vdots & \vdots \\ \phi_k : & f_k \end{Bmatrix} = \text{casemax} \left[\begin{Bmatrix} \phi_1 : & f_1 \\ \neg\phi_1 : & -\infty \end{Bmatrix}, \cdots, \begin{Bmatrix} \phi_k : & f_k \\ \neg\phi_k : & -\infty \end{Bmatrix} \right] = \text{casemax}_{i=1, \dots, k} \begin{Bmatrix} \phi_i : & f_i \\ \neg\phi_i : & -\infty \end{Bmatrix}$$

where $\text{casemax}_{i=1, \dots, k}$ denotes casemax for $1 \leq i \leq k$; and since we have $\text{casemax}(F, -\infty) = F$, a partition $\begin{Bmatrix} \phi_i : & f_i \\ \neg\phi_i : & -\infty \end{Bmatrix}$ can be represented only by the case partition $\phi_i : f_i$.

Finally, we use these transformations to compute the $\max_y f(\vec{b}, \vec{x}, y)$ as follows:

$$\max_y f(\vec{b}, \vec{x}, y) = \max_y \text{casemax}_{i=1, \dots, k} \phi_i(\vec{b}, \vec{x}, y) : f_i(\vec{b}, \vec{x}, y). \quad (9)$$

Because \max_y and casemax_i are commutative, i.e., can be reordered, we compute \max_y for *each case partition individually* as:

$$g(\vec{b}, \vec{x}) = \max_y f(\vec{b}, \vec{x}, y) = \text{casemax}_i \boxed{\max_y \phi_i(\vec{b}, \vec{x}, y) : f_i(\vec{b}, \vec{x}, y)}. \quad (10)$$

Maximum of a single case partition. We can now show how to (symbolically) compute the maximum for a single case partition $\max_y \phi_i(\vec{b}, \vec{x}, y) : f_i(\vec{b}, \vec{x}, y)$.

We know that the maximum of 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 in the quadratic case, in the *Root* (i.e., zero) of $\frac{\partial}{\partial y} f_i$. Notice that each of *UB*, *LB*, and *Root* is a symbolic function of \vec{b} and \vec{x} .

We observe that each constraint of the conjunction⁶ ϕ_i serves one of purposes:

- if the constraint is of the kind $y < \alpha$ or $y \leq \alpha$ then $\alpha \in$ *upper bound (UB) on y*;
- if the constraint is of the kind $y > \alpha$ or $y \geq \alpha$ then $\alpha \in$ *lower bound (LB) on y*⁷;

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

6. As we have mentioned in the case representation definition (Section 3.1), we can assume without loss of generality that partitions are only conjunctions.

7. 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.

- if the constraints do not contain y and can be safely factored outside of the \max_y then it belongs to an *independent set* (*Ind*).

Note that there can be multiple symbolic upper and lower bounds on y , thus in general we will need to determine the highest lower bound (*HLB*) and the lowest upper bound (*LUB*):

$$\begin{aligned} HLB &= \text{casemax}(\alpha_j), \forall \alpha_j \in LB \\ LUB &= \text{casemin}(\alpha_j), \forall \alpha_j \in UB \end{aligned}$$

Given the *potential* maximal points of f , $y = LUB$, $y = HLB$, and $y = \text{Root}$ 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:

$$Max = \begin{cases} \exists \text{Root and } HLB \leq \text{Root} \leq LUB: & \text{casemax}(f_i\{y/\text{Root}\}, f_i\{y/LUB\}, f_i\{y/HLB\}) \\ \text{elseif } HLB \leq LUB: & \text{casemax}(f_i\{y/UB\}, f_i\{y/LB\}) \end{cases} \quad (11)$$

The substitution operator $\{y/f\}$ replaces y with case statement f , as we have previously defined. Additional constraints arise from the symbolic nature of the *LUB*, *HLB*, and *Root*. Specifically, we need to ensure that indeed $HLB \leq \text{Root} \leq LUB$ (or if no root exists, then $HLB \leq LUB$)

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 two issues: (i) the incorporation of the independent (*Ind*) constraints (factored out previously). Finally, we express the final result as a single case partition:

$$\max_y \phi_i(\vec{b}, \vec{x}, y) f_i(\vec{b}, \vec{x}, y) = \text{Ind} : Max. \quad (12)$$

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. In the next section we will illustrate these operations using the *INVENTORY CONTROL* example.

3.4 Symbolic Value Iteration (SVI)

In this section the symbolic value iteration algorithm (SVI) for HMDPs is presented. The Value Iteration (Algorithm 1) takes 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 optimal value function V^h at horizon h in the form of a case statement. We use the *CAIC* example from the Introduction Section to clarify each step of this algorithm.

For the base case ($h = 0$) in line 2, the Value Iteration algorithm sets $V^0(\vec{b}, \vec{x}) = 0$ (or to the reward case statement, if it is not action dependent). This operation is trivially performed in the form of a case statement.

For $h > 0$ and for each action Value Iteration calls the *Regress* algorithm in Line 6. 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 (Algorithm 2) can be performed using the following steps:

Algorithm 1: $\text{SVI}(\text{HMDP}, H) \rightarrow (V^h, \pi^{*,h})$

```

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

```

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

```

1 begin
2    $Q \leftarrow \text{Prime}(V)$  // All  $v_i \rightarrow v'_i (\equiv \text{all } b_i \rightarrow b'_i \text{ and all } x_i \rightarrow x'_i)$ 
3   if  $v'$  in  $R$  then
4      $Q \leftarrow R(\vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y}) \oplus (\gamma \cdot Q)$ 
5
6   foreach  $v'$  in  $Q$  do
7     if  $v' = x'_j$  then
8       //Continuous marginal integration
9        $Q \leftarrow \int Q \otimes P(x'_j | \vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y}) d_{x'_j}$ 
10    if  $v' = b'_i$  then
11      // Discrete marginal summation
12       $Q \leftarrow [Q \otimes P(b'_i | \vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y})] |_{b'_i=1} \oplus [Q \otimes P(b'_i | \vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y})] |_{b'_i=0}$ 
13
14    if  $\neg (v' \text{ in } R)$  then
15       $Q \leftarrow R(\vec{b}, \vec{b}', \vec{x}, \vec{x}', a, \vec{y}) \oplus (\gamma \cdot Q)$ 
16
17    return  $Q$ 
18 end

```

1. *Prime the Value Function:* V^h will be the “next state” in value iteration thus the substitution operation of $\sigma = \{b_1/b'_1, \dots, b_n/b'_n, x_1/x'_1, \dots, x_m/x'_m\}$ is used to obtain $V^h = V^h \sigma$ (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:

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

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:

$$Q = \begin{cases} x < 0 \vee x > 500 & : -\infty \\ d \wedge (x \geq 150) \wedge (150 \leq (x+a) \leq 650) & : 150 - 0.1 * a - 0.05 * x \\ d \wedge (x \geq 150) \wedge ((x+a \geq 650) \vee (x+a \leq 150)) & : -\infty \\ d \wedge (x \leq 150) \wedge (150 \leq (x+a) \leq 650) & : 0.95 * x - 0.1 * a \\ d \wedge (x \leq 150) \wedge ((x+a \geq 650) \vee (x+a \leq 150)) & : -\infty \\ \neg d \wedge (x \geq 50) \wedge (50 \leq (x+a) \leq 550) & : 50 - 0.1 * a - 0.05 * x \\ \neg d \wedge (x \geq 50) \wedge ((x+a \geq 550) \vee (x+a \leq 50)) & : -\infty \\ \neg d \wedge (x \leq 50) \wedge (50 \leq (x+a) \leq 550) & : 0.95 * x - 0.1 * a \\ \neg d \wedge (x \leq 50) \wedge ((x+a \geq 550) \vee (x+a \leq 50)) & : -\infty \end{cases} \quad (13)$$

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 $P(b'_i | \dots)$ 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 (13).

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 $\max_{y_1} \dots \max_{y_{|\vec{y}|}}$; hence it suffices to provide just the *univariate* \max_y solution:

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

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.⁸ Each step of this algorithm is followed using one of the partitions of the Q-function in (13) in this case the first partition with the constraints and function value defined below:

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

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

$$\begin{aligned} LB &= \text{casemax}(0, 150 - x) = \begin{cases} x > 150 : & 0 \\ x \leq 150 : & 150 - x \end{cases} \\ UB &= \text{casemin}(1000000, 650 - x) = \begin{cases} x > -1000000 : & 650 - x \\ x \leq -1000000 : & 1000000 \end{cases} \end{aligned}$$

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

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

8. From here out we assume that all case partition conditions ϕ_i of f consist of conjunctions of non-negated linear inequalities and possibly negated boolean variables — conditions easy to enforce 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($y, f(\vec{b}, \vec{x}, y) \rightarrow (\max_y f(\vec{b}, \vec{x}, y))$)

```

1 begin
2    $LB \leftarrow -\infty, UB \leftarrow +\infty, IND, CONS \leftarrow true, Case_{max} \leftarrow \emptyset$ 
3   for  $\phi_i \in f$  (For all partitions of  $f$ ) do
4     for  $c_i \in \phi_i$  (For all conditions  $c$  of  $\phi_i$ ) do
5       if  $c_i \leq y$  then
6          $LB \leftarrow casemax(LB, c_i)$  //Add  $c_i$  to  $LB$ , take max of all  $LB$ s
7       if  $c_i \geq y$  then
8          $UB \leftarrow casemin(UB, c_i)$  //Add  $c_i$  to  $UB$ , take min of all  $UB$ s
9       else
10         $IND \leftarrow [IND, c_i]$  //Add constraint  $c_i$  to independent constraint set
11
12
13    $Root \leftarrow \text{SolveForVar}(y, f_i)$ 
14   if ( $Root \neq null$ ) then
15      $CONS \leftarrow (\mathbb{I}[LB] \leq \mathbb{I}[Root]) \wedge (\mathbb{I}[Root] \leq \mathbb{I}[UB])$ 
16   else
17      $CONS \leftarrow (\mathbb{I}[LB] \leq \mathbb{I}[UB])$ 
18   //Conditions and value of continuous max for this partition
19    $Max \leftarrow IND \wedge CONS : casemax(f_i\{y/LB\}, f_i\{y/UB\}, f_i\{y/Root\})$ 
20   //Take maximum of this partition and all other partitions
21    $Case_{max} \leftarrow \max(Case_{max}, Max)$ 
22   return  $Case_{max}$ 
23 end
```

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 :

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

\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 (10), we have now specified the inner operation (shown in the \square) for this partition after removing all redundant and inconsistent cases.⁹

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

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 $a \in \{a_1(\vec{y}_1), \dots, a_p(\vec{y}_p)\}$, obtaining V^{h+1} in case format as defined in (6) requires sequentially applying *symbolic maximization* in line 14:

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

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 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 Algebraic 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.

