Artificial Intelligence 175 (2011) 1498–1527

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Efficient solutions to factored MDPs with imprecise transition probabilities

Karina Valdivia Delgado[a,*], Scott Sanner[b], Leliane Nunes de Barros[a,c]

[a] *EACH, Universidade de São Paulo, Av. Arlindo Béttio, 1000 – Ermelino Matarazzo São Paulo – SP, Brazil*
[b] *NICTA and the Australian National University, Canberra, ACT 2601, Australia*
[c] *IME, Universidade de São Paulo, Rua de Matão, 1010 – Cidade Universitária São Paulo – SP, Brazil*

ARTICLE INFO

Article history:
Received 3 December 2009
Received in revised form 31 December 2010
Accepted 31 December 2010
Available online 4 January 2011

Keywords:
Probabilistic planning
Markov Decision Process
Robust planning

ABSTRACT

When modeling real-world decision-theoretic planning problems in the Markov Decision Process (MDP) framework, it is often impossible to obtain a completely accurate estimate of transition probabilities. For example, natural uncertainty arises in the transition specification due to elicitation of MDP transition models from an expert or estimation from data, or non-stationary transition distributions arising from insufficient state knowledge. In the interest of obtaining the most robust policy under transition uncertainty, the Markov Decision Process with Imprecise Transition Probabilities (MDP-IPs) has been introduced to model such scenarios. Unfortunately, while various solution algorithms exist for MDP-IPs, they often require external calls to optimization routines and thus can be extremely time-consuming in practice. To address this deficiency, we introduce the *factored* MDP-IP and propose efficient dynamic programming methods to exploit its structure. Noting that the key computational bottleneck in the solution of factored MDP-IPs is the need to repeatedly solve nonlinear constrained optimization problems, we show how to target approximation techniques to drastically reduce the computational overhead of the nonlinear solver while producing bounded, approximately optimal solutions. Our results show up to two orders of magnitude speedup in comparison to traditional “flat” dynamic programming approaches and up to an order of magnitude speedup over the extension of factored MDP approximate value iteration techniques to MDP-IPs while producing the lowest error of any approximation algorithm evaluated.

1. Introduction

Markov Decision Processes (MDP) [1] have become the *de facto* standard model for decision-theoretic planning problems and a great deal of research in recent years has aimed to exploit structure in order to compactly represent and efficiently solve factored MDPs [2–5]. However, in many real-world problems, it is simply impossible to obtain a precise representation of the transition probabilities in an MDP. This may occur for many reasons, including (a) imprecise or conflicting elicitations from experts, (b) insufficient data from which to estimate reliable precise transition models, or (c) non-stationary transition probabilities due to insufficient state information.

For example, in an MDP for traffic light control, it is difficult to estimate the turn probabilities for each traffic lane that has the option of going straight or turning. These lane-turning probabilities may change during the day or throughout the year, as a function of traffic at other intersections, and based on holidays and special events; in general it is impossible to

[*] Corresponding author. Tels.: +55 11 73326036, +55 11 30919878; fax: +55 11 30916134.
E-mail addresses: kvd@usp.br (K.V. Delgado), ssanner@nicta.com.au (S. Sanner), leliane@ime.usp.br (L.N. de Barros).

0004-3702/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2011.01.001

accurately model all of these complex dependencies. In this case it would be ideal to have a traffic control policy optimized over a range of turn probabilities in order to be robust to inherent non-stationarity in the turn probabilities.

To accommodate optimal models of sequential decision-making in the presence of strict uncertainty over the transition model, the MDP with imprecise transition probabilities (MDP-IP) was introduced [6,7]. While the MDP-IP poses a robust framework for the real-world application of decision-theoretic planning, its general solution requires the use of computationally expensive optimization routines that are extremely time-consuming in practice.

To address this computational deficiency, we extend the factored MDP model to MDP-IPs by proposing to replace the usual Dynamic Bayes Net (DBN) [8] used in factored MDPs with Dynamic Credal Nets (DCNs) [9] to support compact factored structure in the imprecise transition model of factored MDP-IPs. Then we propose efficient, scalable algorithms for solving these factored MDP-IPs. This leads to the following novel contributions in this work:

- We introduce the parameterized ADD (PADD) with polynomial expressions at its leaves and explain how to extend ADD properties and operations to PADDs.
- We extend the decision-diagram based SPUDD and APRICODD algorithms for MDPs [3,4] to MDP-IP algorithms that exploit DCN structure via PADDs.
- As shown in our experimental evaluation, the generalization of SPUDD and APRICODD to MDP-IPs using PADDs is just the first step in obtaining efficient solutions. Observing that the key computational bottleneck in the solution of MDP-IPs is the need to repeatedly solve nonlinear constrained optimization problems, we show how to target our approximations to drastically reduce the computational overhead of the nonlinear solver while producing provably bounded, approximately optimal solutions.

As our results will demonstrate, using the above contributions we can obtain up to two orders of magnitude speedup in comparison to traditional “flat” dynamic programming approaches [6]. In addition, our best approximate factored MDP-IP solver yields an order of magnitude speedup over a direct generalization of state-of-the-art approximate factored MDP solvers [4] for factored MDP-IPs (also implemented in this work) and consistently produces the lowest error of all approximate solution algorithms evaluated.

2. Markov decision processes

Formally, an MDP is defined by the tuple $\mathcal{M} = \langle S, A, P, R, T, \gamma\rangle$, where [1,10]:

- S is a finite set of fully observable states;
- A is a finite set of actions;
- $P(s'|s,a)$ is the conditional probability of reaching state $s' \in S$ when action $a \in A$ is taken from state $s \in S$;
- $R : S \times A \to \mathbb{R}$ is a fixed reward function associated with every state and action;
- T is the time horizon (number of decision stages remaining) for decision-making;
- $\gamma = [0,1)$ is a *discount factor* (the reward obtained t stages into the future is discounted in the sense that it is multiplied by γ^t).

A stationary policy $\pi : S \to A$ indicates the action $a = \pi(s)$ to take in each state s (regardless of stage). The value of a stationary policy π is defined as the expected sum of discounted rewards over an infinite horizon ($|T| = \infty$) starting in state s_0 at stage 0 and following π

$$V_\pi(s) = E_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_t \,\Big|\, s_0 = s\right], \tag{1}$$

where R_t (abbreviation of $R_t(s_t, \pi(s_t))$) is the reward obtained at stage t when the agent is in state s_t and takes action $\pi(s_t)$. (1) can be decomposed and rewritten recursively based on the values of the possible successor states $s' \in \mathcal{S}$ as follows:

$$V_\pi(s) = R\big(s, \pi(s)\big) + \gamma \sum_{s' \in \mathcal{S}} P\big(s'|s, \pi(s)\big) V_\pi\big(s'\big). \tag{2}$$

Our objective is to find an optimal policy π^* that yields the maximal value in each state, i.e., $\forall s, \pi' V_{\pi^*}(s) \geqslant V_{\pi'}(s)$.

A well-known algorithm to solve an MDP is *value iteration* [1]. For $t > 0$, it constructs a series of t-stage-to-go value functions V^t. Starting with arbitrary V^0, value iteration performs value updates for all states s, computing V^t based on V^{t-1}. The Q-value for state s and action a is:

$$Q^t(s,a) = R(s,a) + \gamma \sum_{s' \in S} P\big(s'|s,a\big) V^{t-1}\big(s'\big) \tag{3}$$

where the best value attainable at decision stage t and state s is

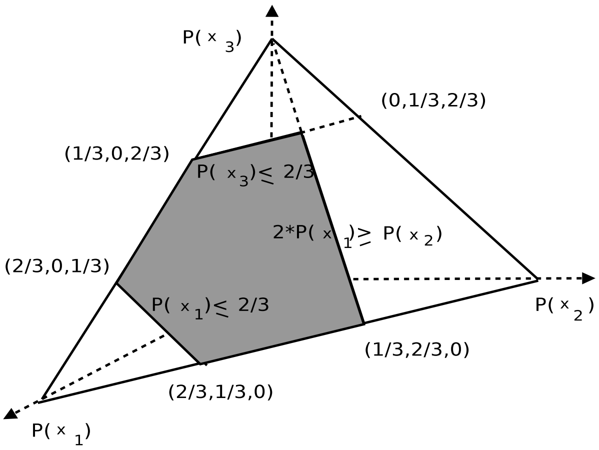


Fig. 1. A credal set example represented by the gray region. The credal set is defined by the triplets $\{P(x_1), P(x_2), P(x_3)\}$ that belong to this region.

$$V^t(s) = \max_{a \in A} Q^t(s,a). \tag{4}$$

We define the greedy policy π_V w.r.t. some V as follows:

$$\pi_V(s) = \arg\max_{a \in A}\left(R(s,a) + \gamma \sum_{s' \in S} P\big(s'|s,a\big)V\big(s'\big)\right). \tag{5}$$

At the infinite horizon, the value function provably converges

$$\lim_{t \to \infty} \max_s \big|V^t(s) - V^{t-1}(s)\big| = 0 \tag{6}$$

leading to a stationary, deterministic optimal policy $\pi^* = \pi_{V^\infty}$ [1]. For practical MDP solutions, we are often only concerned with ϵ-optimality. If we terminate the MDP when the following condition is met:

$$\max_s \big|V^t(s) - V^{t-1}(s)\big| < \frac{\epsilon(1-\gamma)}{2\gamma} \tag{7}$$

then we guarantee that the greedy policy π_{V^t} loses no more than ϵ in value over an infinite horizon in comparison to π^* [1].

3. MDPs with imprecise transitions

As described in our introductory traffic example, it is often necessary to work with imprecise probabilities in order to represent incomplete, ambiguous or conflicting expert beliefs about transition probabilities. An *MDP with imprecise transition probabilities* (*MDP-IP*)[1] is specifically designed for this setting and is simply an extension of the MDP where the transition probabilities can be imprecisely specified. That is, instead of a probability measure $P(\cdot|s,a)$ over the state space S, we have a *set* of probability measures. For example, let $P(X)$ be the probability density function for $X = \{x_1, x_2, x_3\}$ defined with the following constraint set:

$$\begin{aligned} C = \big\{&P(x_1) \leqslant 2/3, \\ &P(x_3) \leqslant 2/3, \\ &2P(x_1) \geqslant P(x_2), \\ &P(x_1) + P(x_2) + P(x_3) = 1\big\}. \end{aligned} \tag{8}$$

The two-dimensional region of all probability measures that satisfy C is shown as the gray region in Fig. 1. This is referred to as a *credal set*, i.e., a set of probability measures (or a set of distributions for a random variable) [11]. We denote a credal set of distributions for variable X by $K(X)$.

Next we slightly specialize the definition of credal set to specify uncertainty in MDP-IP transition probabilities:

[1] The term MDP-IP was proposed by White III and Eldeib [7], while Satia and Lave Jr. [6] adopt instead the term *MDP with Uncertain Transition Probabilities*.

Definition 3.1 *(Transition credal set).* A credal set containing conditional distributions over the next state s', given a state s and an action a, is referred to as a *transition credal sets* [11] and denoted by $K(s'|s,a)$. Thus, we have $P(\cdot|s,a) \in K(\cdot|s,a)$ to define imprecisely specified transition probabilities.

We assume that all credal sets are closed and convex, an assumption that is often used in the literature, and that encompasses most practical applications [12]. We further assume stationarity for the transition credal sets $K(s'|s,a)$; that is, they do not depend on the stage t. While $K(s'|s,a)$ is non-stationary, we note that this does not require $P(s'|s,a)$ to be stationary in an MDP-IP: distributions $P(s'|s,a)$ may be selected from the corresponding credal sets in a time-dependent manner [13].

Formally, an MDP-IP is defined by $\mathcal{M}_{IP} = (S, A, K, R, T, \gamma)$. This definition is identical to the MDP $\mathcal{M}$, except that the transition distribution P is replaced with a transition credal set K. We will represent K implicitly as the set of transition probabilities consistent with a set of side linear inequality constraints C, like (8), over the probability parameters.

There are several optimization criteria that can be used to define the value of a policy in an MDP-IP. In the context of the discounted infinite horizon setting that we focus on in this work, there is always a deterministic stationary policy that is *maximin* optimal [6] (i.e., no other policy could achieve greater value under the assumption that Nature's selects $P(s'|s,a)$ adversarially to minimize value); moreover, given the assumption that A is finite and the credal set K is closed, this policy induces an optimal value function that is the unique fixed-point solution of

$$V^*(s) = \max_{a \in A} \min_{P \in K} \left\{ R(s,a) + \gamma \sum_{s' \in S} P\left(s'|s,a\right) V^*\left(s'\right) \right\}. \tag{9}$$

There are various algorithms for solving *flat* (i.e., enumerated state) MDP-IPs based on dynamic programming [6,7]. In this work, we build on a flat value iteration solution to MDP-IPs [6]:

$$V^t(s) = \max_{a \in A} \min_{P \in K} \left\{ R(s,a) + \gamma \sum_{s' \in S} P\left(s'|s,a\right) V^{t-1}\left(s'\right) \right\}. \tag{10}$$

Value iteration for MDP-IPs is the same as that given in (3) and (4) for MDPs except that now for *every* state s, we optimize our action choice $a \in A$ w.r.t. the *worst-case* distribution $P \in K$ that minimizes the future expected value. Thus we ensure that the resulting value function and policy are robust to the worst outcome that Nature could choose in light of the future value $V^{t-1}(s')$ that we expect to achieve.

As we noted before, Nature's *true* transition function P may be non-stationary; Nature can choose a *different* $P \in K$ for *every* action a and *every* state s and *every* decision stage t. As an example of such non-stationarity that may occur in practice, in the previously discussed traffic scenario, we observed that traffic turn probabilities may differ on holidays versus normal weekdays even though the embedded traffic controller may not be explicitly aware of the holiday in its state description. However, as long as such transition non-stationarity can be bounded by $P \in K$, convergence properties of MDP-IP value iteration in (10) *still* hold [13].

In [14,9] we have shown how MDP-IP solutions can be formulated as a bilevel or multilinear programming problem. In this paper we are interested in extending the dynamic programming solution for MDP-IPs [6,7] outlined above to efficiently solve problems with a factored state description, which we discuss next.

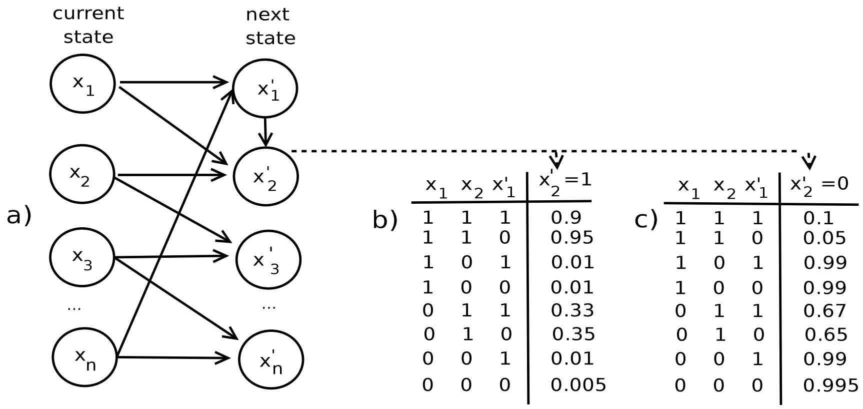
4. Factored MDP and MDP-IPs

4.1. Factored MDP

In many MDPs, it is often natural to think of the state as an assignment to multiple state variables and a transition function that compactly specifies the probabilistic dependence of variables in the next state on a subset of variables in the current state. Such an approach naturally leads us to define a *Factored MDP* [2], where $S = \{\vec{x}\}$. Here, $\vec{x} = (x_1, \ldots, x_n)$ where each state variable $x_i \in \{0,1\}$.[2]

The definition of actions $a \in A$ is unchanged between MDPs and factored MDPs, so the reward can simply be specified as $R(\vec{x}, a)$. The transition probabilities in a factored MDP are encoded using *Dynamic Bayesian Networks* (*DBNs*) [8]. A DBN is a *directed acyclic graph* (*DAG*) with two layers: one layer represents the variables in the current state and the other layer represents the next state (Fig. 2a). Nodes x_i and x_i' refer to the respective current and next state variables. The connection between these two layers defines the dependences between state variables w.r.t. the execution of an action $a \in A$. Directed edges are allowed *from* nodes in the first layer *into* the second layer, and also between nodes in the second layer (these latter edges are termed *synchronic arcs*). We denote by $pa_a(x_i')$ the parents of x_i' in the graph for action a. The graph encodes the standard Bayes net conditional independence assumption that a variable x_i' is conditionally independent of its nondescendants given its parents, which incidentally for a DBN also encodes the Markov assumption (the current state is

[2] While our extensions are not necessarily restricted to binary state variables, we make this restriction here for simplicity of notation.



b)

x_1	x_2	x'_1	$x'_2=1$
1	1	1	0.9
1	1	0	0.95
1	0	1	0.01
1	0	0	0.01
0	1	1	0.33
0	1	0	0.35
0	0	1	0.01
0	0	0	0.005

c)

x_1	x_2	x'_1	$x'_2=0$
1	1	1	0.1
1	1	0	0.05
1	0	1	0.99
1	0	0	0.99
0	1	1	0.67
0	1	0	0.65
0	0	1	0.99
0	0	0	0.995

Fig. 2. a) A Dynamic Bayesian Network (DBN) for an action a; b) conditional probability table for $x'_2=1$; c) conditional probability table for $x'_2=0$.

independent of the history given the previous state). The use of a DBN leads to the following factorization of transition probabilities:

$$P(\vec{x}'|\vec{x},a)=\prod_{i=1}^{n}P(x'_i|pa_a(x'_i),a). \tag{11}$$

Fig. 2(b) shows the conditional probability table (CPT) for $P(x'_2=1|pa_a(x'_2),a)$; Fig. 2(c) shows the same CPT for $x'_2=0$. The tables show all combinations of variable assignments for the parents of x'_2, i.e., $pa(x'_2)$; by definition, the sum of each row in Fig. 2(b) and Fig. 2(c) must be 1, which can be easily verified.