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

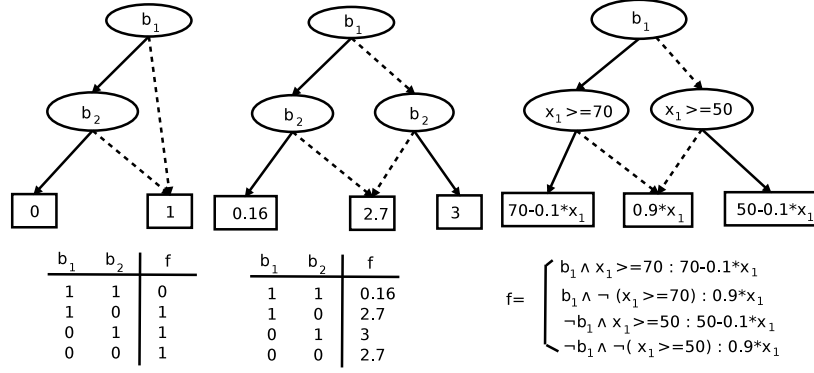


Figure 3: Examples of decision diagrams: (left) a BDD, (middle) an ADD and (right) an XADD.

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 algebraic functions instead of boolean functions. Figure 3 shows examples of three decision diagrams: (i) (left) a BDD where leaves are boolean values and the internal nodes are boolean decisions representing $f(b_1, b_2) = b_1 \bar{b}_2 + \bar{b}_1$ as shown in the truth table; (ii) (middle) an ADD where leaves are real values and the internal nodes are boolean decisions; and (iii) (right) an XADD where leaves are polynomial functions and internal nodes are polynomial inequalities.

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 *context-specifically independent* defined as the following:

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:

$$P(A|B, C = c) = P(A|C = c) \text{ whenever } P(B, C = c) > 0. \quad (14)$$

XADDs extend ADDs to allow representing functions with both boolean and continuous variables, i.e. $\{0, 1\}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$. Here decisions can be boolean variable tests or continuous inequalities and leaves can be defined as arbitrary continuous expressions. We next formally define XADDs and the operations and algorithms required to solve HMDP problems using SDP. Pruning algorithms are then introduced to make this representation even more efficient.

4.1 XADD Formal Definitions

An XADD is a function represented by a DAG involving variables (\vec{b}, \vec{x}) where $b_i \in \{0, 1\}$ ($1 \leq i \leq m$) is a boolean variable and $x_j \in [x_j^{\min}, x_j^{\max}]$ is a continuous variable, with $1 \leq j \leq n$; $x_j^{\min}, x_j^{\max} \in \mathbb{R}$; and $x_j^{\min} < x_j^{\max}$. An XADD structure consists of internal nodes (decision nodes) and leaf nodes (linear or nonlinear expressions). XADD allows an efficient implementation of piecewise functions represented in general case partition notation (Definition 3.1). Given a piecewise function:

$$f = \begin{cases} \phi_1(\vec{b}, \vec{x}) : & f_1(\vec{x}) \\ \vdots & \vdots \\ \phi_k(\vec{b}, \vec{x}) : & f_k(\vec{x}), \end{cases} \quad (15)$$

represented as an XADD F , each f_i is a leaf node and each $\phi_i(\vec{b}, \vec{x})$ (a logical formula) corresponds to a disjunction of the paths of decision nodes from root leading to the leaf f_i . For example, the XADD in Figure 1 (left) corresponds to the case statement of Equation 3. Similar to ADDs, an XADD 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, a general XADD allows arbitrary decision nodes and leaf expressions.

Definition (Linear Expression): Giving a set of continuous variables x_i and a set of constants c_i , a linear expression can be canonically defined as the following:

$$linearExpression = c_0 + \sum_i c_i x_i, 1 \leq i \leq n. \quad (16)$$

Definition (Linear XADD): Formally, an XADD with linear decisions and linear leaves expressions is defined using the following BNF grammar:

$$\begin{aligned} F &::= linearExpression \mid decisionNode \\ decisionNode &::= \text{if}(F_{dec}) \text{ then } F \text{ else } F \\ F_{dec} &::= b \mid (linearExpression \leq 0) \mid (linearExpression \geq 0) \\ b &::= 0 \mid 1 \end{aligned}$$

A node F of a Linear XADD is either a leaf node (also called terminal node) with a linear expression or a decision node (also called internal node) that contains a decision test F_{dec} and two branches F_h and F_l (both of type F). When following an XADD path, if F_{dec} is true, branch F_h is taken and if dec is false, F_l is taken. A decision node can be also represented by the triple $\langle F_{dec}, F_h, F_l \rangle$.

The decision test F_{dec} can be either a boolean variable $b \in \{0, 1\}$ or a linear inequality over continuous variables x_i ¹⁰. Figure 3 (right) represents an XADD consisting of leaf nodes such as the node with the linear expression $0.9 * x_1$ or decision nodes such as the nodes with the decision tests b_1 and $x_1 \geq 70$.

We can also define a polynomial XADD involving polynomial Expressions defined next.

Definition (Polynomial Expression): Giving a set of continuous variables x_i and a set of constants c_i , a polynomial expression can be canonically defined as the following:

$$polynomialExpression = \sum_i c_i \prod_j x_j^{P_{ij}}, 1 \leq i \leq k, 1 \leq j \leq n, \quad (17)$$

where k is the number of terms of the polynomial expression and P_{ij} is the power of variable x_j in the i -th term.

Definition (Polynomial XADD): Formally, an XADD with polynomial decisions and polynomial leaves expressions is defined using the following BNF grammar:

$$\begin{aligned} F & ::= polynomialExpression \mid decisionNode \\ decisionNode & ::= \text{if}(F_{dec}) \text{ then } F \text{ else } F \\ F_{dec} & ::= b \mid (polynomialExpression \leq 0) \mid (polynomialExpression \geq 0) \\ b & ::= 0 \mid 1 \end{aligned}$$

Note that a special case of polynomial XADD are the univariate quadratic XADDs, where the expressions are in the form of $c_0 + c_1 x_j^2 + \sum_i c_i x_i$ ($2 \leq i \leq n$). Since the univariate quadratic can be linearized, it can be transformed into a Linear XADD.

As we can evaluate a piecewise function with a given variable assignment, an XADD (linear or polynomial) can be evaluated as we define next.

Definition (XADD Evaluation): Given (linear or polynomial) XADD F that represents functions of type $\{0, 1\}^m \times \mathbb{R}^n \rightarrow \mathbb{R}$ and a variable assignment $\vec{\rho} \in \{\{0, 1\}^m, \mathbb{R}^n\}$ to (\vec{b}, \vec{x}) , an XADD evaluation, $Val(F, \vec{\rho})$, is a real value. $Val(F, \vec{\rho})$ can be computed recursively by:

$$Val(F, \vec{\rho}) = \begin{cases} \vec{\rho}(F), & \text{if } F = Expression \\ Val(F_h, \vec{\rho}), & \text{if } F = decisionNode\langle F_{dec}, F_h, F_l \rangle \text{ and } \vec{\rho}(F_{dec}) = true \\ Val(F_l, \vec{\rho}), & \text{if } F = decisionNode\langle F_{dec}, F_h, F_l \rangle \text{ and } \vec{\rho}(F_{dec}) = false, \end{cases}$$

where $\vec{\rho}(F) \in \mathbb{R}$, for $F = Expression$ is the value of F in the variable assignment $\vec{\rho}$. $\vec{\rho}(F_{dec})$ (a relational expression) is the value of F_{dec} in the variable assignment $\vec{\rho}$, that can be true or false.

10. Note we assume continuous functions. Thus, in the boundary point (equality) of two expressions, they must have the same value. This continuous property allows us to replace $< (>)$ with $\leq (\geq)$.

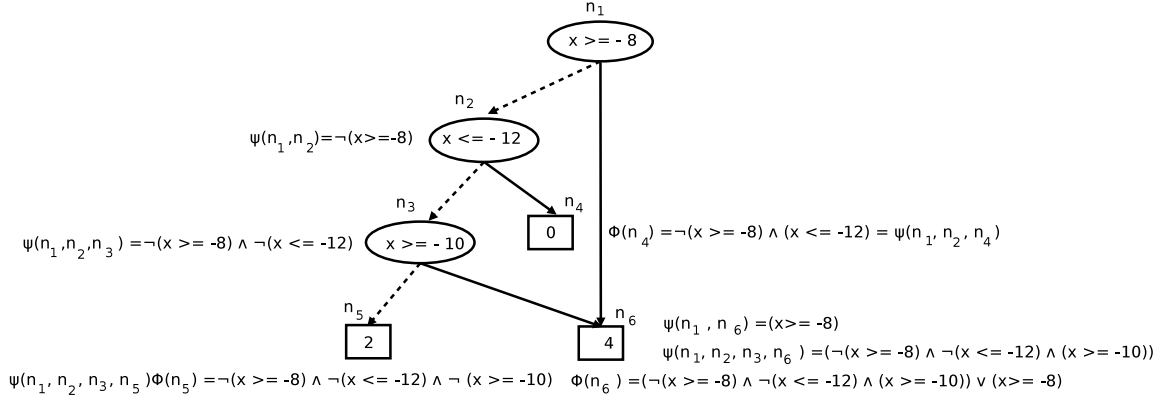


Figure 4: Path formulas and case partitions in an XADD.

This recursive definition reflects the structural evaluation of F starting at its root node and following the corresponding branch at each decision node according to $\vec{\rho}(F_{dec})$. The evaluation continues until a leaf node F has been reached, which then returns $\vec{\rho}(F)$. As an example, in Figure 3 (right), an assignment of $\vec{\rho} = \{1, 90\}$ to (b_1, x_1) yields to $Val(F, \vec{\rho}) = 8.1$.

Definition (Path Formula): Let $C = (F^0, F^1, F^2, \dots, F^k)$ be a path of nodes of an XADD F , from root to F^k (where F^k can be either a leaf node or a decision node). A path formula $\psi(C)$ is defined as:

$$\psi(C) \equiv \bigwedge_{i=0}^{k-1} \begin{cases} F_{dec}^i & \text{if } F^{i+1} = F_h^i \\ \neg F_{dec}^i & \text{if } F^{i+1} = F_l^i. \end{cases} \quad (18)$$

Figure 4 shows an XADD with formulas associated to each path ending at a decision node or leaf node. For instance, the path $C = (n_1, n_2, n_3)$ defines a path formula $\psi(C) = \neg(x \geq -8) \wedge \neg(x \leq -12)$.

Recall the leaf node F^i has the value f_i in the case representation of Equation 15. We can now formally define a *case partition* $\langle \phi_i(\vec{b}, \vec{x}) : f_i(\vec{x}) \rangle$ using the path formula definition.

Definition (Case Partition): Let F^i be a leaf node of an XADD; and $\mathbb{C} = \{C_1^i, \dots, C_d^i\}$ be all paths from the root to F^i . A case partition $\phi(F^i)$ is the disjunction of the path formula for each path in \mathbb{C} , i.e.:

$$\phi(F^i) \equiv \bigvee_{j=1}^d \psi(C_j^i), \quad (19)$$

i.e., a disjunction of all paths that leads to the node F^i . For example, in Figure 4, a case partition for the leaf node with value equal to 4 (n_6) is $\phi(n_6) = \{(x \geq -8) \vee (\neg(x \geq -8) \wedge \neg(x \leq -12) \wedge (x \geq -10))\}$.