4.2. Factored MDP-IP

As our first major contribution, we extend the factored MDP representation [2] to compactly represent MDP-IPs. This simply requires modifying the DBN transition representation to account for uncertainty over the exact transition probabilities. Before we formally describe this transition function though, we first introduce one possible extension of the SysAdmin factored MDP to allow for imprecise transition probabilities, which we use from here out as a running example of a factored MDP-IP.

SysAdmin *domain [5].* In the SysAdmin domain we have n computers $c_1,\dots,c_n$ connected via different directed graph topologies: (a) unidirectional ring, (b) bidirectional ring and (c) independent bidirectional rings of pairs of computers (Fig. 3).

Let state variable x_i denote whether computer c_i is up and running ($x_i=1$) or not ($x_i=0$). Let $Conn(c_j,c_i)$ denote a connection from c_j to c_i. Formally, the CPTs for this domain have the following form:

$$P(x'_i=1|\vec{x},a)=\begin{cases}\text{(i)} & \textit{if } a=reboot(c_i)\textit{: then } 1\\ \text{(ii)} & \textit{if } a\neq reboot(c_i)\wedge x_i=1\textit{: then}\\ & p_{i1}\cdot\frac{|\{x_j|j\neq i\wedge x_j=1\wedge Conn(c_j,c_i)\}|+1}{|\{x_j|j\neq i\wedge Conn(c_j,c_i)\}|+1}\\ \text{(iii)} & \textit{if } a\neq reboot(c_i)\wedge x_i=0\textit{: then}\\ & p_{i2}\cdot\frac{|\{x_j|j\neq i\wedge x_j=1\wedge Conn(c_j,c_i)\}|+1}{|\{x_j|j\neq i\wedge Conn(c_j,c_i)\}|+1}\end{cases} \tag{12}$$

and the constraints C on the probabilities variables are

$$C=\{0.85+p_{i2}\leqslant p_{i1}\leqslant 0.95\}.$$

We have $n+1$ actions: $reboot(c_1),\dots,reboot(c_n)$ and $notreboot$, the latter of which indicates that no machine is rebooted. The intuition behind Eq. (12) is that if a computer is rebooted then its probability of running in the next time step is 1 (situation i); if a computer is not rebooted and its current state is running (situation ii) or not running (situation iii), the probability depends on the fraction of computers with incoming connections that are also currently running. The probability parameters p_{i1}, p_{i2} and the constraint C over them define the credal sets $K(\cdot|\vec{x},a)$.

The reward for SysAdmin is simply 1 if all computers are running at any time step otherwise the reward is 0, i.e., $R(\vec{x})=\prod_{i=1}^{n}x_i$. An optimal policy in this problem will reboot the computer that has the most impact on the expected future discounted reward given the network configuration.

Like the previous definition of an enumerated state MDP-IP, the set of all legal transition distributions for a factored MDP-IP is defined as a *credal set* K. The challenge then is to specify such transition credal sets in a factored manner that is

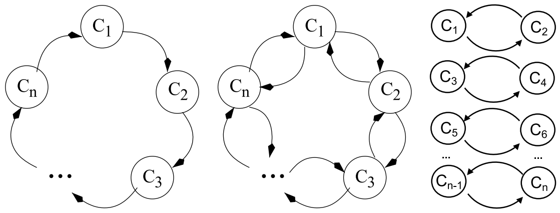
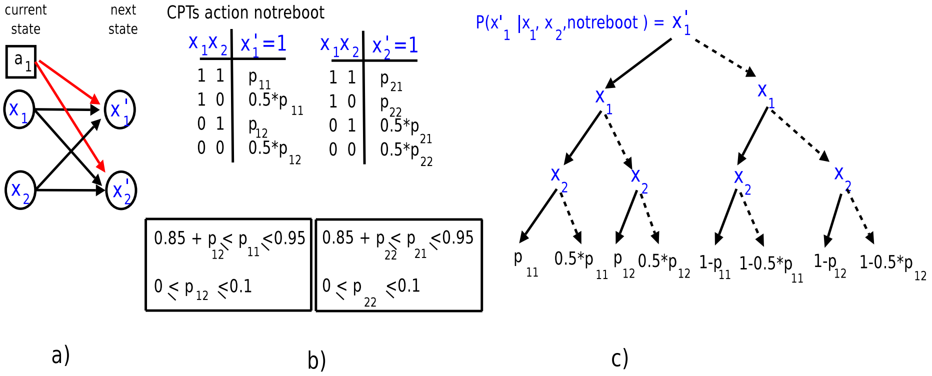


Fig. 3. Connection topologies for the SysAdmin example: a) unidirectional-ring, b) bidirectional ring and c) independent bidirectional rings of pairs of computers [5].



x_1x_2	$x'_1=1$
1 1	p_{11}
1 0	$0.5*p_{11}$
0 1	p_{12}
0 0	$0.5*p_{12}$

x_1x_2	$x'_2=1$
1 1	p_{21}
1 0	p_{22}
0 1	$0.5*p_{21}$
0 0	$0.5*p_{22}$

$0.85 + p_{12} \leq p_{11} \leq 0.95$

$0 \leq p_{12} \leq 0.1$

$0.85 + p_{22} \leq p_{21} \leq 0.95$

$0 \leq p_{22} \leq 0.1$

Fig. 4. a) Dynamic Credal Network for action *notreboot* for an unidirectional-ring topology of SysAdmin domain with 2 computers. b) Conditional probability table for the state variables $x'_1 = 1$ and $x'_2 = 1$ and the constraints related to the probabilities. c) The Parameterized ADD representation for $P(x'_1|x_1, x_2, notreboot)$ that we call $CPT^{x'_1}_{notreboot}$. A solid line indicates the true (1) branch of a variable test and a dashed line indicates the false (0) branch.

itself compact. For this, we propose to use *dynamic credal networks* (*DCNs*), a special case of credal networks [11,15], as an appropriate language to express factored transition credal sets.

Definition 4.1 *(Factored transition credal set).* A credal set containing conditional distributions over the values of a variable x_i, given the values of $pa_a(x_i)$ (the parents of x_i in the graph for action a), is referred to as a *factored transition credal set* and denoted by $K_a(x_i|pa_a(x_i))$.

Definition 4.2 *(Dynamic credal network).* A Dynamic credal network (DCN) is a generalization of a DBN. Different from the definition of a DBN, in a DCN each variable x_i is associated with factored transition credal sets $K_a(x_i|pa_a(x_i))$ for each value of $pa_a(x_i)$. We assume that a DCN represents a *joint credal set* [15,11] over all of its variables consisting of all distributions that satisfy the factorization in Eq. (11), where each CPT distribution $P(x'_i|pa_a(x'_i), a)$ is an element of the transition credal set $K_a(x'_i|pa_a(x'_i))$ associated with the DCN.

A DCN example is shown in Fig. 4(a). For each variable x'_i in a DCN, we have a *conditional probability table* (*CPT*) with imprecise probabilities. If we examine the CPTs in Fig. 4(b), we note that entries are specified by probability parameters p_{ij} (i for variable x'_i and j for the jth parameter in the CPT for x'_i). Furthermore, we note that we have a set of side linear constraints on these p_{ij} (shown in the boxes below the CPT, collectively call this constraint set C). We use $\vec{p}$ to denote a vector containing all parameter values that are free to vary within the given credal sets (i.e., that satisfy the probability constraints C of the DCN).

We note that the joint transition probability may be nonlinear in the probability parameters $\vec{p}$. However, we explicitly introduce the following restriction to prevent exponents exceeding 1:

Restriction 4.3 *(DCN parameter restriction for factored MDP-IP CPTs). a parameter p_{ij} may only appear in the CPT for x'_i.*

This restriction prevents the multiplication of p_{ij} by itself when CPTs for each x'_i are multiplied together to determine the joint transition distribution in the DCN. This subset of nonlinear expressions, where the exponent of each p_{ij} is either 0 or 1, is referred to as a *multilinear* expression. To see the multilinearity of the transition probability in Fig. 4, we observe $P(x'_1 = 1, x'_2 = 1|x_1 = 1, x_2 = 1, notreboot) = p_{11}p_{21}$.

When combined with a set of constraints C on the p_{ij}, there are efficient implementations that we can use in practice to solve the resulting *multilinear program*. Interestingly, because there are no additional restrictions on the linear constraints C defined over the p_{ij} in a multilinear program, Restriction 4.3 actually turns out to be a minor limitation in practice as we demonstrate in the experimental domains of Section 8.

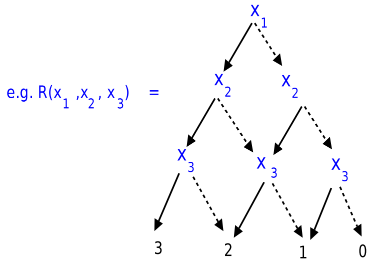


Fig. 5. An example reward function $R(x_1, x_2, x_3) = \sum_{i=1}^{3} x_i$ represented as an ADD.

Even though we can qualitatively represent the conditional independence properties of a distribution using DCNs, there are certain independences that we cannot represent with the Credal network structure, e.g., independences that hold for specific contexts (assignments of values to certain variables) known as *context-specific independence* (CSI) [16]. In order to compactly represent CSI and shared function structure in the CPTs for an MDP-IP, we propose a novel extension of *algebraic decision diagrams* (*ADDs*) [17] called *parameterized ADDs* (*PADDs*) since the leaves are parameterized expressions as shown in Fig. 4(c). PADDs will not only allow us to compactly represent the CPTs for factored MDP-IPs, but they will also enable efficient computations for factored MDP-IP value iteration operations as we outline next.

5. Parameterized algebraic decision diagrams

Algebraic decision diagrams (*ADDs*) [17] are a generalization of ordered *binary decision diagrams* (*BDDs*) that represent boolean functions $\{0,1\}^n \to \{0,1\}$ [18]. A BDD is a data structure that has decision nodes, each node labeled with a boolean *test* variable with two successor nodes: l (low) and h (high). The arc from a node to its successor l (h) represents an assignment 0(1) to the test variable. BDDs are DAGs whose variable tests on any path from root to leaf follow a fixed total variable ordering. BDDs are used to generate the value of a boolean function as follows: given assignments to the boolean test variables in a BDD, we follow branches l or h, until we get to a leaf, which is the boolean value returned by the function. The only difference between an ADD and a BDD is that terminal nodes in an ADD are real values, i.e., ADDs permit the compact representation of functions $\{0,1\}^n \to \mathbb{R}$. BDDs and ADDs often provide an efficient representation of functions with context-specific independence [16] and shared function structure. For example, the reward function $R(x_1, x_2, x_3) = \sum_{i=1}^{3} x_i$ represented in Fig. 5 as an ADD exploits the redundant structure of sub-diagrams through its DAG representation.

Operations on ADDs can be performed efficiently by exploiting their DAG structure and fixed variable ordering. Examples of efficient ADD operations are unary operations such as min, max (return the minimum or maximum value in the leaves of a given ADD), marginalization over variables ($\sum_{x_i}$) that eliminates a variable x_i of an ADD; binary operations such as addition ($\oplus$), subtraction ($\ominus$), multiplication ($\otimes$), division ($\oslash$), and even $\min(\cdot,\cdot)$ and $\max(\cdot,\cdot)$ (return an ADD with min/max values in the leaves). We refer the reader to [17] for details.

Parameterized ADDs (**PADDs**) are an extension of ADDs that allow for a compact representation of functions from $\{0,1\}^n \to \mathbb{E}$, where $\mathbb{E}$ is the space of expressions parameterized by $\vec{p}$ (in our case, we further restrict this to the space of multilinear expressions of $\vec{p}$). For example, the CPT in Fig. 6 represented as a PADD contains leaves consisting of single parameters while Fig. 8(d) shows a PADD with a leaf containing a more complex parameterized expression.

In the following, we formally define PADDs and their basic operations needed to construct efficient solutions for MDP-IPs. Because PADDs are introduced to solve MDP-IPs, we make the following restrictive assumptions: (a) we allow only multilinear expressions in the leaves; (b) we only define a subset of PADD operations that could be inherited from ADDs; and (c) we only show these operations are closed (i.e., yield a resulting PADD with multilinear leaves) for the operations needed in MDP-IPs. Finally, we contribute a new unary operation *MinParameterOut* ($\min_{\vec{p}}$) specific to PADDs.

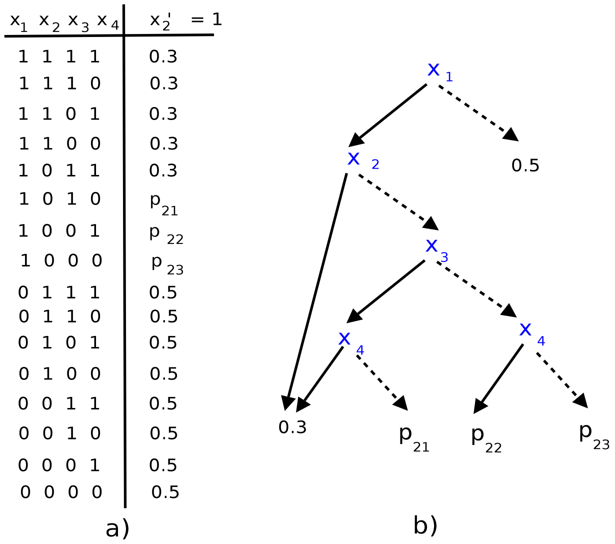
5.1. PADD: Formal definition, properties and operations

PADDs generalize the constant real-valued leaves of ADDs to polynomials (*Poly*) expressed in a sum-of-products canonical form:

$$d_0 + \sum_i d_i \prod_j p_{ij} \tag{13}$$

where the d_i are constants and the p_{ij} are parameters. Formally, we can define a PADD by the following BNF grammar[3]:

[3] We will adopt lowercase (f) to refer to a mathematical function, and uppercase (F) to refer to the function represented structurally as a PADD.



x_1 x_2 x_3 x_4	x_2' = 1
1 1 1 1	0.3
1 1 1 0	0.3
1 1 0 1	0.3
1 1 0 0	0.3
1 0 1 1	0.3
1 0 1 0	p_{21}
1 0 0 1	p_{22}
1 0 0 0	p_{23}
0 1 1 1	0.5
0 1 1 0	0.5
0 1 0 1	0.5
0 1 0 0	0.5
0 0 1 1	0.5
0 0 1 0	0.5
0 0 0 1	0.5
0 0 0 0	0.5

Fig. 6. a) Conditional probability table for the state variable x_2' for action a_1. b) The Parameterized ADD representation for $P(x_2' = 1|x_1, x_2, x_3, x_4, a_1)$.

$$F ::= Poly|\text{if}(F^{var}) \text{ then } F_h \text{ else } F_l$$
$$Poly ::= d_0 + \sum_i d_i \prod_j p_{ij}.$$

This grammar is notationally overloaded, so we briefly explain: a PADD node F can either be a terminal node with an expression of type *Poly* or a decision node with variable test F^{var} (e.g., x_1 or x_n) and two branches F_h and F_l (both of grammar non-terminal type F), where F_h is taken when $F^{var} = 1$ and F_l is taken when $F^{var} = 0$.

The value returned by a function f represented as a PADD F containing (a subset of) the variables $\{x_1, \ldots, x_n\}$ with variable assignment $\rho \in \{0, 1\}^n$ can be defined recursively by:

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

This recursive definition of $Val(F, \rho)$ reflects the structural evaluation of a PADD F by starting at its root node and following the branch at each decision node corresponding to the variable assignment in ρ — this continuing until a leaf node is reached, which is then returned as $Val(F, \rho)$. As an example, for the PADD represented in Fig. 6, assigning $\rho = \{1, 0, 1, 0\}$ for variables $\{x_1, x_2, x_3, x_4\}$ yields $Val(F, \rho) = p_{21}$.