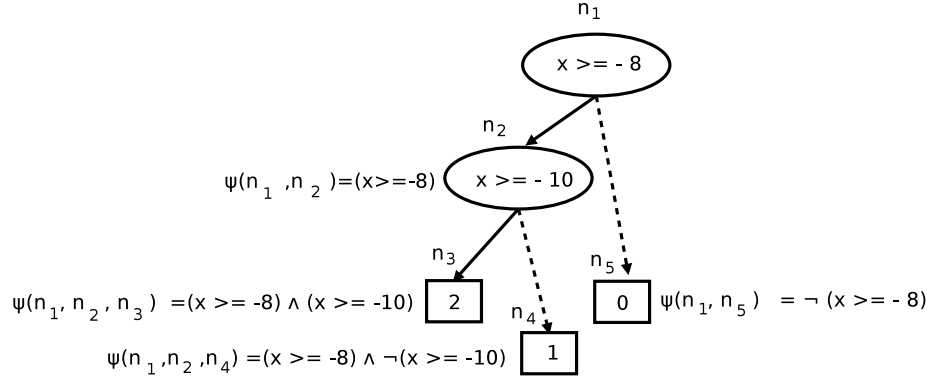


Figure 5: XADD with an Inconsistent Path Formula $\psi(n_1, n_2, n_4)$, i.e. $((x \geq -8) \wedge \neg(x \geq -10)) \models \perp$.

Definition (XADDs equivalency): Let f^1 and f^2 be two piecewise functions represented by the XADDs F^1 and F^2 , respectively. F^1 is equivalent to F^2 (denoted by $F^1 \equiv F^2$) if $f^1(\vec{b}, \vec{x}) = f^2(\vec{b}, \vec{x}), \forall(\vec{b}, \vec{x})$.

Another important aspect of an XADD is the introduction of redundant nodes or nodes that result in inconsistent path formula.

Definition (Inconsistent path formula): A path formula ψ in XADD F is inconsistent if $\psi \models \perp$.

The XADD of Figure 5 has an inconsistent path formula for path $C = (n_1, n_2, n_4)$ on which the decision $\neg(x \geq -10)$ violates the previous constraint of $x \geq -8$ (in n_1), i.e.: $(x \geq -8) \wedge \neg(x \geq -10) \models \perp$.

Definition (Redundant node): 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. A decision node F is redundant if it can be replaced by one of its branches. That is, $F \equiv F_h$ or $F \equiv F_l$ (according with XADD equivalency definition).

As an example, consider the XADD of Figure 6 where removing node n_1 will not affect the XADD function: if $x < -8$ then $f^{n_1}(x) = f^{n_2}(x)$, otherwise $(x \geq -8)$ $f^{n_1}(x) = f^{n_2}(x) = 4$. Therefore $F^{n_1} \equiv F_l^{n_1}$ and thus n_1 is redundant.

In the next section we will present algorithms that can detect and prune inconsistent paths and redundant nodes of an XADD.

4.2 XADD Operations

One of the most important feature of XADDs is to compactly represent a piecewise function f . There are two main functions used to make a more compact (reduced) representation of an XADD: Reduce and GetNode. A reduced XADD is an XADD where identical branches are merged. The function Reduce calls GetNode to create a new XADD node or merge identical branches of a node.

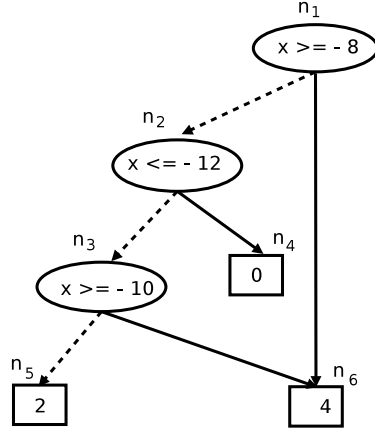


Figure 6: XADD with a Redundant Node n_1 : it is redundant since n_2 represents the same function as n_1 , i.e., $F^{n_1} \equiv F_l^{n_1}$.

Algorithm 4: $\text{GETNODE}(F_{dec}, F_h, F_l) \rightarrow \langle F^r \rangle$

```

1 begin
2   //NodeCache is a hash table that stores an id to each node  $F$ 
3   //redundant branches
4   if  $F_l = F_h$  then
5     | return  $F_l$ 
6    $id \leftarrow \text{NodeCache}(\langle F_{dec}, F_h, F_l \rangle)$ 
7   //check if the node already exists
8   if  $id = \text{null}$  then
9     |  $id \leftarrow$  new unallocated id
10    |  $\text{NodeCache}(\langle F_{dec}, F_h, F_l \rangle) \leftarrow id$ 
11  return  $id$ 
12 end
    
```

4.2.1 REDUCE AND GETNODE

The function *GetNode* returns a compact representation of a single internal decision node of an XADD (Algorithm 4). *GetNode* is called by several XADD algorithms we will introduce. It checks the branches of a node F so that it does not have identical F_h and F_l . If a repeated branch is found, it simply returns one of them (Lines 4–5 of Algorithm 4). Otherwise this function must build a new node with a unique id. Each new id is stored in a *NodeCache* hash table. Before adding a new node, the cache is checked for an identical node (Lines 7–9 of Algorithm 4).

The function *ReduceXADD* (Algorithm 5) allows the construction of a compact XADD representation for a given decision diagram. This algorithm recursively constructs a reduced XADD from the bottom up. If F is a leaf node, i.e. a linear expression, the algorithm returns the canonical form of F (Equation 16) (Lines 5–6 of Algorithm 5). If F is a decision node

Algorithm 5: REDUCEXADD(F) $\rightarrow \langle F^r \rangle$

```

1 begin
2   //F is a root node id for an arbitrary ordered decision diagram
3   //Fr is a root node id for the reduced XADD
4   // ReduceCache is a hash table that stores previously redundant XADDs
5   if  $F$  is terminal node then
6     | return terminal node for  $F$ , i.e., a canonical linear expression
7   //else reduce decision node  $F$ 
8    $F^r \leftarrow \text{ReduceCache}(F)$ 
9   if  $F^r = \text{null}$  then
10    |  $F_h \leftarrow \text{REDUCEXADD}(F_h)$ 
11    |  $F_l \leftarrow \text{REDUCEXADD}(F_l)$ 
12    | //get a canonical internal node id
13    |  $F^r \leftarrow \text{GETNODE}(F_{dec}, F_h, F_l)$ 
14    |  $\text{ReduceCache}(F) \leftarrow F^r$ 
15  return  $F^r$ 
16 end

```

then the algorithm recursively computes the reduced XADDs of the low and high branches and returns a final reduced XADD F^r (Lines 8–14 of Algorithm 5). Note that for a decision node $\langle F_{dec}, F_h, F_l \rangle$, F_h and F_l are the id for high and low branches, respectively. Algorithm 5 uses the *GetNode* function in Line 12 to merge identical branches. The reduced XADD F^r is then stored in the *ReduceCache* hash table. *ReduceCache* ensures that each node is visited only once by Algorithm 5; it has linear running time according to the number of nodes in F^r . As in the case representation, the symbolic operations required to perform SDP (Section 3.1) can be also performed using XADD operations, mainly categorized into unary and binary operations.

4.2.2 UNARY XADD OPERATIONS

According to Section 3 the unary operations required in SDP are scalar multiplication ($c.f$), negation ($-f$), restriction ($f|_\phi$), marginalization ($\sum_{b_i} f$), substitution (f_σ), integration ($\int_x f$) and continuous maximization ($\max_{\vec{y}} f$) and linearize, where f is a piecewise linear function, $c \in \mathbb{R}$, ϕ is a logical formula and σ is a set of variables and their substitution. In this section we briefly describe how this operations are performed using XADD representation.

Scalar multiplication, $c.F$, where $c \in \mathbb{R}$ and F is an XADD is performed by multiplying all the XADD leaves by c while *negation* of an XADD F is the multiplication $-1.F$.

The *restriction* operation for an XADD ($F|_\phi$) is done only over binary variables. Thus ϕ is of the form $b_i = \text{true}$ or $b_i = \text{false}$. Restriction of an XADD F to some formula ϕ is performed by making a conjunction of ϕ to each case partition ϕ_i , i.e., $\phi_i \wedge \phi$. This is equal to taking the true or false branch of the decision node b_i . This operation can also be used for *marginalizing boolean variables*, represented by \sum_{b_i} which eliminates variable b_i from F by computing the sum $F|_{b_i=\text{true}} \oplus F|_{b_i=\text{false}}$ (the \oplus binary operation is performed

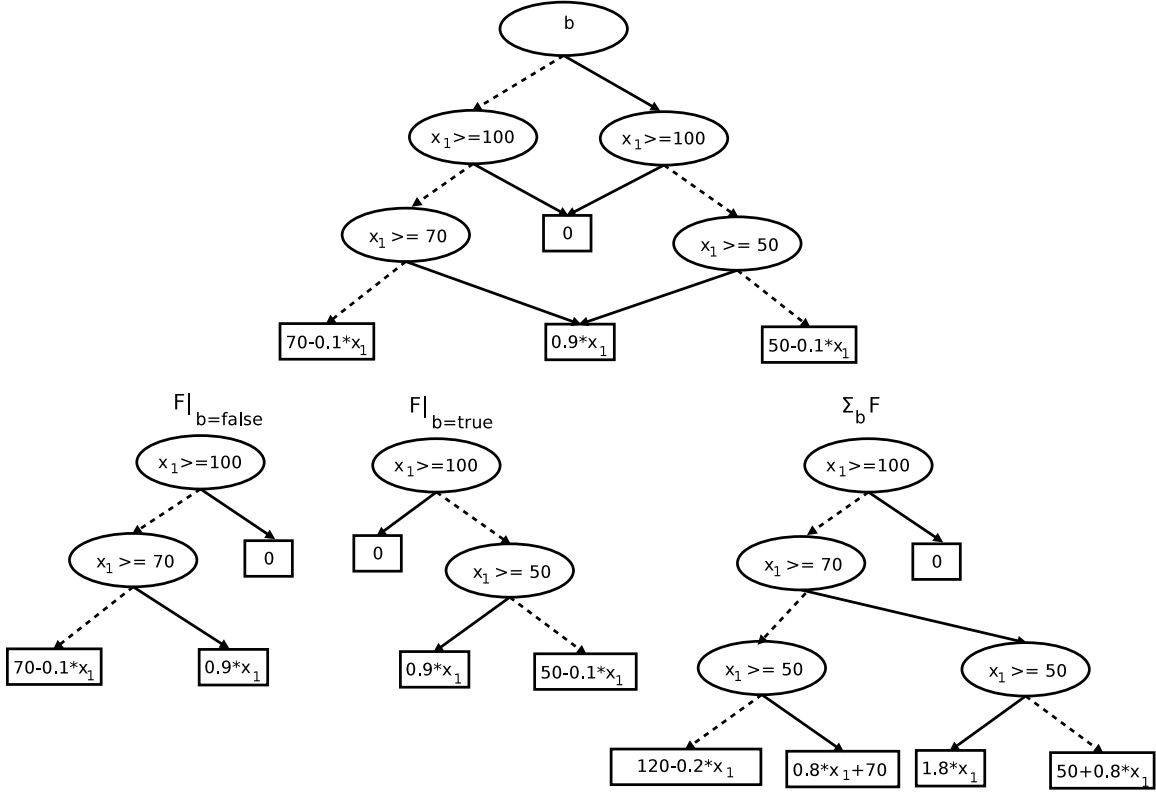


Figure 7: (bottom left) Example of restriction operation $F|_{b=true}$ and $F|_{b=false}$ on the XADD F (top). (bottom right) The resulting XADD after marginalization $\sum_b F$.

by Algorithm 9 described in the next section). Figure 7 illustrates the operations $F|_{d=true}$, $F|_{d=false}$ and $\sum_d F$.

Substitution of a given XADD F by σ is performed by applying σ in each case statement as in Equation 7. The substitution operand affects both leaves and decision nodes and changes them according to the variable substitution. Decisions may become unordered after the substitution operation and a reorder algorithm has to be applied after this operation. Algorithm 6 reorders an XADD F by recursively applying operations of \otimes and \oplus to decision nodes. Note that $\mathbb{I}[F_{dec}]$ is the indicator function for F_{dec} , i.e.:

$$\mathbb{I}[F_{dec}] = \begin{cases} F_{dec} & : \quad 1 \\ \neg F_{dec} & : \quad 0 \end{cases}$$

Similarly to previous algorithms, a *ReorderCache* is used to prevent unnecessary computations. An example of the application of reorder is shown in Figure 8, which reorder the XADD of Figure 3(right), with the new order of nodes given by: $\text{Order}(x_1 \geq 70) < \text{Order}(x_1 \geq 50) < \text{Order}(b_1)$. The computation of F_h and F_l (lines 7 and 8 of Algorithm

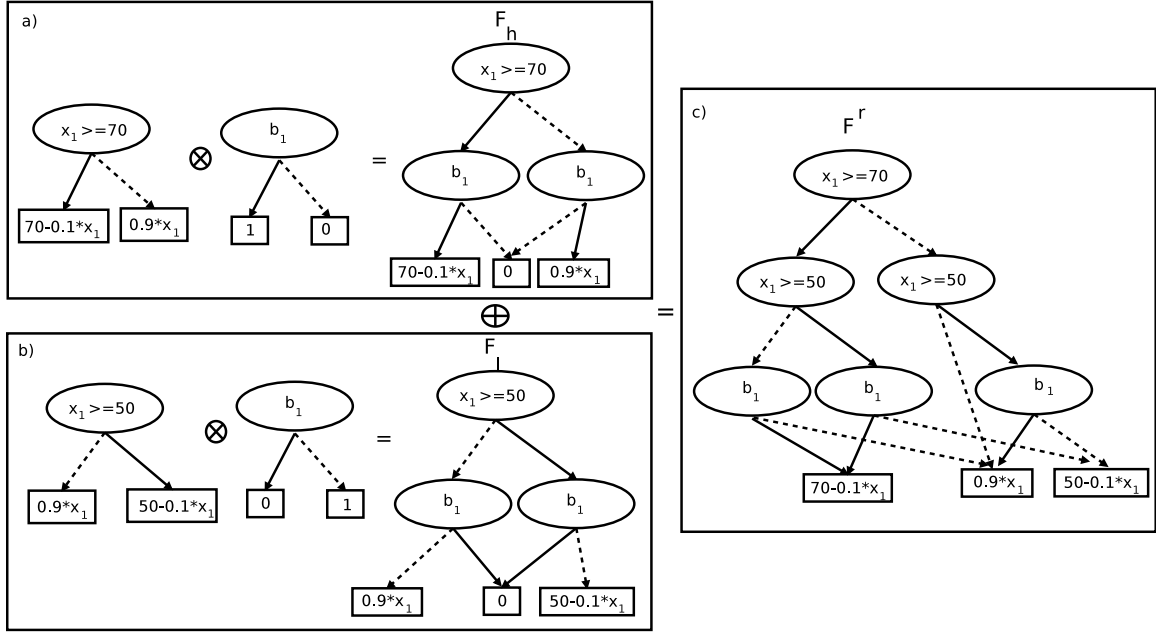


Figure 8: Applying reorder on the XADD of Figure 3 (right) with the new order of nodes given by: $\text{Order}(x_1 \geq 70) < \text{Order}(x_1 \geq 50) < \text{Order}(b_1)$. (a) The resulting reorder XADD F_h . (b) The resulting reorder XADD F_l . (c) The resulting reordered XADD F^r .

Algorithm 6: $\text{REORDER}(F) \rightarrow \langle F^r \rangle$

```

1 begin
2   if  $F$  is a terminal node then
3     return canonical terminal node of  $F$ 
4   // If  $F$  is not a terminal node,  $F$  is  $\langle F_{dec}, F_h, F_l \rangle$ 
5    $F^r \leftarrow \text{ReorderCache}(F)$ 
6   if  $F^r = \text{null}$  then
7      $F_h \leftarrow \text{REORDER}(F_h) \otimes \mathbb{I}[F_{dec}]$ 
8      $F_l \leftarrow \text{REORDER}(F_l) \otimes \mathbb{I}[\neg F_{dec}]$ 
9      $F^r \leftarrow F_h \oplus F_l$ 
10     $\text{ReorderCache}(F) \leftarrow F^r$ 
11  return  $F^r$ 
12 end

```

6) are showed in Figure 8(a) and Figure 8(b), respectively. Finally, the resulting XADD F^r is shown in Figure 8(c).

Algorithm 7: REDUCELINEARIZE(F) \longrightarrow $\langle F^r \rangle$

```

1 begin
2   if  $F$  is terminal node then
3     return canonical terminal node of  $F$ 
4    $F^r \leftarrow \text{LinearCache}(F)$ 
5   if  $F^r = \text{null}$  then
6     //use recursion to reduce sub diagrams
7      $F_h \leftarrow \text{LINEARIZE}(F_h)$ 
8      $F_l \leftarrow \text{LINEARIZE}(F_l)$ 
9     //get a linearized internal node id
10     $F^r \leftarrow \text{LINEARIZEDECISIONNODE}(F_{dec}, F_h, F_l)$ 
11     $\text{LinearCache}(F) \leftarrow F^r$ 
12  return  $F^r$ 
13 end

```

As we have seen in Section 3.3.4, *marginalizing a continuous variable* y is performed using the integration of the δ -function on variable y . In particular for SDP we require computing $\int_y \delta[y - g(\vec{x})] f dy$ which triggers the substitution $f\{y/g(\vec{x})\}$ on f .

The unary operation *linearize* takes an XADD F with linear and univariate quadratic decision nodes and transforms into an XADD F^r where all decision nodes are linear decisions. The Linearize algorithm (Algorithm 7) similarly to other XADD algorithms uses 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 *LinearizeDecisionNode* that finds the roots of the non-linear function and creates new linear decisions with these roots. By recursively applying this algorithm on the low and high branches, all nodes of F are linearized.

Finally, the other operation required to solve CA-HMDPs using SDP is the *continuous maximization* ($\max_{\vec{y}} f$) over continuous action parameters \vec{y} . As we have seen in Section 3.3.5, it suffices to provide just the univariate \max_y solution, to be performed at each case partition and the max of each partition can be computed by Equation 10. The operation of continuous maximization can also be defined using XADDs. As we have defined in Section 4.1 each XADD path from root to leaf node is treated as a single case partition with conjunctive constraints.

An example of $\max_{\vec{y}} f$ for a single case partition is presented in Figures 9, 10, 11 and 12 using XADD representation. This example is more complex compared to the CAIC partition mentioned in Section 3.4 since it involves a case partition with a quadratic expression leaf. The six steps follows the procedure specified in Section 3.3.5 for the maximum of a single case partition: (1) finding the sets UB, LB, Root, and Ind; (2) compute HLB and LUB; (3) apply the substitution operator $\{y/f\}$, with Root, HLB and LUB; (4) Compute the conditions $HLB \leq LUB$ and $HLB \leq \text{Root} \leq LUB$ (5) compute casemax of the substitutions result; and (6) computing the final result as a new case partition Equation 12.

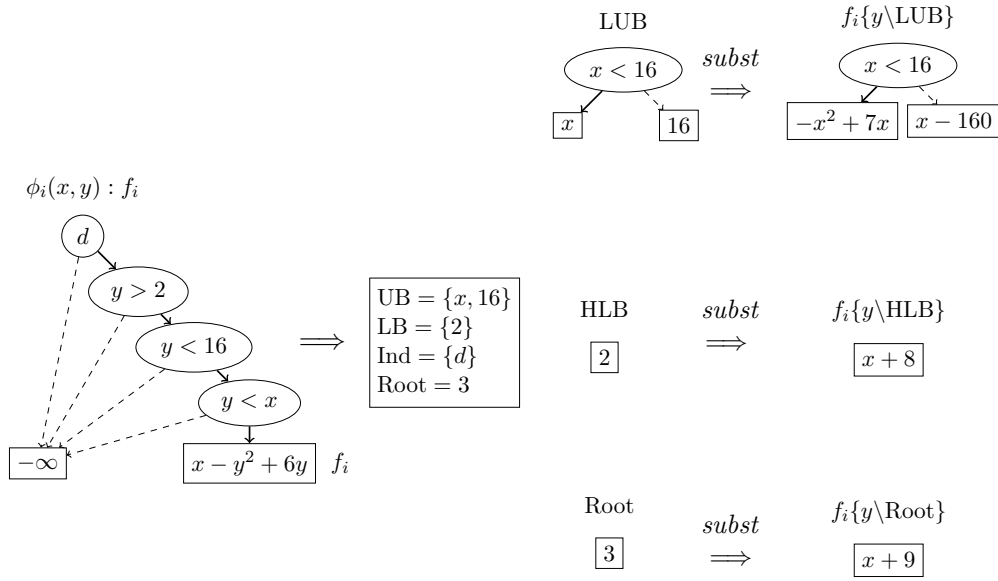


Figure 9: Step 1, 2 and 3 of XADD Continuous Maximization for one case partition: $\phi_i(x, d, y) : f_i$ where $\phi_i = d \wedge (y > 2) \wedge (y < 16) \wedge (y < x)$ and $f_i = x - y^2 + 6y$. (1) Finding the sets UB, LB, Root, and Ind; (2) compute HLB and LUB; and (3) apply the substitution operator $\{y/f\}$, with Root, HLB and LUB.

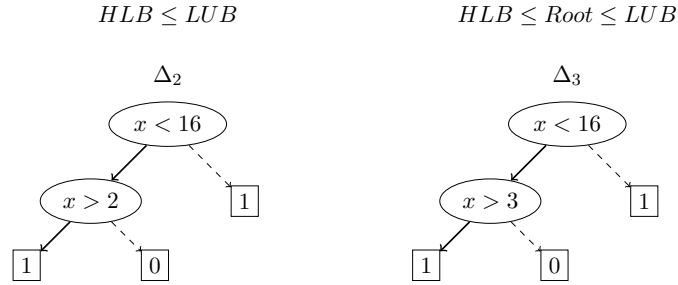


Figure 10: Step 4 of XADD Continuous Maximization for one case partition: Compute the conditions $HLB \leq LUB$ and $HLB \leq Root \leq LUB$.

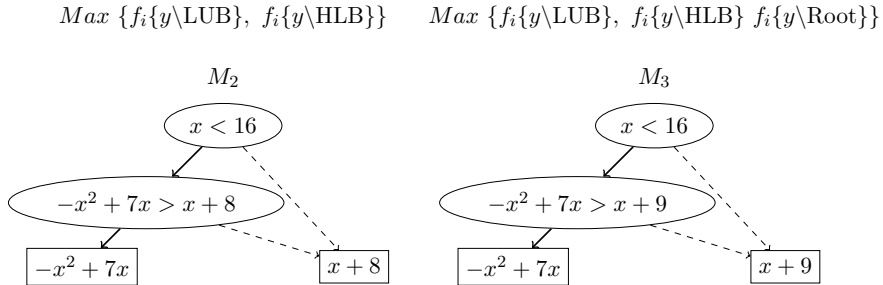


Figure 11: Step 5 of XADD Continuous Maximization for one case partition: Compute casemax of the substitutions result.