Like ADDs, for any function $f(x_1, \ldots, x_n)$ and a fixed variable ordering over $x_1, \ldots, x_n$, a reduced PADD is defined as the minimally sized ordered decision diagram representation of a function f.

Lemma 5.1. *There exists a unique reduced PADD F (the canonical PADD representation of f) satisfying the given variable ordering such that for all $\rho \in \{0, 1\}^n$ we have $f(\rho) = Val(F, \rho)$.*

The proof of this lemma for BDDs was provided by [19] and can be trivially generalized to ADDs and PADDs. Since PADDs allow polynomial leaves, the only change for demonstrating this lemma is that we need to ensure that there exists a way to identify when two leaf expressions are identical, which can be easily done by (a) sorting the parameters in each multilinear term, (b) factoring out (grouping terms with the same ordered set of parameters) and summing constants in identical multilinear terms, and (c) sorting the list of terms according to the lowest variable index and number of parameters. With such a unique leaf identification method, the proof of [19] generalizes to PADDs and shows that there is a unique canonical PADD representation for every function from $\{0, 1\}^n$ to polynomials in the form of (13).

In fact, not only does such a minimal, reduced PADD always exist for a function f that can be represented as a PADD, but there is a straightforward algorithm for computing it called *ReducePADD*, which we present in Section 5.2.1. Before we present formal PADD algorithms though, we first discuss extensions of the unary and binary operations from ADDs to PADDs. Fortunately, this only requires that operations on the leaves of ADDs are modified to accept and produce resulting polynomials in the form of (13).

5.1.1. Binary operations on PADDs

The binary operations $\oplus$ (sum) and $\ominus$ (subtraction) as defined for ADDs [17] can be extended for PADDs and are always *closed* since these operations yield PADDs with leaves in the form of (13). However, the binary operation $\otimes$ (product) can

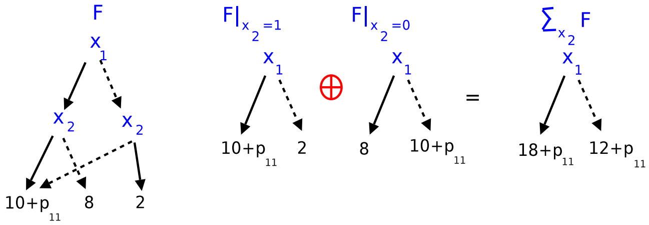


Fig. 7. An example application of *Restrict* operation and *Marginalization* operation on a PADD.

only yield a PADD with leaves in the form of (13) if the set of parameters $\vec{p}$ in the leaves of each operand are disjoint. Fortunately, for factored MDP-IPs, we note that the only place $\otimes$ is used is to compute the product of the DCN CPTs; because of Restriction 4.3 on the usage of parameters p_{ij} in these CPTs, we note that the condition for closed $\otimes$ operations on PADDs is always satisfied for the required factored MDP-IP operations.

However, not all PADD binary operations have simple conditions under which they are closed. We note that PADDs are not closed under $\oslash$ (binary division), i.e., the resulting leaves could be a polynomial fraction and hence cannot be expressed as (13). Similarly, the *binary* $\min(\cdot,\cdot)$ and $\max(\cdot,\cdot)$ operations defined for ADDs [17] cannot generally be computed in closed form unless the actual assignment to the parameters $\vec{p}$ is known. Fortunately, $\oslash, \min(\cdot,\cdot)$, and $\max(\cdot,\cdot)$ will not be needed in our proposed solution to factored MDP-IPs.

5.1.2. Unary operations on PADDs

The two important classical unary operations for ADDs are *restriction* $(F|_{x_i})$ and *marginalization* $(\sum_{x_i})$ and can be easily extended to PADDs as follows:

- *Restriction* of a variable x_i to either *true* $(F|_{x_i=1})$ or *false* $(F|_{x_i=0})$ can be calculated by replacing all decision nodes for variable x_i with either the *high* or *low* branch, respectively. This operation can be used to do marginalization as we show next. This operation does not affect the leaves of the decision diagram, so its extension from ADDs to PADDs is straightforward.
- The *marginalization* or *sum_out* operation (represented as $\sum_{x_i}$) eliminates a variable x_i from an ADD. It is computed as the sum of the *true* and *false* restricted functions, i.e., $(F|_{x_i=1} \oplus F|_{x_i=0})$. Since $\oplus$ is closed for PADDs, marginalization is also closed for PADDs. An example is shown in Fig. 7.

The classical *unary* $\min(\cdot)$ and $\max(\cdot)$ operations for ADDs cannot generally be computed for PADDs unless the actual assignment to the parameters $\vec{p}$ is known. However, we will not need this particular PADD operation for factored MDP-IPs, but rather a new unary operation for PADDs called *MinParameterOut*, which in our case will make the choices of Nature in Eq. (9).

Definition 5.2 *(MinParameterOut operation).* Represented as $\min_{\vec{p}}(F)$, this operation takes as input (1) a PADD F and (2) a set C of global constraints over the PADD's parameters, and returns an ADD. We note that an ADD is a special case of a PADD with constant expressions at its leaves, which implies that $\min_{\vec{p}}(F)$ is closed for PADDs. This unary operation calls a nonlinear solver for each leaf expression e in the form of (13) to compute $c = \min_{\vec{p}}(e)$ w.r.t. constraints C and replaces the leaf e with the constant c.

Because the set of variable assignments that can reach each PADD leaf are disjoint, each leaf can be minimized independently of the others. This is precisely the operation we'll need for factored MDP-IPs, since we note that Nature performs it's minimization independently per state s in (9), and every path in the PADD will correspond to a different state assignment. An example of $\min_{\vec{p}}(F)$ is shown in Fig. 12.

5.2. PADD algorithms

Previously we have discussed PADD algorithms conceptually, in this subsection, we discuss how to implement efficient operations for PADDs. In the following algorithms we use four hash tables: *ReduceCache*, *NodeCache*, *ApplyCache* and *MinParCache*. We use the notation *key* → *value* to represent key/value pairs in the hash table. The table *NodeCache* stores a unique identification for each node (representing sub-diagrams by unique identifiers), the hash table *ReduceCache* stores the reduced canonical nodes (the results of previous *Reduce* operations), the table *ApplyCache* stores the results of previous *Apply* operations (so we can avoid redundant calculations) and the hash table *MinParCache* stores the results of previous *MinParameterOut* operations.

Algorithm 1: ReducePADD(F)

```
input  : $F$ (root node id for an arbitrary ordered decision diagram)
output: $F_r$ (root node id for reduced PADD)
1  begin
2      //if terminal node, return canonical terminal node
3      if $F$ is terminal node then
4          return canonical terminal node for polynomial of $F$;
5      //use recursion to reduce sub diagrams
6      if $F \to F_r$ is not in ReduceCache then
7          $F_h$ = ReducePADD ( $F_h$ ) ;
8          $F_l$ = ReducePADD ( $F_l$ ) ;
9          //get a canonical internal node id
10         $F_r$ = GetNode ( $F^{var}$, $F_h$, $F_l$ ) ;
11         insert $F \to F_r$ in ReduceCache;
12     return $F_r$;
13 end
```

Algorithm 2: GetNode($\langle var, F_h, F_l \rangle$)

```
input  : $\langle var, F_h, F_l \rangle$ (variable and true and false branches node ids for internal node)
output: $F_r$ (canonical internal node id)
begin
    //redundant branches
    if $F_l = F_h$ then
        return $F_l$;
    //check if the node exists previously
    if $\langle var, F_h, F_l \rangle \to id$ is not in NodeCache then
        id = new unallocated id;
        insert $\langle var, F_h, F_l \rangle \to id$ in NodeCache;
    return $id$;
end
```

5.2.1. Reduce algorithm for PADDs

While we know there exists a unique canonical form for every function expressible as a PADD (Lemma 5.1), the algorithm *ReducePADD* actually allows the efficient construction of this unique canonical PADD representation from an arbitrary ordered decision diagram with polynomial leaves of type (13).

Algorithm 1 recursively constructs such a reduced PADD from the bottom up. In this algorithm, an internal node is represented as $\langle F^{var}, F_h, F_l \rangle$, where F^{var} is the variable name, and F_h and F_l are the true and false branch node ids, respectively. Additionally, the input F refers to an arbitrary node, while the returned value F_r refers to a canonical node id. Reduced canonical nodes are stored in the hash table *ReduceCache* and the helper function *GetNode* (Algorithm 2) ensures that redundant decision tests at internal nodes are removed. The table *NodeCache* used in the function *GetNode* stores a unique identification for each node.

An example of the application of the *ReducePADD* algorithm is shown in Fig. 8. The hollow arrow points to the internal node F that is being evaluated by *ReducePADD* after the two recursive calls to *ReducePADD* (lines 7 and 8) but before line 10. Fig. 8(a) shows the input diagram for the algorithm where x_3 is being evaluated by *ReducePADD* creating two canonical terminal nodes for $0.3 + 5p_{12}$ and 0. Note that while evaluating node x_3 (on the left), the execution of line 10 will result in the insertion of $\langle x_3, 0.3 + 5p_{12}, 0 \rangle$ in the *NodeCache* hash table. Fig. 8(b) shows the resulting evaluation of node x_3 on the right, which returns the same previous canonical terminal nodes for $0.3 + 5p_{12}$ and 0. And again, after executing line 10, the *GetNode* algorithm will return the same id for $\langle x_3, 0.3 + 5p_{12}, 0 \rangle$, previously inserted in the *NodeCache*. Fig. 8(c) shows the evaluation of x_2. Note that F_h and F_l are equal, thus after *getNode* is called, F_l is returned and as a consequence x_2 disappears. Finally, Fig. 8(d) shows the canonical PADD representation of the input. Note that *ReducePADD* (F_l) returned the same canonical terminal node that exists previously for the node 0.

The running time and space of *ReducePADD* are linear in the size of the input diagram since the use of the *ReduceCache* guarantees that each node is visited only once and at most one unique reduced node is generated in the canonical diagram for every node visited.

5.2.2. Apply algorithm for binary operations for PADDs

The notation we will use in this paper for PADDs is shown in Fig. 9. Any operation with two PADDs, F_1 and F_2, results in a new canonical PADD F_r, with eventually a new root node F_r^{var} and two new sub-diagrams F_h and F_l. Note that $F_{i,h}$ and $F_{i,l}$ represent sub-diagrams.

For *all* binary operations, we use the generic function *Apply*(F_1, F_2, op) (Algorithm 3) and the result computation table in the helper function *ComputeResult* (Table 1) that supports operations between arbitrary PADD nodes and polynomial leaves.

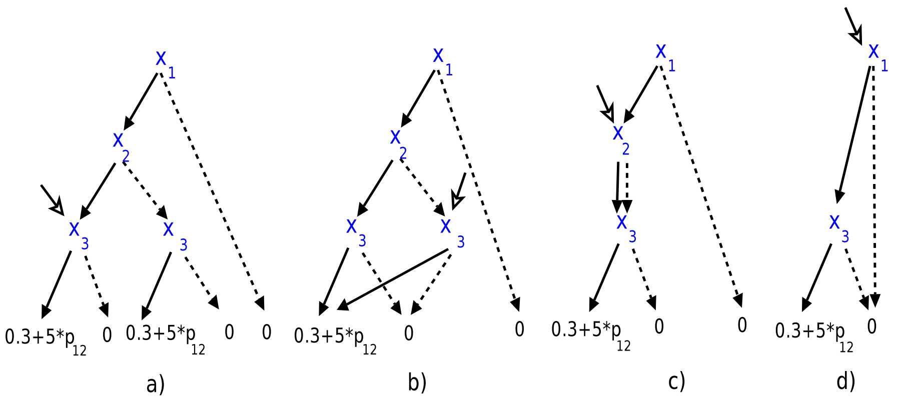


Fig. 8. A step-by-step illustration of the application of *ReducePADD* algorithm (Algorithm 1) where a) and d) are the input and output PADDs, respectively.

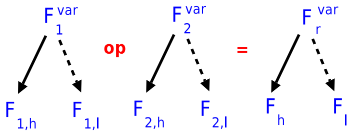


Fig. 9. The notation used in the *Apply* and *ChooseVarBranch* algorithms.

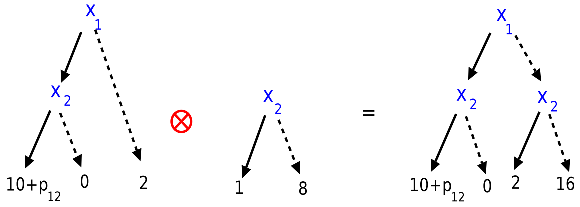


Fig. 10. An example of PADDs multiplication.

Table 1 is implemented as a method named *ComputeResult*, which is simply a case structure for each line of Table 1. Notice that lines 2–9 of Table 1 define the result of PADD operations in special cases that avoid unnecessary computation in *Apply*.

The *Apply* algorithm (Algorithm 3) has as input two operands represented as canonical PADDs, F_1 and F_2, and a binary operator $op \in \{\oplus, \ominus, \otimes\}$; the output is the result of the function application represented as a canonical PADD F_r. *Apply* (F_1, F_2, op) first checks if the result can be immediately computed by calling the method *ComputeResult* (line 3). If the result is *null*, it then checks whether the result was previously computed by checking in the *ApplyCache*, which stores the results of previous *Apply* operations (line 6). If there is not a cache hit, *Apply* chooses the earliest variable in the ordering to branch on by calling the auxiliary function *ChooseVarBranch* (Algorithm 4) and then branches on this variable with two recursive *Apply* calls, one to compute F_l and other to compute F_h. After that, the results of these two operations are checked for redundancy elimination throughout *GetNode* function. An example of PADD multiplication via *Apply* algorithm is shown in Fig. 10.

5.2.3. *MinParameterOut algorithm for PADDs*

The *MinParameterOut* algorithm (Algorithm 5) has as input a canonical PADD F and a set of constraints C over the PADD's parameters; the output is the result of calling the nonlinear solver for each PADD leaf, represented as a canonical ADD F_r. *MinParameterOut* first checks if F is a constant terminal node, in this case it is not necessary to call the nonlinear solver for this leaf. If the terminal node is not a constant then we need to make a call to the nonlinear solver passing the leaf expression as an objective to minimize subject to C (line 7). If F is not a terminal node, Algorithm 5 recursively traverses the PADD. Similar to *ReducePADD*, an internal node is represented as $\langle F^{var}, F_h, F_l\rangle$ and previously computed canonical nodes are stored in the hash table *MinParCache*. The helper function *GetNode* (Algorithm 2) ensures again that redundant decision tests at internal nodes are removed.

With this last specification of *MinParameterOut*, we have formally described almost all of the PADD algorithms we will need in our factored MDP-IP solution. We omit the restriction and marginalization algorithms for PADDs since they are identical to the same operations for ADDs (i.e., these operations don't modify the leaves, which is the only place that PADDs and ADDs differ).

Algorithm 3: APPLY(F_1, F_2, op)

```
input : $F_1$ (root node id for operand 1),
        $F_2$ (root node id for operand 2),
        $op$ (binary operator, $op \in \{\oplus, \ominus, \otimes\}$)
output: $F_r$ (root node id for the resulting reduced PADD)
begin
    //check if the result can be immediately computed
    if COMPUTERESULT($F_1, F_2, op$) $\rightarrow F_r \neq null$ then
        return $F_r$;
    //check if we previously computed the same operation
    if $\langle F_1, F_2, op\rangle \rightarrow F_r$ is not in ApplyCache then
        //choose variable to branch
        var = CHOOSEVARBRANCH($F_1, F_2$);
        //set up nodes for recursion
        if $F_1$ is non-terminal $\wedge$ $var = F_1^{var}$ then
            $F_l^{v1} = F_{1,l}$;
            $F_h^{v1} = F_{1,h}$;
        else
            $F_{l,h}^{v1} = F_1$;
        if $F_2$ is non-terminal $\wedge var = F_2^{var}$ then
            $F_l^{v2} = F_{2,l}$;
            $F_h^{v2} = F_{2,h}$;
        else
            $F_{l,h}^{v2} = F_2$;
        //use recursion to compute true and false branches for resulting PADD
        $F_l$ = Apply($F_l^{v1}, F_l^{v2}, op$);
        $F_h$ = Apply($F_h^{v1}, F_h^{v2}, op$);
        $F_r$ = GETNODE($var, F_h, F_l$);
        //save the result to reuse in the future
        insert $\langle F_1, F_2, op\rangle \rightarrow F_r$ into ApplyCache;
    return $F_r$;
end
```

Algorithm 4: CHOOSEVARBRANCH(F_1, F_2)

```
input : $F_1$ (root node id for operand 1),
        $F_2$ (root node id for operand 2)
output: $var$ (selected variable to branch)
begin
    //select the variable to branch based on the order criterion
    if $F_1$ is a non-terminal node then
        if $F_2$ is a non-terminal node then
            if $F_1^{var}$ comes before $F_2^{var}$ then
                $var = F_1^{var}$;
            else
                $var = F_2^{var}$;

        else
            $var = F_1^{var}$;

    else
        $var = F_2^{var}$;
    return $var$;
end
```

6. Factored MDP-IP value iteration

In the two previous sections, we showed a compact representation for factored MDP-IPs based on dynamic credal networks (DCNs) and parameterized ADDs (PADDs) and the respective algorithms needed to manipulate DCNs and PADDs. In this section, we will present our first exact value iteration solution that exploits both of these representations. This solution is an extension of the *SPUDD* [3] algorithm. First, we give a mathematical description of the proposed solution and then proceed to formally specify the algorithm that computes it.