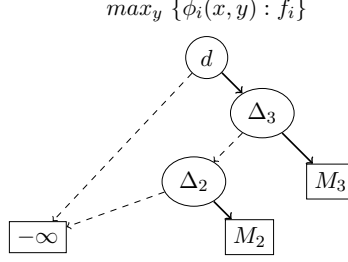


Figure 12: Step 6 of XADD Continuous Maximization for one case partition: Computing the final result as a new case partition Equation 12.

Algorithm 8: CHOOSEEARLIESTDEC(F^1, F^2) \longrightarrow $\langle F_{dec} \rangle$

```

1 begin
2   //select the decision to branch based on the order
3   if  $F^1$  is a terminal node then
4     | return  $F_{dec}^2$ 
5   if  $F^2$  is a terminal node then
6     | return  $F_{dec}^1$ 
7   if  $Order(F_{dec}^1) < Order(F_{dec}^2)$  then
8     | return  $F_{dec}^1$ 
9   else
10    | return  $F_{dec}^2$ 
11
12 end
    
```

4.2.3 BINARY XADD OPERATIONS: THE APPLY ALGORITHM

For *all* binary operations, the function $Apply(F^1, F^2, op)$ (Algorithm 9) computes the resulting XADD. The inputs of the algorithm are: two reduced XADD operands F^1 and F^2 and a binary operator $op \in \{\oplus, \ominus, \otimes, \max, \min\}$. The output F^r is a reduced XADD after applying the binary operation.

If the input of Apply algorithm is one of the special cases described in Table 1 (*ComputeResult* method), the result can be immediately computed (without recursion). Otherwise it checks the *ApplyCache* for any previously stored result (Line 6). If there is not a cache hit, the earliest decision in the XADD ordering is chosen to branch according to Algorithm 8. Two recursive *Apply* calls are then made on the branches of this decision to compute F_l and F_h (Line 26 and 27). Finally *GetNode* checks for any repeated branches in F^r before storing it in the cache and returning the resulting XADD. This algorithm can be described in details using the following steps:

- **Terminal computation:** The function *ComputeResult* called in Line 2 determines if the result can be computed without recursion using the pruning optimization described in Table 1. For the discrete maximization (minimization) operation, for two leaf

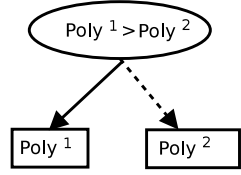
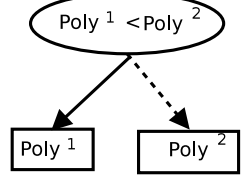
Input Case	Result
$F^1 \text{ op } F^2; F^1 = Poly^1; F^2 = Poly^2$	$Poly^1 \text{ op } Poly^2$
$F \oplus 0$ or $0 \oplus F$	F
$F \ominus 0$	F
$F \otimes 1$ or $1 \otimes F$	F
$F \otimes 0$ or $0 \otimes F$	0
$F \oplus \infty$ or $\infty \oplus F$	∞
$F \otimes +\infty; F > 0$ or $F \otimes -\infty; F < 0$	$+\infty$
$F \otimes +\infty; F < 0$ or $F \otimes -\infty; F > 0$	$-\infty$
$\text{casemax}(F, +\infty)$ or $\text{casemax}(+\infty, F)$	$+\infty$
$\text{casemax}(F, -\infty)$ or $\text{casemax}(-\infty, F)$	F
$\text{casemin}(F, +\infty)$ or $\text{casemin}(+\infty, F)$	F
$\text{casemin}(F, -\infty)$ or $\text{casemin}(-\infty, F)$	$-\infty$
$\text{casemax}(F^1, F^2), F^1 = Poly^1 \text{ and } F^2 = Poly^2$	
$\text{casemin}(F^1, F^2), F^1 = Poly^1 \text{ and } F^2 = Poly^2$	
other	<i>null</i>

Table 1: *ComputeResult* method: Special input cases and immediate results used in the algorithm *Apply* (Algorithm 9).

nodes $poly^1$ and $poly^2$ an additional decision node $poly^1 > poly^2$ ($poly^1 < poly^2$) is introduced to represent the maximum (minimum). Note that, this may cause out-of-order decisions which will be reordered by Algorithm 6.

- **Caching:** If the result of *ComputeResult* is null, we check the *ApplyCache* in Line 6 for any previously computed operation using these operands and operation. 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 (Lines 10-28). Since the *ComputeResult* takes care of the case when both operands are terminal nodes, we know that at least one of them is a decision node so one of the following conditions applies:

Algorithm 9: $\text{APPLY}(F^1, F^2, op) \rightarrow \langle F^r \rangle$

```

1 begin
2    $F^r \leftarrow \text{COMPUTERESULT}(F^1, F^2, op)$ 
3   //check if the result can be immediately computed in Table 1
4   if  $F^r \neq \text{null}$  then
5     | return  $F^r$ 
6    $F^r \leftarrow \text{ApplyCache}(\langle F^1, F^2, op \rangle)$ 
7   //check if we previously computed the same operation
8   if  $F^r \neq \text{null}$  then
9     | return  $F^r$ 
10  //choose decision to branch
11   $F_{dec} \leftarrow \text{CHOOSEEARLIESTDEC}(F^1, F^2)$ 
12  //set up nodes for recursion
13  if  $F^1$  is non-terminal  $\wedge F_{dec} = F_{dec}^1$  then
14    |  $F_l^{v1} \leftarrow F_l^1$ 
15    |  $F_h^{v1} \leftarrow F_h^1$ 
16  else
17    |  $F_l^{v1} \leftarrow F^1$ 
18    |  $F_h^{v1} \leftarrow F_1$ 
19  if  $F^2$  is non-terminal  $\wedge F_{dec} = F_{dec}^2$  then
20    |  $F_l^{v2} \leftarrow F_l^2$ 
21    |  $F_h^{v2} \leftarrow F_h^2$ 
22  else
23    |  $F_l^{v2} \leftarrow F^2$ 
24    |  $F_h^{v2} \leftarrow F_2$ 
25  //use recursion to compute true and false branches for resulting XADD
26   $F_l \leftarrow \text{Apply}(F_l^{v1}, F_l^{v2}, op)$ 
27   $F_h \leftarrow \text{Apply}(F_h^{v1}, F_h^{v2}, op)$ 
28   $F^r \leftarrow \text{GETNODE}(F_{dec}, F_h, F_l)$ 
29  //Use Algorithm 6 to reorder decisions if necessary
30  if  $op = \text{casemax}$  or  $op = \text{casemin}$  then
31    |  $F^r \leftarrow \text{REORDER}(F^r)$ 
32  //save the result to reuse in the future
33   $\text{ApplyCache}(\langle F^1, F^2, op \rangle) \leftarrow F^r$ 
34  return  $F^r$ 
35 end

```

- F^1 or F^2 is a terminal node or $F_{dec}^1 \neq F_{dec}^2$: One of the decision nodes is chosen (e.g. F^2) by *ChooseEarliestDec* method and two recursive *Apply* calls are performed using its corresponding branches (F_h^2 and F_l^2) with the other operand (e.g., F^1). The result of these two *Apply* functions is used with the decision F_{dec}^2

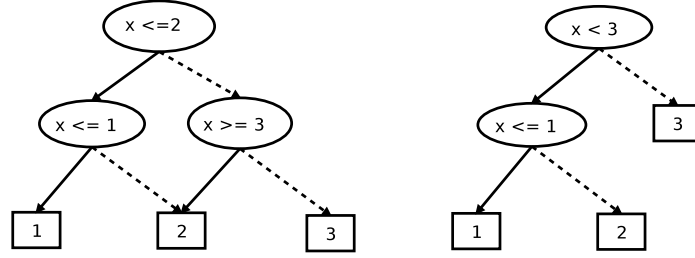


Figure 13: (left) A counterexample for an XADD that is not canonical after applying consistency and redundancy checking; (right) The minimal XADD which can not be derived from the left diagram.

for a reduced result:

$$\begin{aligned}
 F_h &\leftarrow \text{Apply}(F^1, F_h^2, \text{op}) \\
 F_l &\leftarrow \text{Apply}(F^1, F_l^2, \text{op}) \\
 F^r &\leftarrow \text{GetNode}(F_{dec}^2, F_h, F_l)
 \end{aligned}$$

- F^1 and F^2 are decision nodes and $F_{dec}^1 = F_{dec}^2$: Since the decisions are equal, the final result is a decision with the constraint $F_{dec}^1 (= F_{dec}^2)$; the high branch is obtained by calling *Apply* on F_h^1 and F_h^2 and the low branch is the result of the *Apply* function on F_l^1 and F_l^2 , i.e.:

$$\begin{aligned}
 F_h &\leftarrow \text{Apply}(F_h^1, F_h^2, \text{op}) \\
 F_l &\leftarrow \text{Apply}(F_l^1, F_l^2, \text{op}) \\
 F^r &\leftarrow \text{GetNode}(F_{dec}^1, F_h, F_l)
 \end{aligned}$$

Finally the result is a reduced XADD returned by *GetNode* where the decisions are ordered by algorithm *Reorder* if the operation is *op=casemax* or *op=casemin*.

4.3 Inconsistency and Redundancy Pruning

One issue with applying operations on XADDs is the introduction of redundant nodes or nodes that result in inconsistent path formula (Section 4.1). 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 which result in unnecessary nodes. Thus, removing redundant nodes from the XADD is also a requirement for obtaining a minimal XADD representation.

In this section we provide efficient algorithms to prune linear XADDs (currently we don't have efficient algorithms to prune XADDs with nonlinear decisions).