6.1. SPUDD-IP description

We extend the *SPUDD* [3] algorithm for exploiting DBN and ADD structure in the solution of factored MDPs to a novel algorithm *SPUDD-IP* for exploiting DCN and PADD structure in the solution of factored MDP-IPs. We begin by expressing

Table 1
Input case and result for the method *ComputeResult* for binary operations ⊕, ⊖ and ⊗ for PADDs.

Case number	Case operation	Return
1	F_1 *op* F_2; $F_1 = Poly_1$; $F_2 = Poly_2$	$Poly_1$ *op* $Poly_2$
2	$F_1 \oplus F_2$; $F_2 = 0$	F_1
3	$F_1 \oplus F_2$; $F_1 = 0$	F_2
4	$F_1 \ominus F_2$; $F_2 = 0$	F_1
5	$F_1 \otimes F_2$; $F_2 = 1$	F_1
6	$F_1 \otimes F_2$; $F_1 = 1$	F_2
7	$F_1 \otimes F_2$; $F_2 = 0$	0
8	$F_1 \otimes F_2$; $F_1 = 0$	0
9	other	*null*

Algorithm 5: MinParameterOut(F, C)

```
input : $F$ (root node id for a PADD),
        $C$ (set of constraints)
output: $F_r$ (root node id for an ADD)
begin
    //if terminal node, call the solver and return the value
    if $F$ is terminal node then
        node = canonical terminal node for polynomial of $F$;
        if node is a constant then
            return node;
        $c$ = CallNonLinearSolver (node,$C$) ;
        return canonical terminal node for the constant $c$;
    //use recursion to compute sub diagrams
    if $F \rightarrow F_r$ is not in ReduceCacheMinPar then
        $F_h$ = MinParameterOut ( $F_h$ ) ;
        $F_l$ = MinParameterOut ( $F_l$ ) ;
        //get a canonical internal node id
        $F_r$ = GetNode ( $F^{var}$, $F_h$, $F_l$ ) ;
        insert $F \rightarrow F_r$ in ReduceCacheMinPar;
    return $F_r$;
end
```

MDP-IP value iteration from (10) in the following factored form using the transition representation of (11) and operations on decision diagrams[4]:

$$V^t_{DD}(\vec{x}) = \max_{a\in A}\left\{ R_{DD}(\vec{x},a) \oplus \gamma \min_{\vec{p}} \boxed{\sum_{\vec{x}'} \bigotimes_{i=1}^{n} P_{DD}\big(x'_i|pa_a(x'_i),a\big) V^{t-1}_{DD}(\vec{x}')} \right\}. \tag{14}$$

Because the transition CPTs in the MDP-IP DCN contain parameters $\vec{p}$, these CPTs must be represented in decision diagram format as PADDs ($P_{DD}(x'_i|pa_a(x'_i),a)$). On the other hand, the reward $R_{DD}(\vec{x},a)$ can be represented as an ADD since it contains only constants (for the purpose of operations, recall that ADDs are special cases of PADDs). Although it may appear that the form of $V^t_{DD}(\vec{x})$ is a PADD, we note that the parameters $\vec{p}$ are "minimized"-out w.r.t. the side constraints on $\vec{p}$ during the $\min_{\vec{p}} \Box$ operation in (14) (remember that $\min_{\vec{p}} \Box$ is the *MinParameterOut* operation on PADDs, that performs the minimization over the parameters by calling a nonlinear solver for each leaf and returns an ADD). This is crucial, because the $\max_{a\in A}$ can only be performed on ADDs (recall that max is not a closed operation on PADDs). Thus the resulting $V^t_{DD}(\vec{x})$ computed from the $\max_{a\in A}$ has constant leaves and can be expressed as the ADD special case of PADDs.

To explain the efficient evaluation of (14) in more detail, we can exploit the variable elimination algorithm [20] in the marginalization over all next states $\sum_{\vec{x}'}$. For example, if x'_1 is not dependent on any other x'_i for $i \neq 1$, we can "push" the sum over x'_1 inwards to obtain:

$$V^t_{DD}(\vec{x}) = \max_{a\in A}\left\{ R_{DD}(\vec{x},a) \oplus \gamma \min_{\vec{p}} \boxed{\sum_{x'_i(i\neq 1)} \bigotimes_{i=1(i\neq 1)}^{n} P_{DD}\big(x'_i|pa_a(x'_i),a\big) \sum_{x'_1} P_{DD}\big(x'_1|pa_a(x'_1),a\big) V^{t-1}_{DD}(\vec{x}')} \right\}. \tag{15}$$

We show this graphically in the example of Fig. 11. Here, we have the ADD representation for the first value function $V^0_{DD} = R_{DD}$ (Fig. 11(a)), which we multiply by $P_{DD}(x'_1|pa_a(x'_1),a)$ (Fig. 11(b)) yielding the result (Fig. 11(c)) and sum this out over

[4] We use DD for the functions represented by ADDs or PADDs, since the first is a special case of the second.

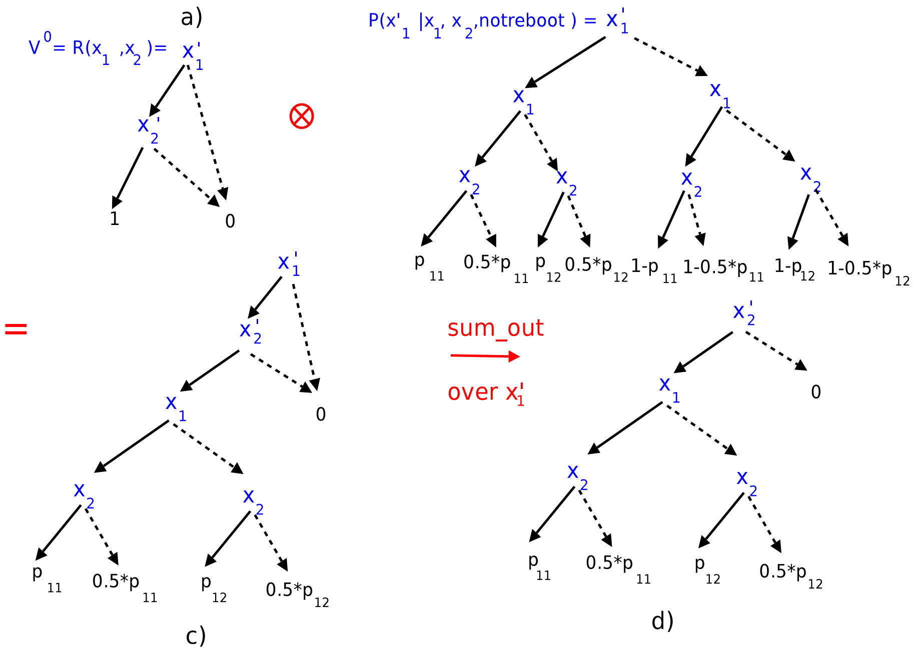


Fig. 11. a) We show $V^0_{ADD} = R(x_1, x_2)$ for the unidirectional-ring topology of SysAdmin domain with 2 computers represented as an ADD. b) The PADD representation for $P(x_1'|x_1, x_2, notreboot)$ ($CPT^{x_1'}_{notreboot}$). c) The multiplication $V^0_{ADD} \otimes CPT^{x_1'}_{notreboot}$ resulting in a PADD. d) The result of summing out over x_1', which is a PADD.

x_1' to obtain the final result (Fig. 11(d)). Then we can continue with x_2', multiplying this result by the $P_{DD}(x_2'|pa_a(x_2'), a)$, summing out over x_2', and repeating for all x_i' to compute □. After this □ does not contain anymore the variables x_i', but only the variables x_i.

Representing the contents of □ as $f(\vec{x}, a, \vec{p})$, we obtain

$$V^t_{DD}(\vec{x}) = \max_{a \in A} \left\{ R_{DD}(\vec{x}, a) \oplus \gamma \min_{\vec{p}} \boxed{f(\vec{x}, a, \vec{p})} \right\}. \qquad (16)$$

Note that $\min_{\vec{p}} f(\vec{x}, a, \vec{p})$ leads to a separate nonlinear expression minimization for every $\vec{x}$ and every a subject to the set C of side linear constraints on $\vec{p}$ (given with the DCN specification) since this follows from the definition of the MDP-IP Bellman equation — every state gets its own minimizer and each PADD leaf corresponds to a set of states with exactly the same minimization objective. This optimization problem may be represented as a simple *multilinear program* due to Restriction 4.3 that guarantees each p_{ij} only appears in the DCN CPT for x_i' (this prevents multiplication of p_{ij} by itself, thereby preventing exponents exceeding 1). This restriction is important to guarantee the existence of exact solutions and the existence of efficient implementations that we can use in practice to solve multilinear programs. We note that this is only a restriction on the factored MDP-IP models themselves.

To demonstrate how the $\min_{\vec{p}} f(\vec{x}, a, \vec{p})$ is performed on PADDs, we refer to Fig. 12. Here, each leaf expression in $f(\vec{x}, a, \vec{p})$ (Fig. 12(a)) given by the PADD corresponds to the function that Nature must minimize in each state. We crucially note that the PADD aggregates states with the same minimization objective, thus saving time-consuming calls to the multilinear solver. We will observe this time savings in our experiments. Now, we need to make a call to the multilinear solver for each leaf, passing the leaf expression as the objective to minimize subject to the side constraints C of our DCN that specify the legal $\vec{p}$ — after the minimization, we can replace this leaf with a constant for the optimal objective value returned by the multilinear solver (Fig. 12(b)). We can see that after the $\min_{\vec{p}}$ operation, all PADDs are simplified to the special case of ADDs with leaf nodes that are constants.

To complete one step of factored MDP-IP value iteration, we take the ADD resulting from the $\min_{\vec{p}}$ operation, multiply it by the scalar γ, add in the reward $R_{DD}(\vec{x}, a)$, and finally perform a sequence of binary *ADD* $\max(\cdot,\cdot)$ operations to compute the $\max_a$, thus yielding the ADD $V^t_{DD}(\vec{x})$ from the ADD for $V^{t-1}_{DD}(\vec{x})$ and completing one step of value iteration from (14).

6.2. SPUDD-IP algorithm

Factored MDP-IP value iteration is formally specified in the following two procedures:

`Solve` (Algorithm 6) constructs a series of t-stage-to-go value functions V^t_{DD} that are represented as ADDs. First we create the PADD representation of all DCN CPTs in the MDP-IP and initialize the first value function to 0 (line 3). The loop is repeated until a maximum number of iterations or until a Bellman error BE termination condition ($BE < tol$) is met.

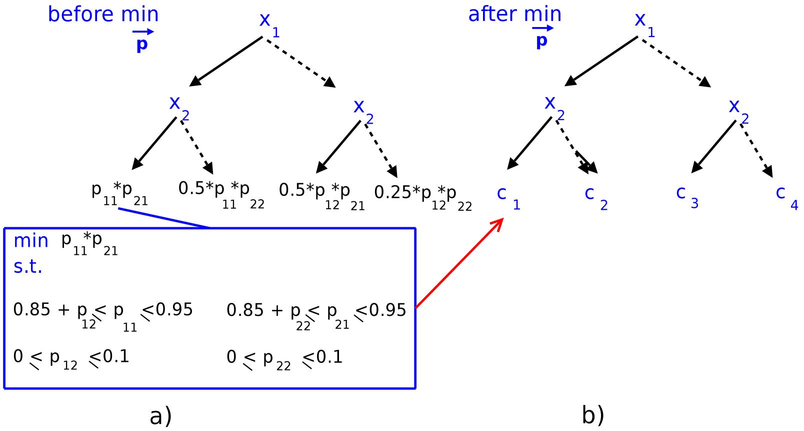


Fig. 12. The *MinParameterOut* operation example. a) The PADD before minimization and a multilinear program for the first leaf, the solution for this leaf is the constant value c_1. b) The resulting ADD after the minimization at all leaves.

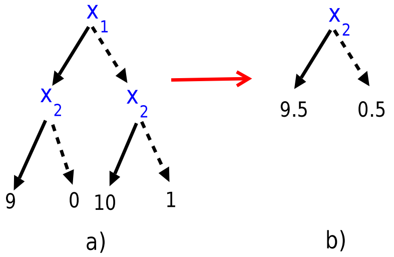


Fig. 13. a) The value function V^t_{DD} represented as an ADD. b) the result of ApproxADD applied to V^t_{DD} with approximation *error* $= 1$; note that the leaves within *error* of each other have been merged and averaged and the resulting ADD simplified.

We note that setting the tolerance *tol* according to (7) guarantees ϵ-optimality for MDP-IPs since the same termination conditions used for MDPs directly generalize to MDP-IPs in the discounted case ($\gamma < 1$). At each iteration the Regress algorithm is called (line 13) and V^t_{DD} is updated with the max over all Q^t_{DD} (there is a Q^t_{DD} for each action a). After this, $BE = \max_{\vec{x}} |V^t(\vec{x}) - V^{t-1}(\vec{x})|$ is computed and tested for termination. We observe in Algorithm 6 the parameters δ, *APRICODD*, *Objective*, and *Vmax* play no role now; they are used for approximation as we explain in the next section (in particular we use $\delta = 0$ to obtain an exact solution by the SPUDD-IP).

Regress (Algorithm 7) computes Q^t_{DD}, i.e, it regresses V^{t-1}_{DD} through action a that provides the values Q^t_{DD} that could be obtained if executing a and acting so as to obtain V^{t-1}_{DD} thereafter. During regression we "prime" the variables using the function convertToPrimes that converts each x_i to x'_i (since the V^i_{DD} is now part of the "next" state) and the CPTs for action a are multiplied in and summed out (lines 4–6). We assume here there are no synchronic arcs among variables x'_i, x'_j for $i \neq j$ in the DCN. If synchronic arcs are present, the algorithm can be simply modified to multiply in all relevant CPTs. After this, the *MinParameterOut* function is performed that calls the multilinear solver to find the minimizing $\vec{p}$ for each leaf in the PADD w.r.t. the side linear constraints C on the DCN (line 11), resulting in an ADD. We note that if a leaf is already a constant, then the multilinear solver call can be avoided altogether; this observation will prove important later when we introduce objective pruning. Finally, the future value is discounted and the reward ADD is added in to complete the regression. *Objective* and *error* are used for approximate value iteration and will be discussed later.

7. Factored MDP-IP approximate value iteration

The previous SPUDD-IP exact value iteration solution to factored MDP-IPs often yields an improvement over flat value iteration as we will demonstrate in our experiments. But as the number of state variables in a problem grows larger, it often becomes impossible to obtain an exact solution due to time and space limitations.