4.3.1 INCONSISTENCY PRUNING ALGORITHM

Given a linear XADD F with possibly inconsistent paths and the set of decision constraints on a path ψ , the output of *PruneInconsistent* (Algorithm 10) is a reduced XADD with

Algorithm 10: $\text{PRUNEINCONSISTENT}(F, \psi) \longrightarrow \langle F^r \rangle$

```

1 begin
2   //  $F$  is the root node represented as  $(F_{dec}, F_h, F_l)$ 
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  $F \in \{0, 1\}^m$  then
7      $low \leftarrow \text{PRUNEINCONSISTENT}(F_l, \psi)$ 
8      $high \leftarrow \text{PRUNEINCONSISTENT}(F_h, \psi)$ 
9     return  $\text{GETNODE}(F_{dec}, high, low)$ 
10  //else  $F$  is a linear decision check if  $\psi \models \perp$ 
11  if  $\text{TESTIMPLIED}(\psi, F_{dec})$  then
12    return  $\text{PRUNEINCONSISTENT}(F_h, \psi)$ 
13  if  $\text{TESTIMPLIED}(\psi, \neg F_{dec})$  then
14    return  $\text{PRUNEINCONSISTENT}(F_l, \psi)$ 
15  //result of TestImplied was false then prune both branches
16   $low \leftarrow \text{PRUNEINCONSISTENT}(F_l, \psi \wedge \neg F_{dec})$ 
17   $high \leftarrow \text{PRUNEINCONSISTENT}(F_h, \psi \wedge F_{dec})$ 
18  // return the new XADD with the pruned branches
19  return  $\text{GETNODE}(F_{dec}, high, low)$ 
20 end

```

Algorithm 11: $\text{PRUNEREDUNDANT}(F, \psi) \longrightarrow \langle F^r \rangle$

```

1 begin
2   //  $F$  is the root node represented as  $(F_{dec}, F_h, F_l)$ 
3   //check if high branch is implied in the low branch for  $F_{dec}$ 
4    $isRedundant = \text{ISREDUNDANT}(\psi \wedge F_{dec}, F_l, F_h)$ 
5   if  $isRedundant$  then
6     return  $\text{PRUNEREDUNDANT}(F_l, \psi)$ 
7   //check if low branch is implied in the high branch for  $\neg F_{dec}$ 
8    $isRedundant = \text{ISREDUNDANT}(\psi \wedge \neg F_{dec}, F_h, F_l)$ 
9   if  $isRedundant$  then
10    return  $\text{PRUNEREDUNDANT}(F_h, \psi)$ 
11  //return the new XADD with the pruned branches
12   $low \leftarrow \text{PRUNEREDUNDANT}(F_l, \psi \wedge \neg F_{dec})$ 
13   $high \leftarrow \text{PRUNEREDUNDANT}(F_h, \psi \wedge F_{dec})$ 
14  return  $\text{GETNODE}(F_{dec}, high, low)$ 
15 end

```

canonical leaves, linear decisions and no unreachable nodes. Similar to Algorithm *Reduce-XADD*, this algorithm is of a recursive bottom-up nature.

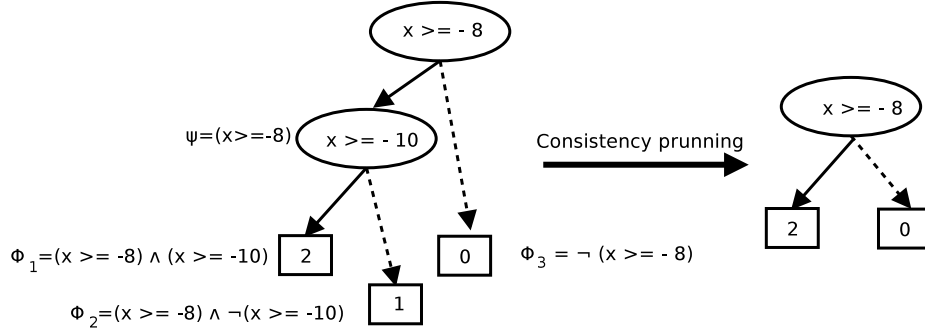


Figure 14: XADD with an Inconsistent Path Formula: *(left)* Path definitions on each node in the XADD with an inconsistent path formula $\psi(n_1, n_2, n_4)$, i.e. $((x \geq -8) \wedge \neg(x \geq -10)) \models \perp$; *(right)* Pruned XADD after removing path $C = (n_1, n_2, n_4)$. Note that n_4 is unreachable and then removed; consequently n_2 is an unnecessary decision and it is also removed.

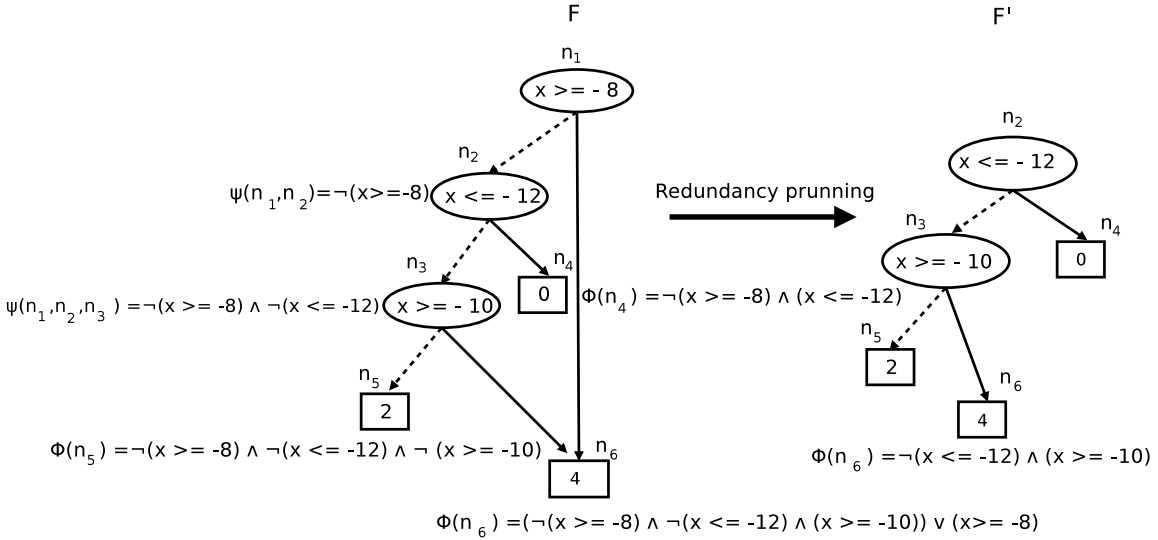


Figure 15: XADD with a Redundant Node: *(left)* Path definitions on each node in the XADD with a redundant node n_1 ($x \geq -8$). It is redundant since n_2 represents the same function as n_1 ; *(right)* The pruned XADD F without the redundant node n_1 . Note that all leaf nodes n_i have the same partition, i.e. $\phi(n_i) \equiv \phi'(n_i)$.

Lines 3–4 of Algorithm 10 return 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 *GetNode*.

Algorithm 12: TESTIMPLIED(ψ, F_{dec})

```

1 begin
2    $\psi \leftarrow \psi \wedge \neg F_{dec}$ 
3    $inconsistent \leftarrow \text{ImplicationCache}(\psi)$ 
4   if  $inconsistent \neq null$  then
5     return true
6    $consistent \leftarrow \text{NonImplicationCache}(\psi)$ 
7   if  $consistent \neq null$  then
8     return false
9   if  $LPSolve.IsFEASIBLE(\psi)$  then
10     $\text{NonImplicationCache}(\psi) \leftarrow true$ 
11    return false
12  else
13     $\text{ImplicationCache}(\psi) \leftarrow true$ 
14    return true
15
16 end

```

If the two mentioned conditions are false, then the current node of F is a linear decision node. In this case, we check if the decision test of this node (F_{dec}) is consistent with the previous constraints on path ψ . Algorithm 10, Lines 11 and 13 use the *TestImplied* function (Algorithm 12) that has a path formula ψ and current decision test F_{dec} as inputs. This function performs the following steps:

- Check if the implication result of ψ exists in an *Implications* cache. This cache stores path formulas ψ that are inconsistent. If there is a previous result which already contains the new decision constraint F_{dec} then return *true*.
- Similarly check the *non-Implications* cache for consistent path formulas ψ . If there is a previous result which already contains the new decision constraint F_{dec} then return *false*.
- If there is no cache hit, then F_{dec} is added to the path: $\psi \leftarrow \psi \wedge \neg F_{dec}$. This allows us to check for the infeasibility property.
- The LP-solver is called on the set of decision constraints in ψ . The result of the LP-solver determines infeasibility with respect to the new decision just added to ψ .
- The path ψ 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 ψ is added to *Implications*, else it is added to *non-Implications*.
- TestImplied returns the result of the LP-solver.

Line 11 of Algorithm 10 checks if the true assignment of F_{dec} is implied by ϕ . If *TestImplied* is *true*, then the low branch is inconsistent and the algorithm returns the high

Algorithm 13: $\text{ISREDUNDANT}(\psi, subtree, goal) \rightarrow \langle true, false \rangle$

```

1 begin
2   if  $subtree = goal$  then
3     return  $true$ 
4   // a terminal node can not achieve goal
5   if  $subtree$  is a terminal node then
6     return  $false$ 
7   if  $goal$  is non-terminal node then
8     // node is not redundant if the goal occurs before subtree in the XADD
      order
9     if  $Order(subtree_{dec}) \geq Order(goal_{dec})$  then
10      return  $false$ 
11
12  // check if  $subtree_{dec}$  is implied by  $\psi$  to prune search
13  if  $\text{TESTIMPLIED}(\psi, \neg subtree_{dec})$  then
14    return  $\text{ISREDUNDANT}(\psi, subtree_l, goal)$ 
15  if  $\text{TESTIMPLIED}(\psi, subtree_{dec})$  then
16    return  $\text{ISREDUNDANT}(\psi, subtree_h, goal)$ 
17  // result of TestImplied was false
18   $isImplied \leftarrow \text{ISREDUNDANT}(\psi \wedge \neg subtree_{dec}, subtree_l, goal)$ 
19  // if the low branch does not imply goal, the node is not redundant
20  if  $\neg isImplied$  then
21    return  $false$ 
22   $isImplied \leftarrow \text{ISREDUNDANT}(\psi \wedge subtree_{dec}, subtree_h, goal)$ 
23  return  $isImplied$ 
24 end

```

branch as the result of this pruning (Line 12). Similarly in Line 13 $\neg F_{dec}$ is checked and if $TestImplied$ is *true* then the high branch is inconsistent and the low branch returned after pruning (Line 14).

In case the current decision node is not an inconsistent path, both low and high branches need to be checked for inconsistency. Line 16 adds the decision constraint $\neg F_{dec}$ to ψ and call *PruneInconsistent* for F_l . Line 17 performs the same for the high branch (F_h) adding F_{dec} . At this stage the current subtree with the computed low and high branches does not contain any inconsistent paths. Thus the algorithm returns a reduced node computed using *GetNode* in Line 19.

4.3.2 REDUNDANCY PRUNING ALGORITHM

The algorithm *isRedundant* directly checks if a node F can be replaced by the *subtree* branch. It receives a path formula ψ and two XADD nodes, *subtree* and *goal*, and checks if the node *goal* is necessarily achieved in the XADD *subtree* when ψ is true. If the goal node is not achieved in some branch of *subtree*, then the replacement can not be done.

Initially Lines 2-3 of Algorithm 13 check if *subtree* and *goal* are equal, in this case the node F is redundant and it can be replaced by *subtree*, therefore the algorithm returns *true*. If the nodes are different then the node F is not redundant and Line 6 returns *false*. Next Lines 7–10 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* can not be achieved in *subtree*.

If neither of the above cases occur, the algorithm must search a goal node in the branches of *subtree*. In Lines 12-16 the method *TestImplied* is used to check if the path formula ψ implies *subtree_{dec}* to restrict the search in one of the branches.

Similar to Algorithm 10, if the result of *TestImplied* is false, both low and high branches need to be checked for redundancy. Line 18 calls the method *IsRedundant* for *subtree_l* and the path formula $\psi \wedge \neg \text{subtree}_{dec}$. Similarly, Line 22 calls *IsRedundant* for *subtree_h* and the path formula $\psi \wedge \text{subtree}_{dec}$. If any of the branches along the way do not imply *goal*, then the node is not redundant and the algorithm returns false.