Approximate value iteration (*AVI*) is one way to trade off time and space with error by approximating the value function after each iteration. In this section, we propose two (bounded) AVI extensions of SPUDD-IP: the APRICODD-IP and the Objective-IP algorithms. Each method uses a different way to approximate the value, but both methods incur a maximum of $\delta \cdot Vmax$ error per iteration where *Vmax* as computed in Solve represents the maximum possible value at each step of value iteration (with $0 < \delta \leqslant 1$). By making the approximation error sensitive to *Vmax* we prevent over-aggressive value approximation in the initial stages of AVI when values are relatively small as suggested in [4]. Even with this value ap-

Algorithm 6: Solve (MDP-IP, *tol*, *maxIter*, δ, *APRICODD*, *Objective*)

```
input  : MDP-IP (given by $\langle S, A, R, K, \gamma \rangle$),
         tol (tolerance that guarantees $\epsilon$-optimality),
         maxIter (maximum number of iterations),
         //variables used for approximate value iteration
         $\delta$ (fraction of the maximum possible value, with $0 < \delta \leqslant 1$),
         APRICODD ($APRICODD = true$ to execute APRICODD-IP),
         Objective ($Objective = true$ to execute Objective-IP)
output: $V^t_{DD}$ (t-state-to-go value function)
1  begin
2      Create PADD: $P_{DD}(x'_i|pa(x'_i), a)$ for MDP-IP;
3      $V^0_{ADD} = 0$;
4      //Vmax is the maximum possible value at each iteration
5      $Vmax = \max(R_{DD})$;
6      $t = 0$;
7      //construct t-stage-to-go value functions $V^t_{DD}$ until termination condition is met
8      while $i < maxIter$ do
9          $t = t + 1$;
10         $V^t_{DD} = -\infty$;
11         //update $V^t_{DD}$ with the max over all $Q^t_{DD}$
12         foreach $a \in A$ do
13             $Q^t_{DD}$=Regress ( $V^{t-1}_{DD}$, $a$, $\delta \cdot Vmax$, Objective ) ;
14             $V^t_{DD}$=max ( $V^t_{DD}$,$Q^t_{DD}$ ) ;
15         //compute Bellman Error (BE) and check for termination
16         $Diff_{DD} = V^t_{DD} \ominus V^{t-1}_{DD}$;
17         $BE = \max(\max(Diff_{DD}), -\min(Diff_{DD}))$;
18         if $BE < tol$ then
19             break;
20         //approximate value iteration: APRICODD-IP
21         if APRICODD pruning then
22             $V^t_{DD}$ =ApproxADD ($V^t_{DD}$, $\delta \cdot Vmax$);
23         $Vmax = \max(R_{DD}) + \gamma Vmax$;
24     return $V^t_{DD}$ ;
25 end
```

Algorithm 7: Regress(V_{DD}, a, *error*, *Objective*)

```
input  : $V_{DD}$ (value function),
         $a$ (action),
         error (maximum error),
         Objective ($Objective = true$ to execute Objective-IP)
output: $Q_{DD}$ (the value function obtained if executing $a$ and acting so as obtain $V_{DD}$ thereafter)
1  begin
2      $Q_{DD}$ = convertToPrimes ( $V_{DD}$ ) ; //convert variables $x_i$ to $x'_i$
3      //CPTs are multiplied in and summed out
4      for all $x'_i$ in $Q_{DD}$ do
5          $Q_{DD} = Q_{DD} \otimes P_{DD}(x'_i|pa(x'_i), a)$;
6          $Q_{DD} = \sum_{x'_i} Q_{DD}$;
7      //approximate value iteration: Objective-IP
8      if Objective pruning then
9          $Q_{DD}$ =approxPADDLeaves ($Q_{DD}$, error);
10     //call the nonlinear solver for each PADD leaf — returns an ADD
11     $Q_{DD}$ = MinParameterOut ( $Q_{DD}$,C);
12     $Q_{DD} = R_{DD} \oplus (\gamma \otimes Q_{DD})$ ;
13     return $Q_{DD}$;
14 end
```

proximation at every iteration, satisfying the termination condition $BE < tol$ for some *tol* still yields strict guarantees on the overall approximation error given by (7) as discussed previously for SPUDD-IP.

7.1. APRICODD-IP algorithm

The APRICODD algorithm [4] provides an efficient way of *approximating the ADD value representation* for a factored MDP, reducing its size and thus reducing computation time per iteration. This approach immediately generalizes to MDP-IPs since the value function V^t_{DD} is also an ADD. To execute *APRICODD-IP* AVI for MDP-IPs, we simply call `Solve` (Algorithm 6) with $APRICODD = true$ and set δ $(0 < \delta \leqslant 1)$ to some fraction of the maximum possible value *Vmax* with which to approximate calling the algorithm `ApproxADD` (line 22 Algorithm 6).

Algorithm 8: ApproxADD($value^i_{DD}$,$error$)

```
input  : $value^i_{DD}$ (an ADD),
         $error$ (maximum error)
output: a new ADD
begin
    //collect all leaves of the ADD
    $leaves_{old}$=collectLeavesADD ($value^i_{DD}$);
    //group the leaves that can be merged within maximum error
    $\{leaves_{old} \rightarrow leaves_{new}\}$ = mergeLeaves ($leaves_{old}$, $error$);
    //return a simplified ADD
    return createNewDD ($value^i_{DD}$, $\{leaves_{old} \rightarrow leaves_{new}\}$);
end
```

ApproxADD (Algorithm 8) has two inputs: (1) a value function represented as an ADD and (2) an approximation error to merge the leaves. The output is a new ADD with the merged leaves. The algorithm first collects all leaves of the ADD and determines which can be merged to form new values without approximating more than *error*. The old values are then replaced with these new values creating a new (minimally reduced) ADD that represents the approximated value function. An illustrative example is shown in Fig. 13.

collectLeavesADD compiles the leaves of the ADD and puts them in a set ($leaves_{old}$). In the example in Fig. 13, the set of old leaves is $\{9, 0, 10, 1\}$.

mergeLeaves groups together the leaves that can be merged within *error* and computes the average for each group, creating a new set of leaves ($leaves_{new}$). In the example of Fig. 13, the groups than can be merged within an $error = 1$ are $\{9, 10\}$ and $\{0, 1\}$; and the new leaves are 9.5 and 0.5.

createNewDD creates a simplified ADD, replacing the old leaves by the new ones. The result of this operation in the example is shown in Fig. 13(b)).

7.2. Objective-IP algorithm

APRICODD is an effective extension of SPUDD for factored MDPs (not MDP-IPs) because it reduces the size of the value function ADDs, which largely dictate the time complexity of the SPUDD algorithm. However, in solving (factored) MDP-IPs, the time is dictated less by the size of the value function ADD and more by the number of calls to the multilinear optimizer to compute the $\min_{\vec{p}} \square$. SPUDD-IP started to attack this source of time complexity by aggregating states with the same objective for the $\min_{\vec{p}} \square$. Our goal with the *Objective-IP* pruning algorithm will be to more closely target the source of time complexity in an AVI version of SPUDD-IP by *approximating the objectives* in an attempt to avoid calling the solver altogether. To execute *Objective-IP* for MDP-IPs, we simply call Solve (Algorithm 6) with $APRICODD = false$, $Objective = true$ and set δ $(0 < \delta \leqslant 1)$ to some fraction of the maximum possible value *Vmax*. Noting that each PADD leaf in Regress function is a multilinear objective, we simplify it by calling ApproxPADDLeaves (line 9 Algorithm 7) just prior to carrying out the multilinear optimization at the leaves of that PADD (line 11 Algorithm 7).

ApproxPADDLeaves (Algorithm 9) is called for a PADD by Regress when $Objective = true$. It takes as input a PADD and the maximum error, and the output is a new PADD with approximated leaves using the upper and lower bounds of the parameters. The main loop of the algorithm attempts to approximate each leaf in a PADD (lines 3–17). To approximate the multilinear term i, Algorithm 9 first computes the average of their maximum and minimum values (line 8), this requires knowing the absolute upper p^U_{ij} and lower bounds p^L_{ij} for any p_{ij}, which can be easily precomputed once for the entire MDP-IP by calling the nonlinear solver to compute $p^U_{ij} = \max p_{ij}$ and $p^L_{ij} = \min p_{ij}$ subject to the side linear constraints C on all CPTs. After that, Algorithm 9 computes the error incurred by using these maximum and minimum values (line 10). If the actual accumulated error for the leaf ($curError + termError_i$) is less than the maximum error (*error*), the term i is removed (line 13) and replaced by the average (line 14). In some cases the complexity of the leaf expression may be reduced, in others, it may actually be reduced to a constant. Note that the leaves are each approximated independently, this can be done since each leaf corresponds to a different state (or set of states) and the system can only be in one state at a time. Furthermore, we can guarantee that no objective pruning at the leaves of the PADD incurs more than *error* after the multilinear optimization is performed:

Theorem *(ApproxPADDLeaves Error Bound). Given an MDP-IP, its precomputed constants p^L_{ij} and p^U_{ij} for all p_{ij}, and the maximum approximation error, then whenever ApproxPADDLeaves (Algorithm 9) reduces a leaf $d_0 + \sum_{i=1}^{\#terms} d_i \prod_j p_{ij}$ to a simpler expression, the approximation error in the objective minimization $(\min_{\vec{p}})$ of that leaf is bounded by error.*

Proof. We begin by showing that the approximation error induced by removing a single term i from the objective is bounded by $termError_i$. To do this, we first find upper and lower bounds on term i $(d_i \prod_j p_{ij})$ based on the legal values of $\vec{p}$. We know the maximal (minimal) possible value for each p_{ij} is p^U_{ij} (p^L_{ij}). Thus for any possible legal values of $\vec{p}$ the term i must be bounded in the interval $[L_i, U_i]$ with L_i and U_i defined as follows:

Algorithm 9: ApproxPADDLeaves(*DD*, *error*)

```
input  : DD (parameterized ADD),
         error (maximum error)
output: DD (simplified parameterized ADD)
begin
    //approximate each leaf independently
    foreach $leaf: d_0 + \sum_{i=1}^{\#terms} d_i \prod_j p_{ij} \in DD$ do
        $i = 1$, $curError = 0$;
        //for all terms of the leaf, prune them if possible
        while $curError < error \wedge i \leqslant \#terms$ do
            //compute the average of max and min values for term $i$
            $newValue = \frac{d_i}{2}(\prod_j p_{ij}^U + \prod_j p_{ij}^L)$ ;
            //compute the error of using max and min values for term $i$
            $termError_i = |\frac{d_i}{2}(\prod_j p_{ij}^U - \prod_j p_{ij}^L)|$ ;
            //if within error, prune term $i$ from leaf
            if $curError + termError_i < error$ then
                remove term $d_i \prod_j p_{ij}$ from $leaf$;
                $d_0 = d_0 + newValue$;
                $curError = curError + termError_i$;
            $i = i + 1$;

    return $DD$;
end
```

$$L_i = \begin{cases} d_i > 0 & d_i \prod_j p_{ij}^L \\ d_i < 0 & d_i \prod_j p_{ij}^U \end{cases}, \qquad U_i = \begin{cases} d_i > 0 & d_i \prod_j p_{ij}^U \\ d_i < 0 & d_i \prod_j p_{ij}^L. \end{cases}$$

Let g be a value for term i and $\widehat{g} = \frac{L_i + U_i}{2}$, then $max_g|g - \widehat{g}|$ occurs at $g = L_i$ or $g = U_i$. So the max $termError_i = max(|L_i - \widehat{g}|, |U_i - \widehat{g}|) = \max(|\frac{L_i - U_i}{2}|, |\frac{U_i - L_i}{2}|) = |\frac{L_i - U_i}{2}|$ as computed in Algorithm 9.

Now, let $OBJ1 = d_0 + \sum_{i=1}^{\#terms} d_i \prod_j p_{ij}$ be the original *non-approximated objective expression* to minimize and v_1 the optimal objective value using $\vec{p} = \vec{p_1}$. Let $OBJ2 = d_0 + \widehat{g} + \sum_{i=2}^{\#terms} d_i \prod_j p_{ij}$ be the *approximated objective expression* to minimize after replacing term 1 with $\frac{L_1 + U_1}{2}$ and v_2 the optimal objective using $\vec{p} = \vec{p_2}$.

We want to prove that $-|\frac{L_1 - U_1}{2}| < v_1 - v_2 < |\frac{L_1 - U_1}{2}|$. First we prove the second part of this inequality. Using $\vec{p_2}$ in *OBJ1* and the *approximated objective expression* we obtain $v_1' = d_0 + eval(d_1 \prod_j p_{1j}, \vec{p_2}) + v_2 - d_0 - \widehat{g}$ (where *eval* is a function to evaluate the term with the assigned values). Because v_1 is optimal $v_1 \leqslant v_1'$ then:

$$v_1 - v_2 \leqslant eval\left(d_1 \prod_j p_{1j}, \vec{p_2}\right) - \widehat{g}. \tag{17}$$

Additionally for any possible legal values of $\vec{p}$ and for $\vec{p_2}$, $|eval(d_1 \prod_j p_{1j}, \vec{p_2}) - \widehat{g}| < |\frac{L_1 - U_1}{2}|$, i.e., $-|\frac{L_1 - U_1}{2}| < eval(d_1 \prod_j p_{1j}, \vec{p_2}) - \widehat{g} < |\frac{L_1 - U_1}{2}|$. From this equation and (17) we obtain $v1 - v2 < |\frac{L_1 - U_1}{2}|$. The proof of the first inequality follows by the same reasoning, but this time substituting $\vec{p_1}$ into *OBJ2* and using the *non-approximated objective expression*. Thus, we obtain $v_2' = d_0 + \widehat{g} + v_1 - d_0 - eval(d_1 \prod_j p_{1j}, \vec{p_1})$. Because v_2 is optimal $v_2 \leqslant v_2'$ then:

$$v_2 - v_1 \leqslant \widehat{g} - eval\left(d_1 \prod_j p_{1j}, \vec{p_1}\right). \tag{18}$$

Additionally for any possible legal values of $\vec{p}$ and for $\vec{p_1}$, $-|\frac{L_1 - U_1}{2}| < eval(d_1 \prod_j p_{1j}, \vec{p_1}) - \widehat{g} < |\frac{L_1 - U_1}{2}|$. From this equation and (18) we obtain $v1 - v2 > -|\frac{L_1 - U_1}{2}|$.

This bounds the objective approximation error for one term approximation and by simple induction, we can additively bound the accumulated error for multiple approximations as calculated using *curError* in Algorithm 9. □

8. Experimental results

Before we delve into experimental results involving the *SPUDD-IP*, *APRICODD-IP*, and *Objective-IP* algorithms contributed in the previous sections, we begin by describing the factored MDP-IP domains used in our experiments.

8.1. Domains

We perform experiments with three factored MDP-IP domains: Factory [4], SysAdmin [5] and Traffic (a new domain). In the following, we review Factory and introduce the new Traffic domain; SysAdmin was already introduced in Section 4.

8.1.1. Factory *domain*

The Factory domain [4] is based on a manufacturing problem in which connected, finished parts are produced. The parts must be shaped, polished, painted and connected by bolting, welding or gluing them. In particular in Factory domain the agent's task is connect two objects A and B. The agent can choose between the following actions: *shape*(x), *handPaint*(x), *polish*(x), *drill*(x), *weld*(x, y), *dip*(x) (paint x by dipping it), *bolt*(x, y) (connect objects x and y by bolting them) and *glue*(x, y) (connect objects x and y by gluing them) and *sprayPaint*(x). *sprayPaint*(x) yields a lower quality of painting than *handPaint*(x).

The main variables in this domain are:

- *connected* and *connectedWell*, that represent if objects A and B are simply connected (e.g. by gluing) or are well connected (e.g. by welding them). The only reason for the objects well connected became not connected is when the agent shapes one of them.
- *apainted*, *bpainted*, *apaintedWell* and *bpaintedWell* are variables to represent the painted state of the object respectively. Painted object remains painted if it is not shaped, polished or drilled.
- *ashaped bshaped* represent object A shaped and B shaped respectively. Shaped part remains shaped if it is not drilled.
- *asmooth* and *bsmooth*, an object becomes smoothed if the agent execute the action polish and it succeeds. Smoothed object remains smoothed if it is not shaped or drilled.
- *adrilled* and *bdrilled*, an object becomes drilled if the action drill is apply and it succeeds.

There are other variables that describe the things that are available in the environment to be used by the agent such as: *spraygun*, *glue*, *bolts*, *drill* and *clamps*. Additionally, the variable *skilledlab* represents the existence of skilled labor.

The quality required for the finished product is represented by the variable *typeneeded* and can be high-quality or low-quality. The process and the reward depend directly on the quality required. For example, when high-quality is required, hand-painted, drilled and bolted objects will have more reward while spray-painted and glued objects will obtain little reward. Additional variables can be included in the problem to generate different instances.

To obtain a factored MDP-IP, we introduce uncertainty in the *bolt* action for the variable *connected* as follows. The success probability of the *bolt* action for two objects that are not connected before, when there are bolts, A is drilled and B is drilled is p_1. In the case when the two objects are not drilled but there are bolts, the success probability is p_2. These probabilities are constrained by $0.2 + p_2 \leqslant p_1 \leqslant 1$ and $0.5 \leqslant p_2 \leqslant 1$. Note that p_1 should always be an equal or higher probability than p_2 (since the process associated with p_1 is more likely to succeed), hence the implied constraint $p_2 \leqslant p_1$.

8.1.2. A new domain: Traffic

We introduce Traffic, a factored MDP-IP domain motivated by a real traffic intersection control problem modeled using *cellular transition model dynamics* [21]. While this is not meant to be an accurate large-scale traffic model over long stretches of road, it should still approximately model local traffic propagation at busy intersections where speeds are necessarily limited by queuing and traffic turn delays.

A graphical representation with examples of state variables are given in Fig. 14. We encode our traffic state as $\vec{x} = (x_1, \ldots, x_n)$ where $\vec{x} \in \{O, U\}^n$ indicating that each traffic cell x_i ($1 \leqslant i \leqslant n$) is either occupied O or unoccupied U.

Our basic traffic model for *intermediate road cells* is that a car will move forward into the next cell as long as it is unoccupied, otherwise it stops in its current cell and waits.

For each *intersection road cell* x_j (i.e., leading into an intersection), we define a state variable $t_j \in \{turn, no\text{-}turn\}$ indicating whether a car in x_i will intend to turn into oncoming traffic or not. The state variable t_j is drawn randomly with probability p_t that a car will turn when a new car arrives. When determining the update for x_j, we note that it can always go straight or turn left on a green, but whether it can cross the opposing lane to make a right turn depends on the opposing traffic light state and the opposing traffic cell states t_o and x_o (two opposing right-turning cars may safely turn though and this is allowed by conditioning on t_o).

We refer to a boundary traffic cell x_k as a *feeder road cell* since new cars are introduced at these points. We assume that when the cell is not occupied, new cars arrive on a time step with probability p_a.

Finally, we have state variables $\vec{c}$ encoding the current state of the light cycle. The action set is simply to remain in the same state or advance in the sequence: $A = \{advance, no\text{-}change\}$. In Fig. 14, we have $\vec{c} = (c_1, c_2, c_3, c_4)$, where one may interpret each binary c_i as indicating whether the intended light is green (or not). However, each c_i need not be binary, it could have an additional state for the period between green lights before advancing to the next cycle. We need not commit to a particular state sequence here, rather we simply rely on a model-specific function *next-state*$(\vec{c})$ to generate the next state from the current when the lights advance.

With this high-level description, we now proceed to define the DCN, reward, and specific Traffic instance configurations used in this article.

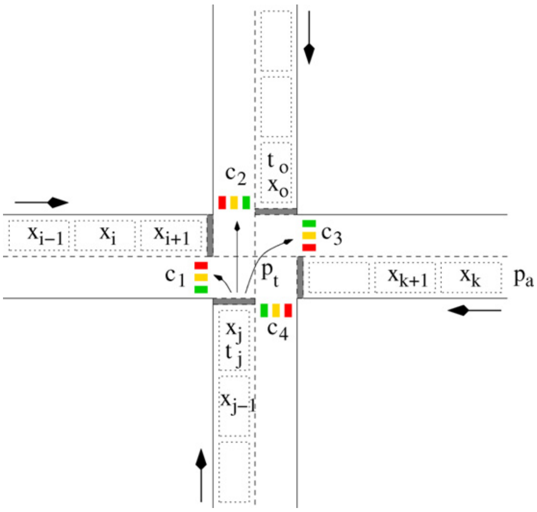
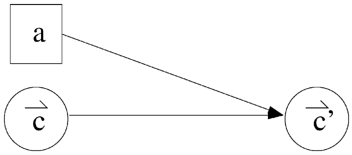


Fig. 14. Diagram showing a 4-way single-lane intersection with cells (dotted boxes) and various state variables used in our state description. Note that we do not model road cells that exit the intersection as we assume that cars freely exit the boundaries of the model once they have passed through the intersection.

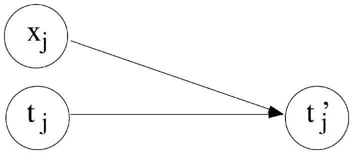
TRAFFIC *DCN transition model.* Based on the above description, the transition model is provided in a compact factored format as a dynamic credal network (DCN) [11,15], subdivided into different functional subcomponents as follows.

Light cycle transition. Here we simply model the effect of a *no-change* or *advance* action on the light state:



$$P\left(\vec{c}'|\vec{c},a\right)=\begin{cases}1.0 & a=\textit{no-change}\wedge\vec{c}'=\vec{c}\\1.0 & a=\textit{advance}\wedge\vec{c}'=\textit{next-state}(\vec{c})\\0.0 & \text{otherwise.}\end{cases}$$

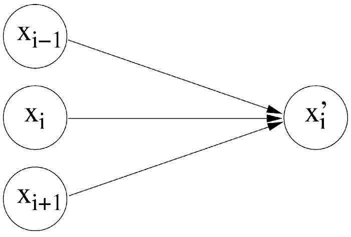
Lane turning indicator. Here we assume that the probability of a car at the head of the queue making a right turn is p_t and that while a car is waiting, its turn decision does not change:



$$P\left(t_j'=\textit{turn}|t_j,x_j\right)=\begin{cases}1.0 & x_j=O\wedge t_j=\textit{turn}\\0.0 & x_j=O\wedge t_j=\textit{no-turn}\\p_t & x_j=U.\end{cases}$$

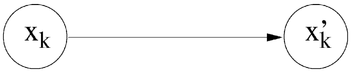
It is difficult in traffic models to obtain an accurate estimate of p_t over all hours of the day, so using our DCN, we allow the turn probability to fluctuate over time and thus model $p_t^{\min}\leqslant p_t\leqslant p_t^{\max}$ for $0\leqslant p_t^{\min}\leqslant p_t^{\max}\leqslant 1$ (to be defined for specific problem instances).

Intermediate road cell. The occupancy of a car in an intermediate road cell x_i depends on whether an occupying car can move forward into the next cell x_{i+1} and if so, whether there is a car in the previous cell x_{i-1} that can move forward to take its place:



$$P\left(x_i' = O \,|\, x_{i-1}, x_i, x_{i+1}\right) = \begin{cases} 1.0 & x_i = O \wedge x_{i+1} = O \\ 1.0 & x_i = U \wedge x_{i-1} = O \\ 0.0 & x_i = \text{otherwise.} \end{cases}$$

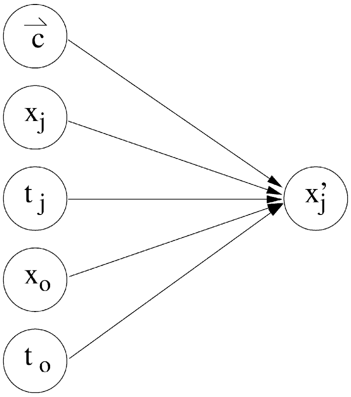
Feeder road cell. A feeder road cell simply serves as an input to the traffic network with cars arriving at each unoccupied feeder cell with probability p_a:



$$P\left(x_k' = O \,|\, x_k\right) = \begin{cases} 1.0 & x_k = O \wedge x_{k+1} = O \\ p_a & \text{otherwise.} \end{cases}$$

It is difficult in traffic models to obtain an accurate estimate of arrival probabilities p_a over all hours of the day, so using our DCN, we allow the arrival probability to fluctuate over time and thus model $p_a^{\min} \leqslant p_a \leqslant p_a^{\max}$ for $0 \leqslant p_a^{\min} \leqslant p_a^{\max} \leqslant 1$ (to be defined for specific problem instances).

Intersection road cell. The intersection road cells are the most complex cells to model in a traffic network as traffic behavior depends on the light state, the occupancy of all cells with green access to the intersection, and the state of turning traffic. Here we attempt to implement a basic model of traffic behavior taking into account all of these contingencies:



$$P\left(x_j' = O \,|\, x_j, t_j, x_o, t_o, \vec{c}\right) = \begin{cases} 0.0 & x_j = U \wedge x_{j-1} = U \\ 1.0 & x_j = U \wedge x_{j-1} = O \\ 0.0 & x_j = O \wedge t_j = \textit{no-turn} \\ 0.0 & x_j = O \wedge t_j = \textit{turn} \wedge x_o = U \\ 0.0 & x_j = O \wedge t_j = \textit{turn} \wedge x_o = O \wedge \neg \textit{green}(\vec{c}, o) \\ 1.0 & x_j = O \wedge t_j = \textit{turn} \wedge x_o = O \wedge \textit{green}(\vec{c}, o) \wedge t_o = \textit{no-turn} \\ 0.0 & x_j = O \wedge t_j = \textit{turn} \wedge x_o = O \wedge \textit{green}(\vec{c}, o) \wedge t_o = \textit{turn}. \end{cases}$$

Here we assume there are user-defined helper functions $\textit{green}(\vec{c}, j)$ that extract the part of the state $\vec{c}$ indicating whether the intersection cell j has a green light (or not). We also assume that when $\neg\textit{green}(\vec{c}, o)$ holds, then $x_j' = x_j$ (thereby making a simplifying assumption of no turns on red).

TRAFFIC *reward model.* Because our goal is to reduce traffic congestion in the intersection, our objective is to minimize the count of occupied road cells around an intersection. Thus, an appropriate reward to maximize would be the count of *unoccupied* cells[5]:

[5] We use $\mathbb{I}[\cdot]$ as an indicator function taking the value 1 when its argument is true and 0 otherwise.

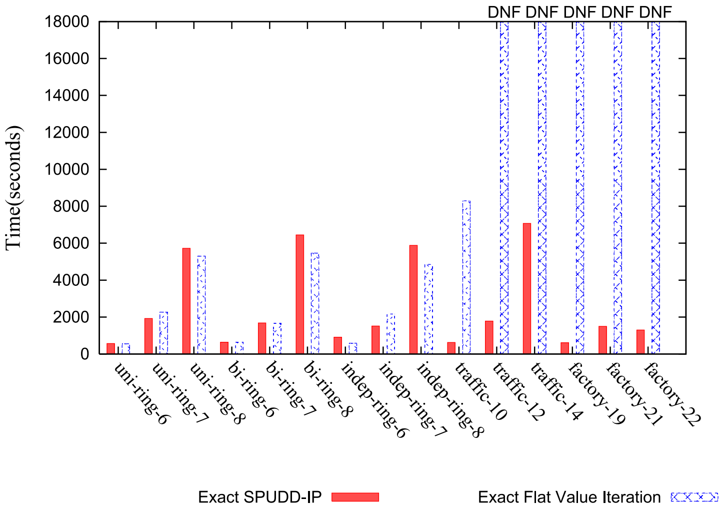


Fig. 15. Time performance comparison for Traffic, SysAdmin and Factory problems using SPUDD-IP and Flat Value Iteration. The name includes the number of variables in each problem, so the corresponding number of states is $2^{\#variables}$.

$$R(\vec{x}) = \sum_{i=1}^{n} \mathbb{I}[x_i = U].$$

Here, we get +1 reward for every cell that is unoccupied.

Traffic *problem instances.* In this article we solve instances of Traffic domain with two opposing lanes. In these particular instances, we set the turn probability minimum as $p_t^{\min} = \mathbf{0}$ and maximum as $p_t^{\max} = \mathbf{1}$ and furthermore constrain the turn probabilities p_1 and p_2 of the two different lanes to be highly correlated using the constraint $|p_1 - p_2| \leqslant 0.1$. Additionally, the probabilities p_3 and p_4 of a car arriving at either of the feeder cells for each lane use the probability bounds $p_a^{\min} = 0.4$ and $p_a^{\max} = 0.6$ and are constrained by $|p_3 - p_4| \leqslant 0.1$.

8.2. Evaluation

In this section, we empirically evaluate four algorithms: *Flat Value Iteration* from (10) and our three contributions from the previous section for solving factored MDP-IPs: (i) *SPUDD-IP* that offers an exact solution; (ii) *APRICODD-IP* and (iii) *Objective-IP* that offer bounded approximate solutions.

As an additional point of comparison, we note that recent years have seen the emergence of very fast approximate factored MDP solvers based on *Approximate Linear Programming* (*ALP*) [5]. Recently, such techniques have been extended to factored MDP-IPs [9]. Thus, we *also* compare the approximate solutions *APRICODD-IP* and *Objective-IP* based on approximate value iteration with an *Approximate Multilinear Programming* (*AMP*) algorithm from [9]. *AMP* performs linear-value function approximation using a fixed set of basis functions and a compact constraint encoding for multilinear optimization problems that exploits structure in the DCN.

For all algorithms, we set $maxIter = 50$ for SysAdmin and $maxIter = 75$ for the other domains with $\gamma = 0.9$. In the next subsections we present our main results.

8.2.1. Flat value iteration vs. SPUDD-IP

In Fig. 15 we compare the running time of the two exact solution methods: SPUDD-IP and Flat Value Iteration which compute $V^*(\vec{x})$.[6] Solutions not completing in five hours are marked *Did Not Finish* (*DNF*). We note that SPUDD-IP did not outperform Flat Value Iteration on the SysAdmin domains because the exact value function has little structure as an ADD. However, both Traffic and Factory had highly structured value functions and *up to two orders magnitude time improvement* is demonstrated by SPUDD-IP, largely due to the ability of the PADDs to aggregate common nonlinear objectives, thus saving a substantial number of calls to the nonlinear solver and therefore time.

8.2.2. APRICODD-IP vs. Objective-IP

In order to see the scalability of our approximate solutions, in Fig. 16 we compare the running time for APRICODD-IP and Objective-IP vs. the number of state variables using $\delta = 0.1$ for Factory, Traffic, and the three configurations of SysAdmin.

[6] We note that to do this comparison, we need to slightly extend Flat Value Iteration algorithm from (10) to allow for multilinear expressions in the transition probability table.

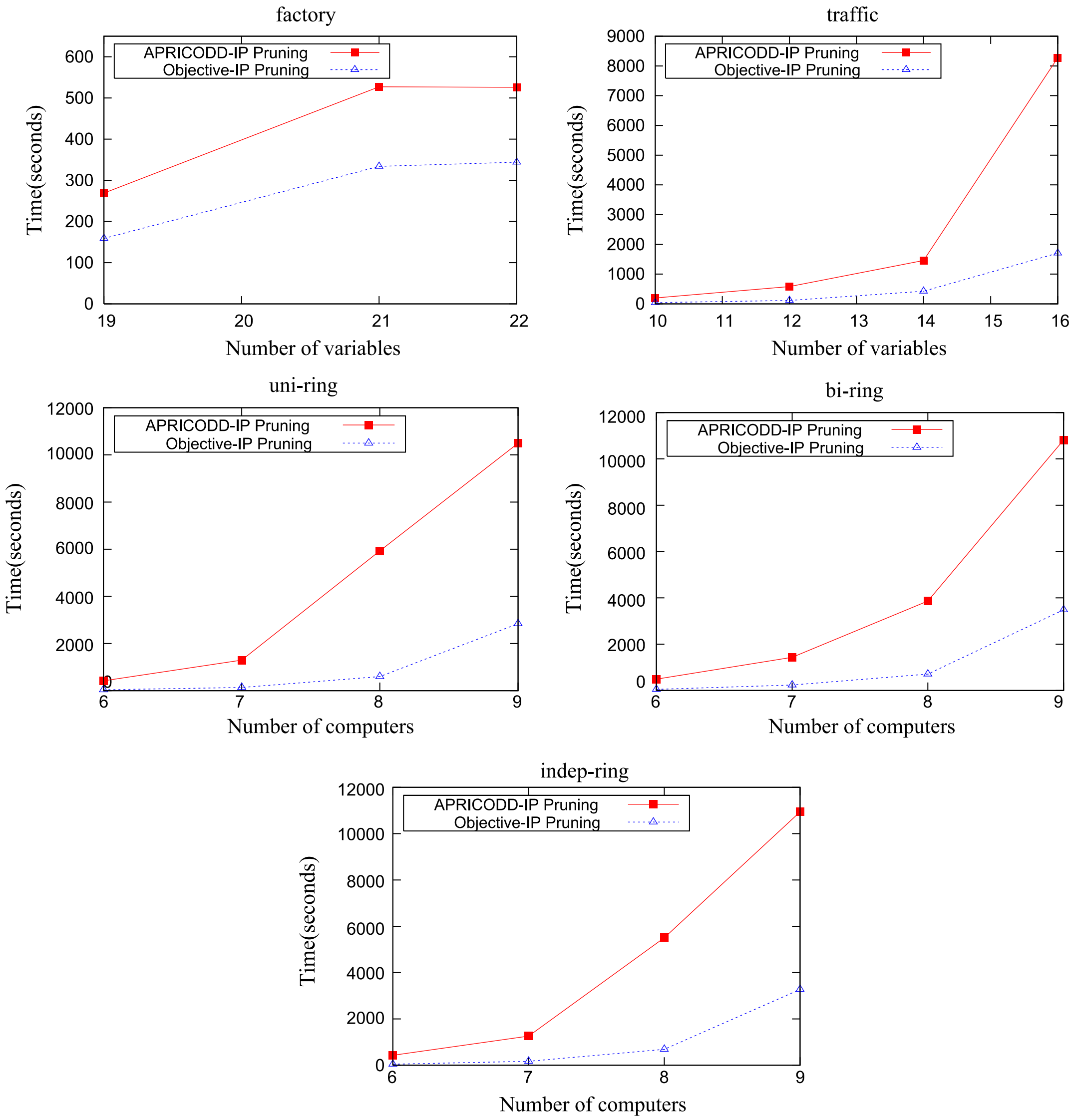


Fig. 16. Time performance of APRICODD-IP and Objective-IP for Traffic, SysAdmin and Factory problems for $\delta = 0.1$.

We note that Objective-IP runs faster than APRICODD-IP in *all* domains when running with a fixed bound on maximum error per iteration (i.e., $\delta = 0.1$).

In order to evaluate the policy returned by our AVI solutions, we compute for each fixed value of δ (δ is the maximum error per iteration w.r.t. V_{max}), the True Approximation Error (TAE) given by:

$$\max_{\vec{x}} \left| V^*(\vec{x}) - V_{approx}(\vec{x}) \right| \tag{19}$$

where $V_{approx}(\vec{x})$ is the value returned by APRICODD-IP or Objective-IP and $V^*(\vec{x})$ is the optimal value computed by SPUDD-IP.