The method *PruneRedundant* (Algorithm 11) removes any redundant nodes from an XADD F and its corresponding path formula (the path from the root to node F). According to the definition, a node is redundant if it represents the same function as one of its branches. For a *subtree* be used to replace a father node, it must contain its sibling node to achieve the same values in the corresponding regions. In Line 4 of Algorithm 11, the method checks if the node F_l can replace the node F . If *isRedundant* returns true, *PruneRedundant* returns the low branch, after a recursive pruning (Line 6); else we check if F_h can replace the node F (Line 8) and, again, if successful returns a high branch after a recursive pruning (Line 10). If the current node is not redundant, we recursively prune both branches (Line 12-13) and return the new node (Line 14).

Figure 15 shows that the node $x \geq -8$ in the left (where n_2 is the subtree and n_6 is the goal node in the successful call of the *isRedundant* method) is redundant and can be removed from the original XADD.

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 have defined, reduced XADDs are derived from the result of the applying redundant branch pruning. However unlike BDDs and ADDs a canonical XADD may not be produced after such pruning approaches. As a counterexample consider the simple XADD in Figure 13 (left) where the values are as follows:

$$\begin{cases} x \leq 1 : & 1 \\ x \geq 3 : & 3 \\ 1 < x < 3 : & 2 \end{cases} \quad (20)$$

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 XADD is not inconsistent and further the node $x < 2$ can not be removed using the redundant technique which replaces a redundant node with one of its branches. This type of redundant node can be removed with the redundant technique by reordering the decisions in the 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 13 is the minimal XADD for the function described in Equation 20. Thus a challenging open problem is the proof of XADD minimality with respect to a given order of the linear decisions.

Having defined the efficient representation of XADDs, next we show results from implementing an SDP algorithm using XADDs.

5. Experimental Results

We implemented two versions of our proposed SVI algorithms using XADDs one for DA-HMDP and other for CA-HMDP. For CA-HMDPs we evaluated SVI on a didactic quadratic MARS ROVER domain and two problems from the INVENTORY CONTROL domain defined in the introduction, and RESERVOIR MANAGEMENT domain. For comparison purposes, these domains are discretized by their action space to generate DA-HMDP.¹¹

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 $\Delta x \in [-10, 10]$. 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 $tp \in \{0, 1\}$ to indicate if the picture has already been taken ($tp = 1$), x' and tp' to denote post-action state, and R to denote reward, we express the MARS ROVER CA-HMDP using piecewise dynamics and reward:

$$\begin{aligned} P(tp'=1|x, tp) &= \begin{cases} tp \vee (x \geq -2 \wedge x \leq 2) : & 1.0 \\ \neg tp \wedge (x < -2 \vee x > 2) : & 0.0 \end{cases} \\ P(x'|x, \Delta x) &= \delta \left(x' - \begin{cases} \Delta x \geq -10 \wedge \Delta x \leq 10 : & x + \Delta x \\ \Delta x < -10 \vee \Delta x > 10 : & x \end{cases} \right) \\ R(x, tp) &= \begin{cases} \neg tp \wedge x \geq -2 \wedge x \leq 2 : & 40 - x^2 \\ tp \vee x < -2 \vee x > 2 : & -1 \end{cases} \end{aligned}$$

The maximum long-term *value* V from a given state in MARS ROVER is defined as a function of state variables:

11. 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 <https://github.com/ssanner/xadd-inference>.

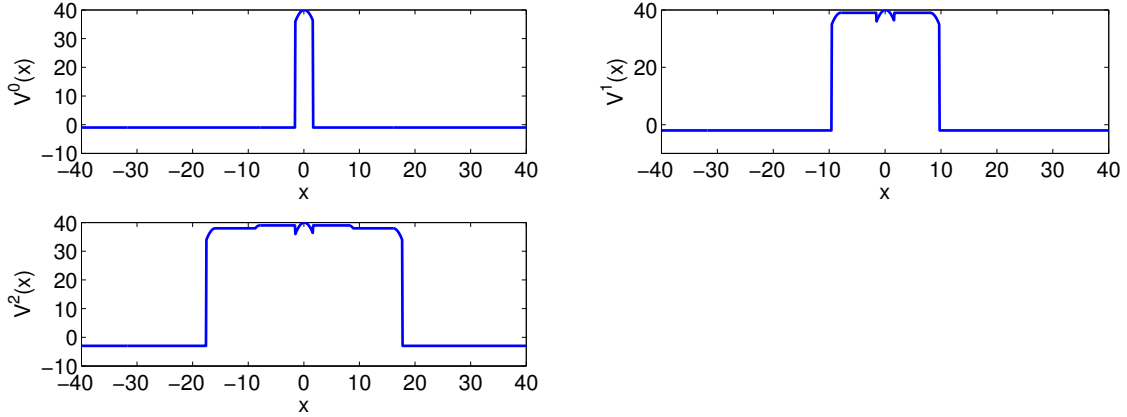


Figure 16: Optimal sum of rewards (value) $V^t(x)$ for $tp = 0$ (*false*) for time horizons (i.e., decision stages remaining) $t = 0$, $t = 1$, and $t = 2$ on the CONTINUOUS ACTION MARS ROVER problem. For $x \in [-2, 2]$, the rover automatically takes a picture and receives a reward quadratic in x . We initialized $V^0(x, tp) = R(x, tp)$; for $V^1(x)$, the rover achieves non-zero value up to $x = \pm 12$ and for $V^2(x)$, up to $x = \pm 22$.

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

The value function is piecewise and non-linear, and contains non-rectangular decision boundaries like $4 - x^2 - y^2 \geq 0$. Figure 16 presents the 0-step, 1-step, 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-drain*(e)); we assume deterministic rainfall replenishment and present the reward function as:

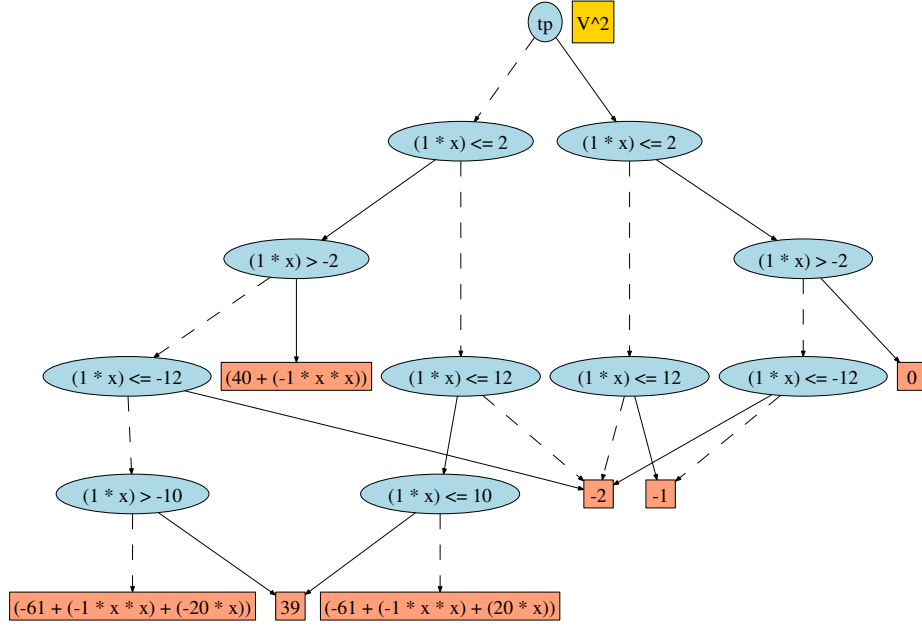


Figure 17: Optimal value function $V^2(x)$ for the CONTINUOUS ACTION MARS ROVER problem represented as an (XADD) equal to the second diagram on the left. To evaluate $V^2(x)$ for any state x , one simply traverses the diagram in a decision-tree like fashion until a leaf is reached where the non-parenthetical expression provides the *optimal value* and the parenthetical expression provides the *optimal policy* ($y = \pi^{*,2}(x)$) to achieve value $V^2(x)$.

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

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.

$$\begin{aligned} l'_1 &= (400 * e + l_1 - 700 * e + 500 * e) \\ l'_2 &= (400 * e + l_2 - 500 * e) \end{aligned}$$

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.

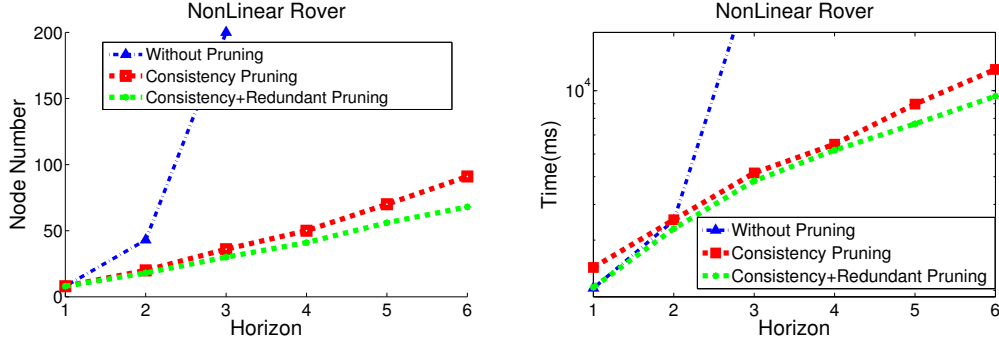


Figure 18: Space (# XADD nodes in value function) and time for different iterations (horizons) of SDP on quadratic CONTINUOUS ACTION MARS ROVER with 4 different results based on pruning techniques. Results are shown for the XADD with no pruning technique, with only consistency checking (using LP-solver) and with both the consistency and redundancy checking with a numerical precision heuristic.

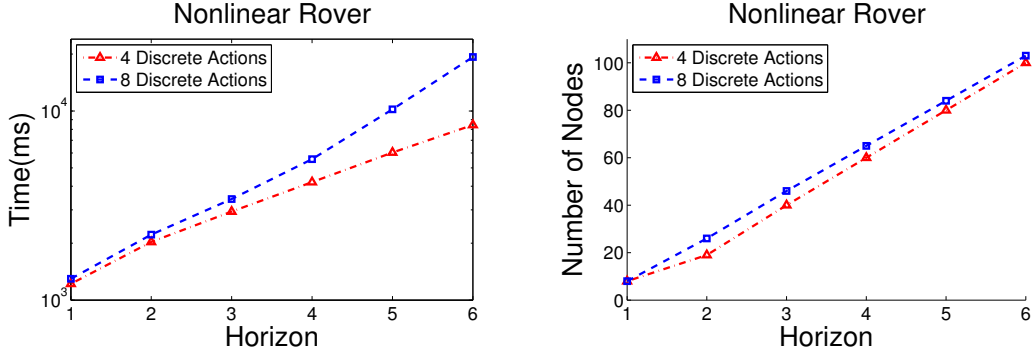


Figure 19: Space and elapsed time vs. horizon for a fixed discretization of 4 and 8 for the MARS ROVER domain. This demonstrates how the number of nodes and time increases for each horizon.

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 18. Here three evaluations are performed based on the pruning algorithms of Section ???. We note that without the inconsistency checking of Algorithm 10, 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

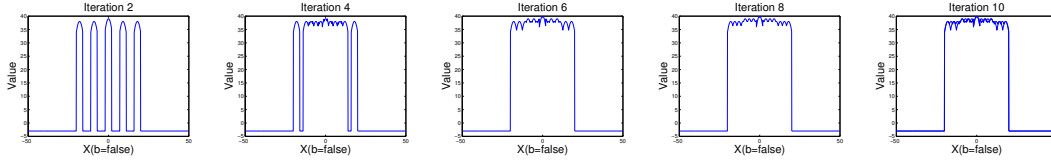


Figure 20: V^3 for different number of DISCRETE ACTIONS in the MARS ROVER problem. As the number of discrete actions increases, the value function more closely resembles the continuous action value of V^2 (defined in Figure 17).