In the following plots we ran *Solve* for a range of δ. In Figs. 17 and 18 we present a detailed comparison of the time, size (number of nodes in the ADD of the last iteration), and number of nonlinear solver calls required by APRICODD-IP and Objective-IP plotted vs. the TAE for *traffic-10*, respectively. We note little relationship between the space required by the ADD value representation (number of nodes) and the running times of the two algorithms (space actually increases slightly for Objective-IP while running time decreases, see Fig. 18). But what is striking about these plots is that the running time of each algorithm is directly correlated with the number of nonlinear solver calls made by the algorithm (taking up to 100 ms in some cases), reflecting our intuitions that the time complexity of solving MDP-IPs is governed by the computational overhead of nonlinear optimization.

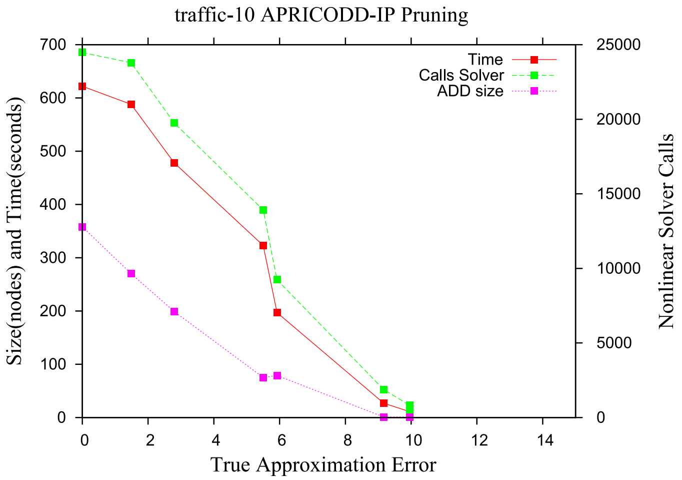


Fig. 17. Time, nonlinear solver calls and ADD size of APRICODD-IP pruning for the traffic problem with 10 variables. Results are plotted for $\delta \in \{0.0, 0.025, 0.05, 0.075, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$.

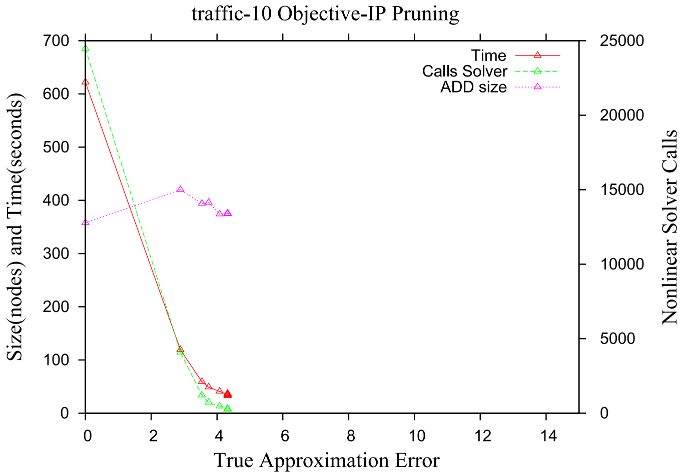


Fig. 18. Time, nonlinear solver calls and ADD size of Objective-IP pruning for the traffic problem with 10 variables. Results are plotted for $\delta \in \{0.0, 0.025, 0.05, 0.075, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$.

Fig. 19 shows the advantage of Objective-IP pruning that uses the upper and lower values to approximate the leaves in PADDs. For all problems, as the number of nodes reduced to a constant grows, we see that the True Approximation Error increases, but also the number of calls to the multilinear solver decreases. These figures also show cases where the Objective-IP approach to PADD reduction occurs with great success, since the original PADD sizes for the exact cases are very large, but can be reduced by orders of magnitude in exchange for a reasonable amount of approximation error.

In Figs. 20, 21, 22, 23 and 24 we show a comparison of the True Approximation Error (TAE) vs. running times for three problems and three different sizes of each problem (varying δ). The results here echo one conclusion: Objective-IP consistently takes less time than APRICODD-IP to achieve the same approximation error and *up to one order of magnitude less time* than APRICODD-IP. This time reduction can be explained by the decreased number of calls to the multilinear solver.

8.2.3. Approximate value iteration vs. approximate multilinear programming

In Figs. 20, 21, 22, 23 and 24 we compare the two approximate solution methods, APRICODD-IP and Objective-IP, with our implementation of approximate multilinear programming (AMP) [9] for MDP-IPs. We used *simple* basis functions (one for each variable in the problem description) and *pairwise* basis functions (one for each pair of variables that have a common child variable in the DCN).

When it does finish within a limit of ten hours, AMP takes only a few seconds to produce an approximate solution for each problem (except for the Factory domain for which it did not return a solution). Comparing the algorithms in terms of their true approximation error, we observe that: (a) in the SysAdmin problem (Figs. 22, 23, 24), AMP with *pair* basis functions outperforms APRICODD-IP and obtains a solution 2–3× larger than the error of Objective-IP, but in significantly less time; (b) for the Traffic problem (Fig. 21), AMP with the *simple* basis solution obtains a solution with 2–3× more error

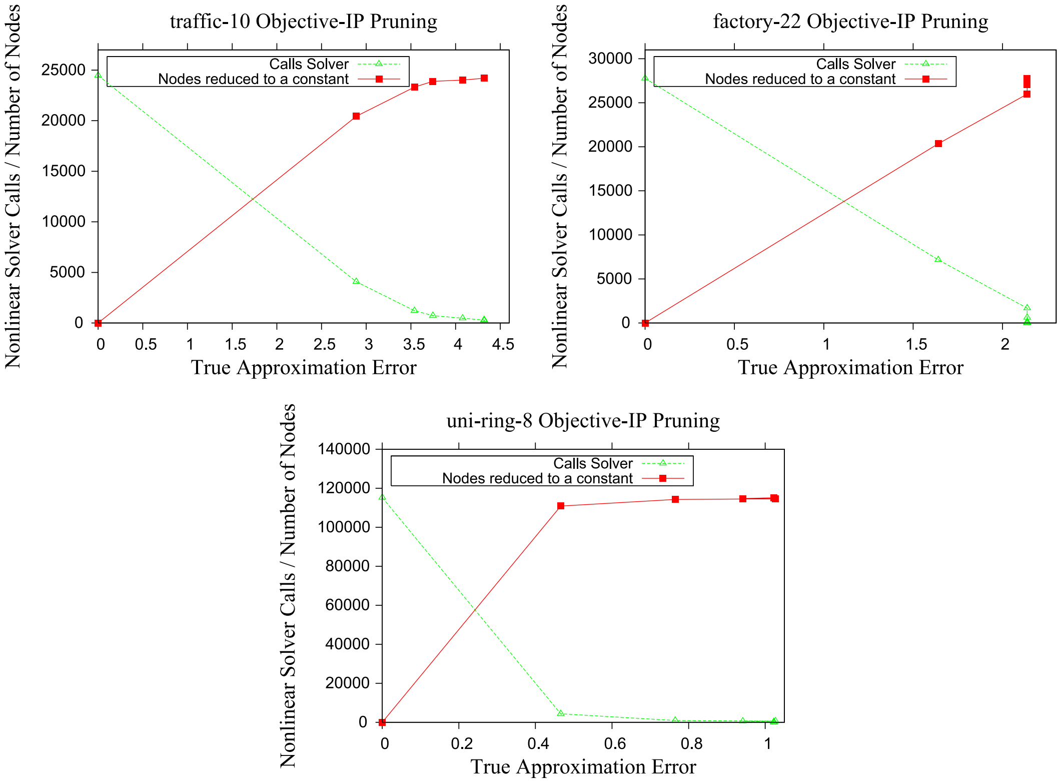


Fig. 19. Number of nonlinear solver calls and number of nodes reduced to a constant vs. True Approximation Error for Objective-IP pruning for three different problems. Results are plotted for $\delta \in \{0.0, 0.025, 0.05, 0.075, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$.

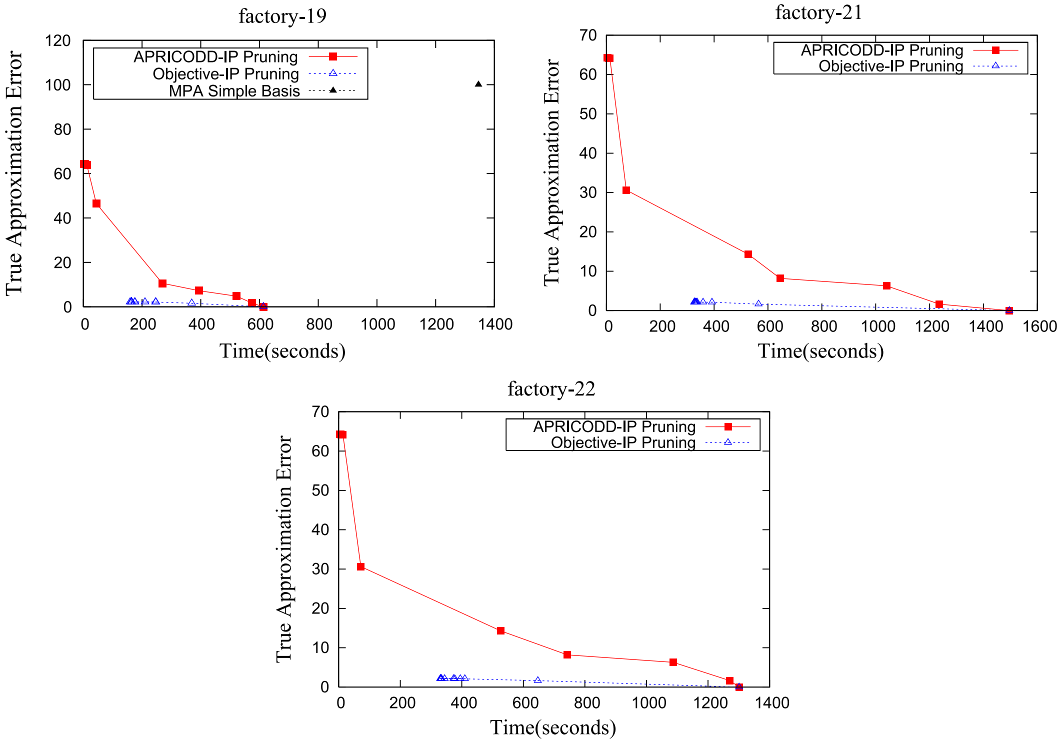


Fig. 20. True Approximation Error vs. time required for APRICODD-IP, Objective-IP and MPA with simple basis functions (MPA pairwise did not finish in a ten hour time limit and MPA with simple basis functions did not finish for two problems) for three Factory problems.

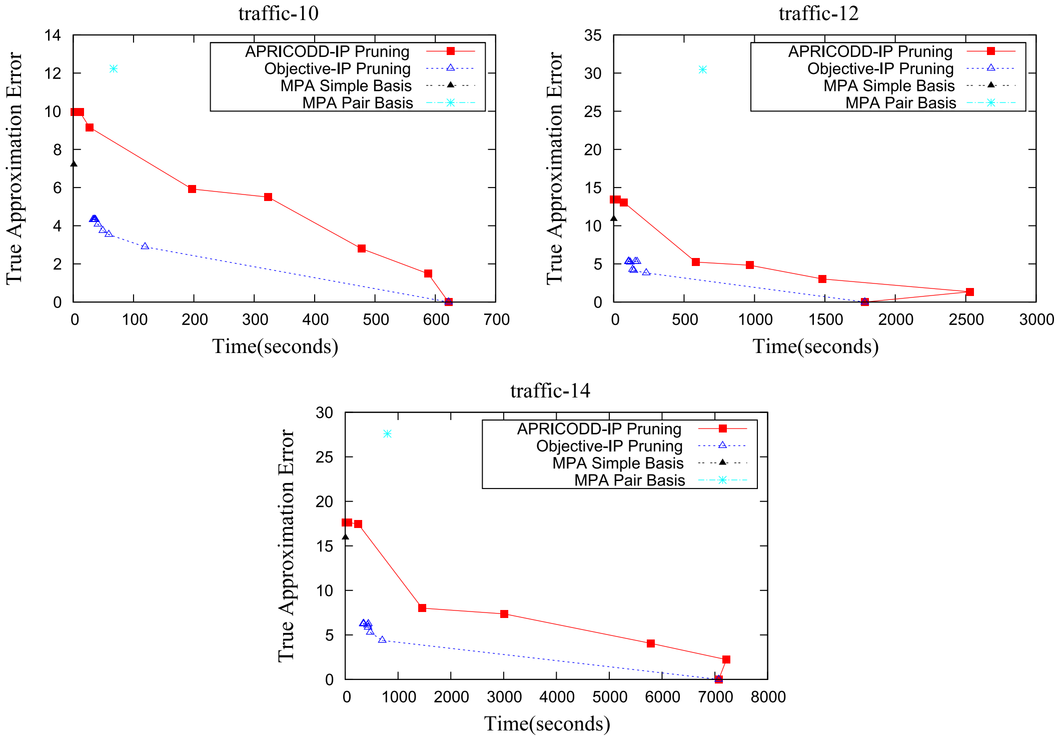


Fig. 21. True Approximation Error vs. time required for APRICODD-IP, Objective-IP and MPA with simple basis and pairwise basis functions for Traffic problem.

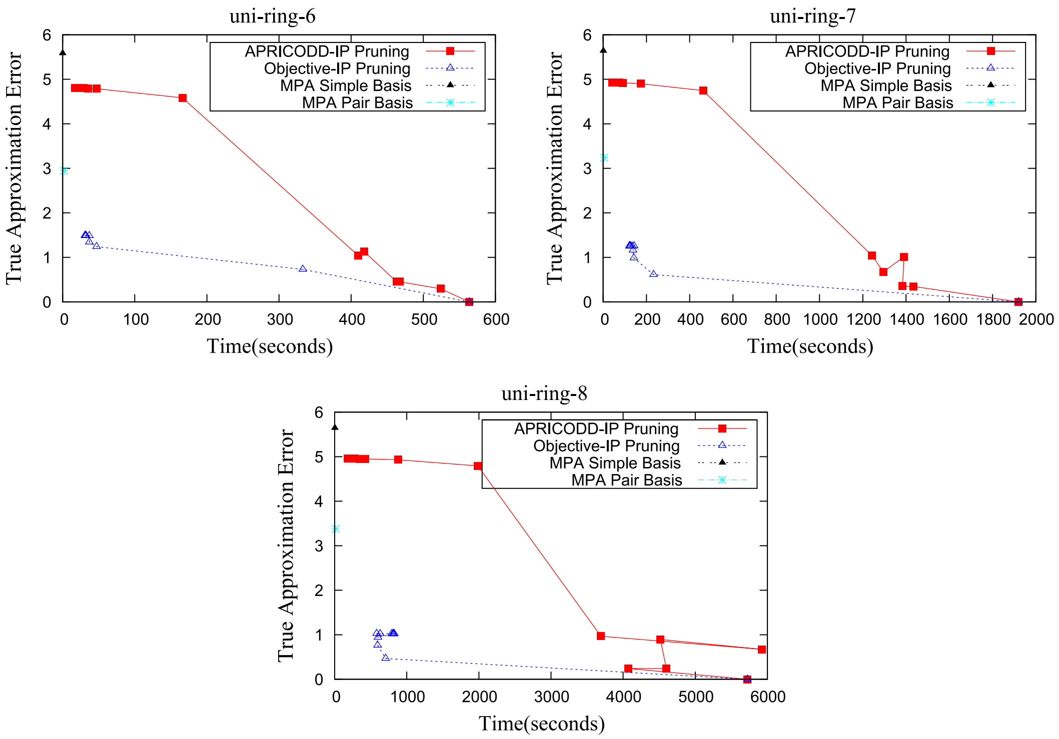


Fig. 22. True Approximation Error vs. time required for APRICODD-IP, Objective-IP and MPA with simple basis and pairwise basis functions for SysAdmin problem with unidirectional-ring topology.

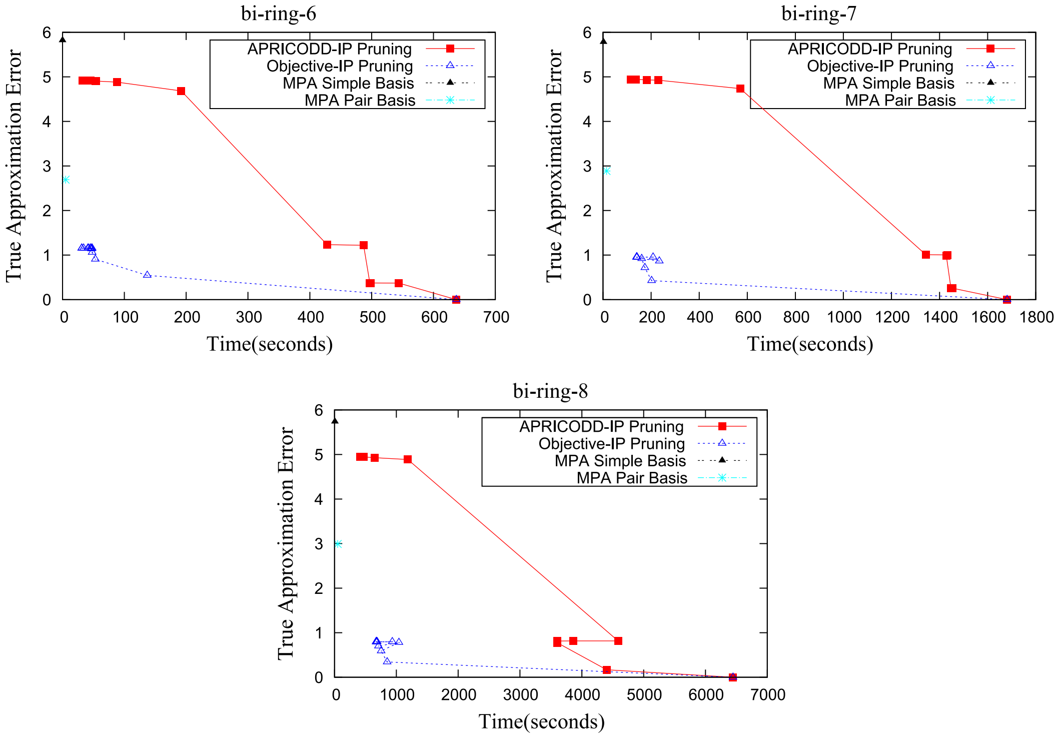


Fig. 23. True Approximation Error vs. time required for APRICODD-IP, Objective-IP and MPA with simple basis and pairwise basis functions for SysAdmin problem with bidirectional-ring topology.

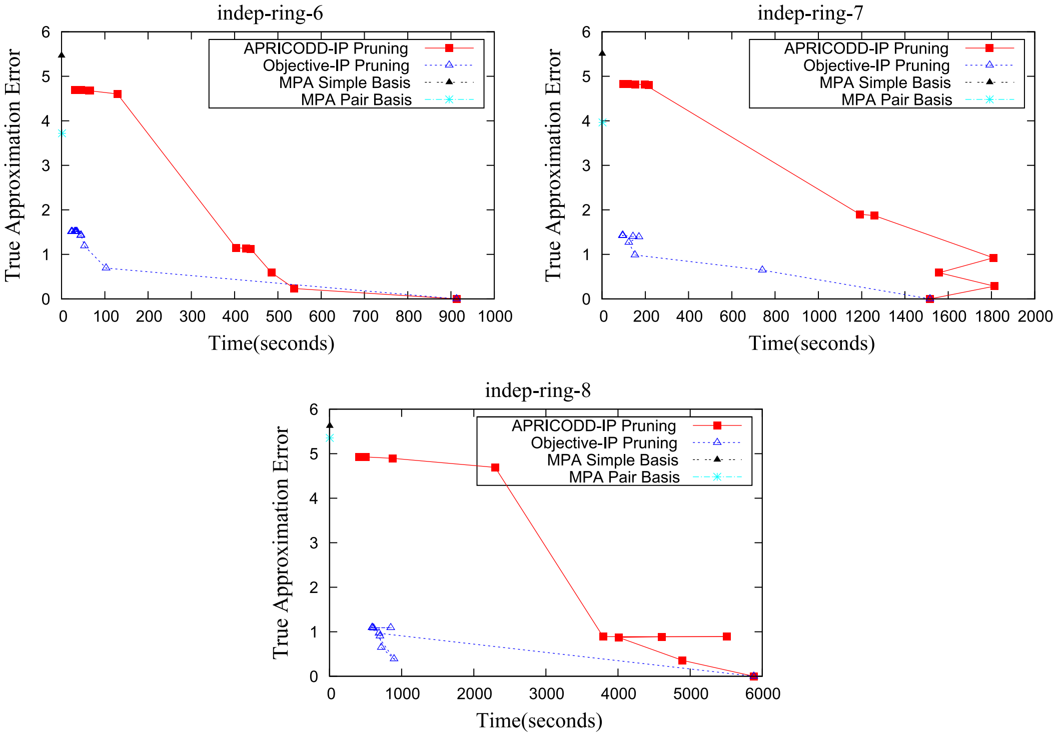


Fig. 24. True Approximation Error vs. time required for APRICODD-IP, Objective-IP and MPA with simple basis and pairwise basis functions for SysAdmin problem with independent bidirectional topology.

than Objective-IP, still in significantly less time; and (c) in the case of the Factory problem (Fig. 20), AMP only can solve one instance, while Objective-IP can solve the rest within the time limit with much lower error. These results lead us to conclude that Objective-IP consistently gives an error at least 2–3× lower than AMP and sometimes runs as fast as the AMP solution, while in other cases running slower.

8.2.4. Results summary

Over all problems, given the unpredictable performance of AMP (which has no error guarantees and often does not finish within the time limit) and the consistently worse performance of APRICODD-IP compared to Objective-IP, Objective-IP stands out as the more reliable option: it offers guaranteed error bounds and empirically it offers consistently lower error rates (the lowest of any algorithm) *with* overall reasonable running times (if not the fastest).

9. Related work

The *Bounded-parameter Markov Decision Process* (*BMDP*) [22] is a special case of an MDP-IP, where the probabilities and rewards are specified by constant intervals. Exploiting the specific structure available in a BMDP given by the intervals, the algorithm in [22] can directly derive the solution without requiring expensive optimization techniques. Recent solutions to BMDPs include extensions of real-time dynamic programming (RTDP) [23] and LAO* [24,25] that search for the best policy under the worst model. The *Markov Decision Process with Set-valued Transitions* (*MDPSTs*) [26] is another subclass of MDP-IPs where probability distributions are given over finite sets of states. Since BMDP and MDPST are special cases of MDP-IPs, we can represent both by "flat" MDP-IPs. Then the algorithms defined in this paper clearly apply to both BMDPs and MDPSTs, however *their solutions do not generalize to the factored MDP-IPs* we examined in this paper, which allow a *multilinear* probability representation resulting from the use of a DCN. Furthermore, MDP-IPs allow for general linear constraints between probabilities, which are prohibited in interval bounded probability settings like BMDPs. This use of general linear constraints is particularly useful when we do not know the probabilities, but only relative constraints between them (e.g., two probabilities in the Traffic problem are unknown but highly correlated).

Previous work on "flat" MDP-IPs [6,7,27] focused on credal sets (represented as polytopes) proposed algorithms based on dynamic programming, but they only solved very small problems. It is important to notice that our factored MDP-IP model is more expressive than the simple "flat" MDP-IPs referred to in those papers; as we saw in Section 4, the joint DCN transition probabilities in factored MDP-IPs may be nonlinear, while for flat MDP-IPs, the transition probability for any next state, given a previous state and action, can only be trivially linear (a single parameter p_i).

As we have discussed in Section 8, it is interesting to note that if we allow *only* interval bounds on the parameters in the CPTs of the DCN of the factored MDP-IP then the result is still a more expressive model than a "flat" MDP-IP or BMDP, i.e., the transition expression for any next state given a previous state and action can be a multilinear expression of $\vec{p}$. Consequently, to define *Flat Value Iteration* for the comparative analysis from the previous section, we note that we already needed to slightly extend previous work to allow for multilinear expressions in the transition probability tables required to match the expressivity of factored MDP-IPs.

A final piece of work that is related with MDP-IPs is a *two-player zero-sum alternating Markov Game* [28] (a.k.a. a *Stochastic Game* [29]). This is a subset of "flat" MDP-IPs *if* intermediate state variables are introduced to represent opponent actions and the parameters specify the distribution over opponent actions is allowed to vary in the full interval $[0, 1]$. However, it might be computationally wasteful to use a "flat" or factored MDP-IP algorithm to solve a Stochastic Game since a minimization over a finite set of opponent actions would likely be computationally cheaper than a (nonlinear) optimization over the probability parameters required in an MDP-IP solution. Hence, it seems more computationally advantageous to use specialized algorithms for the solution of finite action Stochastic Games to exploit the specific structure found there than to attempt to use any of the more general-purpose MDP-IP algorithms presented here.

Finally, probability trees were also used to represent convex sets of probabilities associated to intervals to obtain posterior intervals of probability [30]. Probability trees can compactly represent *context-specific independence* (CSI), but as we saw in Section 5, our parameterized ADDs are DAGs that not only exploit CSI but also shared function structure. Additionally, we used PADDs to represent general probability expressions (multilinear for the case of factored MDP-IPs), not just probability intervals.

10. Concluding remarks

Motivated by the real-world need to solve MDPs with uncertainty in the transition model, we made a number of novel contributions to the literature in this article. In Section 4, we introduced the factored MDP-IP model based on Dynamic Credal Networks (DCNs). In Section 5, we contributed the novel parameterized ADD (PADD) data structure containing leaves with parameterized expressions; we showed how to efficiently obtain a minimal canonical representation of a PADD; and we showed how to efficiently perform a variety of unary and binary operations on PADDs. In Section 6, we contributed the exact factored MDP-IP solution algorithm SPUDD-IP and showed how to efficiently make use of the PADD in all steps of this factored MDP-IP value iteration algorithm. The resulting SPUDD-IP algorithm yielded up to two orders of magnitude speedup over existing value iteration techniques for MDP-IPs.

To further enhance the SPUDD-IP algorithm, in Section 7, we contributed two novel approximate value iteration extensions: APRICODD-IP and Objective-IP. While APRICODD-IP is the obvious extension based on previous work, it did not specifically target the main source of time complexity for solving MDP-IPs — calls to the nonlinear solver during MDP-IP value iteration. Based on this observation, we developed an alternate and novel approximation method that directly approximated the objective of multilinear solver calls, proving the theoretical correctness of this innovative bounded error approximation approach and substantially reducing the number of nonlinear solver calls and thus running time of approximate value iteration. In Section 8, we performed comparisons of the above algorithms to a previously existing "flat" value iteration algorithm as well as a state-of-the-art approximate multilinear programming (AMP) solver for MDP-IPs.

Altogether these novel contributions — and particularly their culmination in the Objective-IP algorithm — enable the (bounded approximate) solution of factored MDP-IPs that can *scale orders of magnitude* beyond existing flat value iteration approaches to MDP-IPs and yield *substantially lower errors* than other existing approximate MDP-IP solvers like approximate multilinear programming (AMP) that have no *a priori* error guarantees and depend on appropriate basis function generation algorithms.

For future work, we note that PADDs represent the tip of the iceberg in the use of advanced decision diagram techniques for solving factored MDP-IPs. Following the success of the Affine extension of ADDs for solving factored MDPs [31] with additive and multiplicative structure, it would be interesting to extend this technique to PADDs to exploit the same structure in MDP-IPs. Such advances would ideally reduce the running time of solutions for factored MDP-IP problems like TRAFFIC that contains significant additive structure in their reward definition and might be amenable to even further exploitation of factored MDP-IP problem structure.

Finally, we note that the exploration of objectives other than maximin optimality for factored MDP-IPs would also be interesting. Although the maximin criteria works fine in a domain with many imprecise parameters (like in the SYSADMIN domain we have used in our experiments), we observe that for a problem with large imprecision in terms of very loose constraints (e.g. $0.1 \geqslant p_{ij} \geqslant 0.9$), the maximin criterion may be *too* adversarial — it may reflect a worst-case that would be extremely unlikely in practice. Hence, future work might also examine other methods of handling transition uncertainty, such as a Bayesian approach [32], and determine whether factored MDP-IPs and PADDs could enhance solution approaches for those alternate criteria.

Acknowledgements

This work was performed while the first author was visiting NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. This work has also been supported by the Brazilian agencies FAPESP (under grant 2008/03995-5) and CAPES.

References

[1] M.L. Puterman, Markov Decision Processes, Wiley Series in Probability and Mathematical Statistics, John Wiley and Sons, New York, 1994.
[2] C. Boutilier, S. Hanks, T. Dean, Decision-theoretic planning: Structural assumptions and computational leverage, JAIR 11 (1999) 1–94.
[3] J. Hoey, R. St-Aubin, A. Hu, C. Boutilier, SPUDD: Stochastic planning using decision diagrams, in: Proceedings UAI, Morgan Kaufmann, 1999, pp. 279–288.
[4] R. St-Aubin, J. Hoey, C. Boutilier, APRICODD: Approximate policy construction using decision diagrams, in: Proceedings NIPS, MIT Press, 2000, pp. 1089–1095.
[5] C. Guestrin, D. Koller, R. Parr, S. Venkataraman, Efficient solution algorithms for factored MDPs, JAIR 19 (2003) 399–468.
[6] J.K. Satia, R.E. Lave Jr., Markovian decision processes with uncertain transition probabilities, Oper. Res. 21 (1970) 728–740.
[7] C.C. White III, H.K. El-Deib, Markov decision processes with imprecise transition probabilities, Oper. Res. 42 (4) (1994) 739–749.
[8] T. Dean, K. Kanazawa, A model for reasoning about persistence and causation, Comput. Intell. 5 (3) (1990) 142–150.
[9] K.V. Delgado, L.N. de Barros, F.G. Cozman, R. Shirota, Representing and solving factored Markov decision processes with imprecise probabilities, in: Proceedings ISIPTA, Durham, United Kingdom, 2009.
[10] D.P. Bertsekas, J.N. Tsitsiklis, An analysis of stochastic shortest path problems, Math. Oper. Res. 16 (3) (1991) 580–595.
[11] F.G. Cozman, Credal networks, Artificial Intelligence 120 (2000) 199–233.
[12] P. Walley, Statistical Reasoning with Imprecise Probabilities, Chapman and Hall, London, 1991.
[13] A. Nilim, L. El Ghaoui, Robust control of Markov decision processes with uncertain transition matrices, Oper. Res. 53 (5) (2005) 780–798.
[14] R. Shirota, F.G. Cozman, F.W. Trevizan, C.P. de Campos, L.N. de Barros, Multilinear and integer programming for Markov decision processes with imprecise probabilities, in: Proceedings ISIPTA, Prague, Czech Republic, 2007, pp. 395–404.
[15] F.G. Cozman, Graphical models for imprecise probabilities, Internat. J. Approx. Reason. 39 (2–3) (2005) 167–184.
[16] C. Boutilier, N. Friedman, M. Goldszmidt, D. Koller, Context-specific independence in Bayesian networks, in: Proceedings UAI, 1996, pp. 115–123.
[17] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, F. Somenzi, Algebraic decision diagrams and their applications, in: Proceedings ICCAD, IEEE Computer Society Press, Los Alamitos, CA, USA, 1993, pp. 188–191.
[18] R.E. Bryant, Symbolic Boolean manipulation with ordered binary-decision diagrams, ACM Comput. Surv. 24 (3) (1992) 293–318.
[19] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput. 35 (8) (1986) 677–691.
[20] N.L. Zhang, D. Poole, A simple approach to Bayesian network computations, in: Proceedings of the Tenth Canadian Conference on Artificial Intelligence, 1994, pp. 171–178.
[21] C.F. Daganzo, The cell transmission model: a dynamic representation of highway traffic consistent with the hydrodynamic theory, Transport. Res. B 28 (4) (1994) 269–287.
[22] R. Givan, S. Leach, T. Dean, Bounded-parameter Markov decision processes, Artificial Intelligence 122 (2000) 71–109, (39).
[23] O. Buffet, D. Aberdeen, Robust planning with LRTDP, in: Proceedings IJCAI, 2005, pp. 1214–1219.
[24] S. Cui, J. Sun, M. Yin, S. Lu, Solving uncertain Markov decision problems: an interval-based method, in: Proceedings ICNC (2), 2006, pp. 948–957.

[25] M. Yin, J. Wang, W. Gu, Solving planning under uncertainty: quantitative and qualitative approach, in: Proceedings IFSA (2), 2007, pp. 612–620.
[26] F.W. Trevizan, F.G. Cozman, L.N. de Barros, Planning under risk and knightian uncertainty, in: Proceedings IJCAI, 2007, pp. 2023–2028.
[27] J.A. Bagnell, A.Y. Ng, J.G. Schneider, Solving uncertain Markov decision processes, Tech. rep., Carnegie Mellon University, 2001.
[28] M.L. Littman, Markov games as a framework for multi-agent reinforcement learning, in: Proceedings ICML, Morgan Kaufmann, 1994, pp. 157–163.
[29] L.S. Shapley, Stochastic games, Proc. Natl. Acad. Sci. USA 39 (1953) 327–332.
[30] A. Cano, S. Moral, Using probability trees to compute marginals with imprecise probabilities, Internat. J. Approx. Reason. 29 (1) (2002) 1–46.
[31] S. Sanner, D. McAllester, Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference, in: Proceedings IJCAI, 2005, pp. 1384–1390.
[32] M.O. Duff, Optimal learning: Computational procedures for Bayes-adaptive Markov decision processes, Ph.D. thesis, University of Massachusetts, Amherst, January 2002.