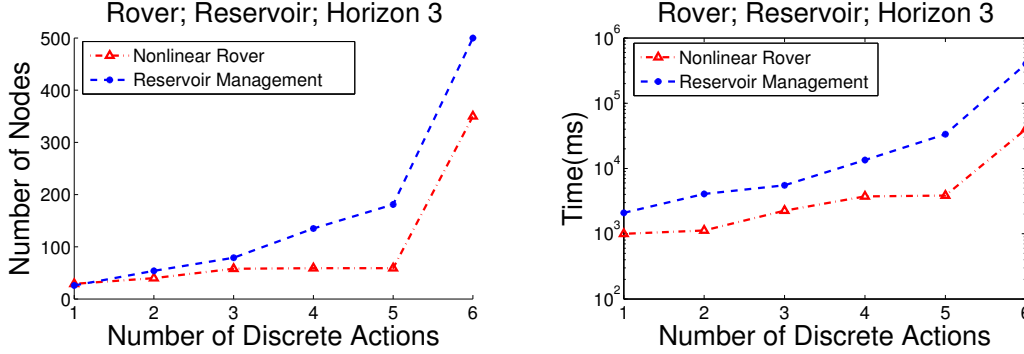


Figure 21: Space and elapsed time vs. number of discrete actions for a fixed horizon of 3 for MARS ROVER and the RESERVOIR MANAGEMENT with discrete actions.

that in some domains using redundancy pruning is 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 19 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 20 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 21 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 22 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. 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

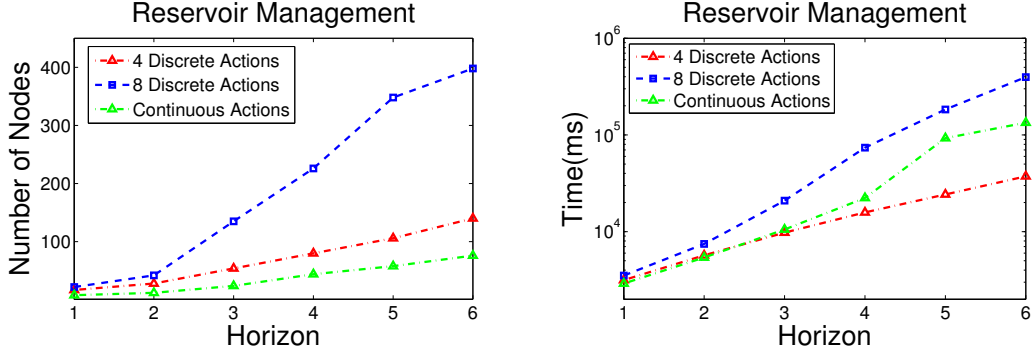


Figure 22: Space and elapsed time vs. horizon for a fixed discretization of 4 and 8 actions for the RESERVOIR MANAGEMENT .

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 23 (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 23 (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 24, 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 25 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 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 26 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

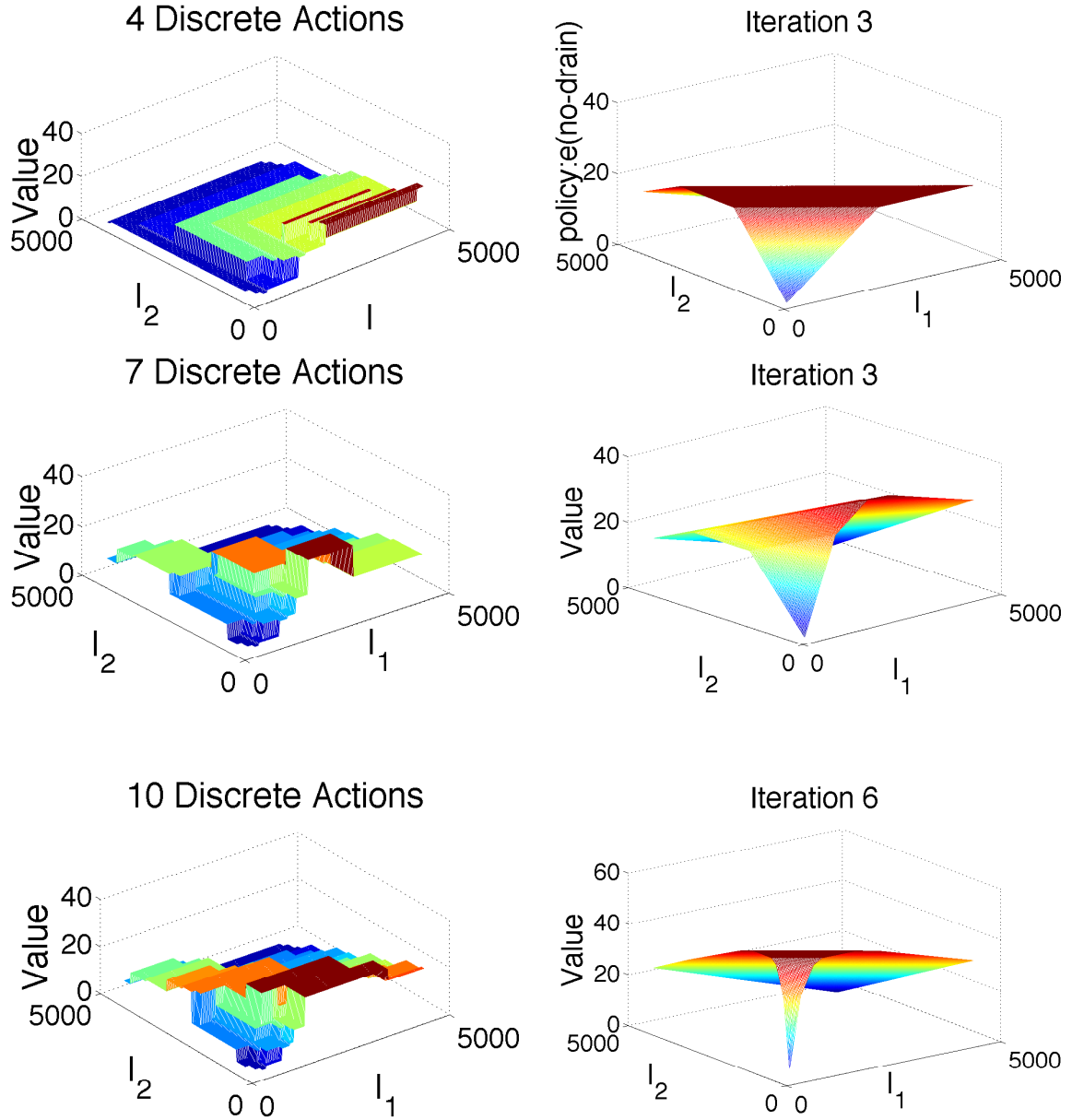


Figure 23: Results for the DISCRETE ACTION and CONTINUOUS ACTION of the RESERVOIR MANAGEMENT problem. (left) 4, 7 and 10 DISCRETE ACTIONS of the RESERVOIR MANAGEMENT problem for different water levels in iteration V^3 . Each discretization draws the value closer to that of the continuous action RESERVOIR 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 $V^6(l_1, l_2)$.

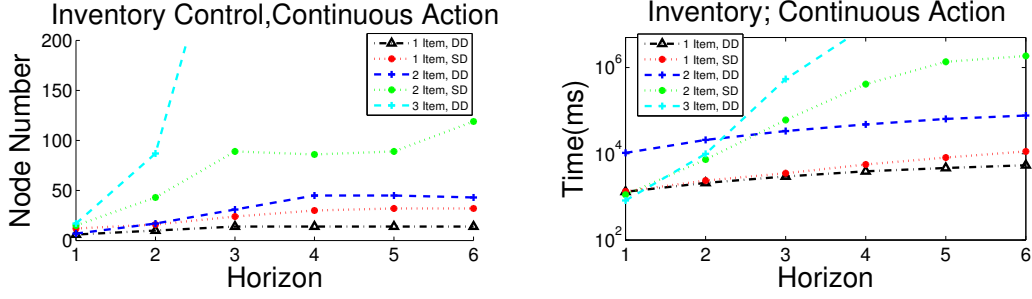


Figure 24: CONTINUOUS ACTION INVENTORY CONTROL : space and time vs. horizon.

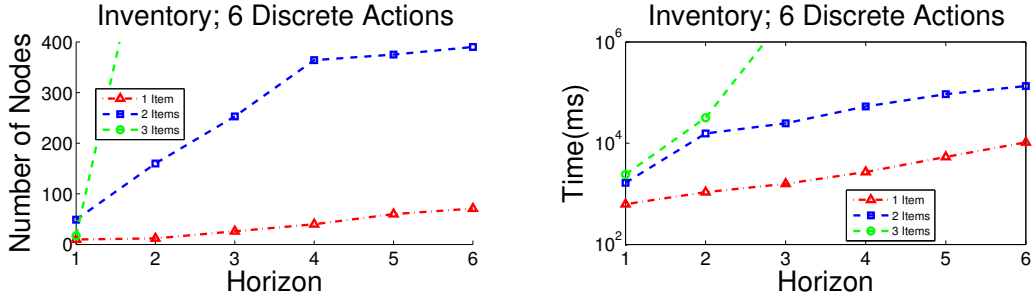


Figure 25: DISCRETE ACTION INVENTORY CONTROL : Space and time vs. horizon for 6 discrete actions. Results are shown for various continuous items in the inventory.

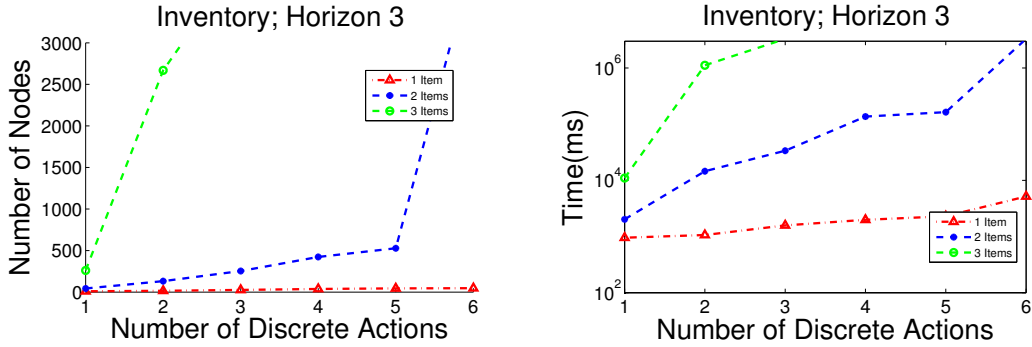


Figure 26: Space and elapsed time vs. horizon for a fixed horizon of 3 and different problem sizes for the DISCRETE ACTION INVENTORY CONTROL .

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.

Verificar:

Note that, for arbitrary functions, finding a canonical expression form may be time consuming or intractable.

Theoretically, this does not affect our solution but in practice if we have non linear decision nodes, we can't prune redundant nodes or inconsistency paths resulting in large XADDs.

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.
- 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.