Inference and Learning in Dynamic Decision Networks Using Knowledge Compilation

Gabriele Venturato[1], Vincent Derkinderen[1], Pedro Zuidberg Dos Martires[2], Luc De Raedt[1,2]

[1]KU Leuven, Belgium
[2]Örebro University, Sweden
firstname.lastname@kuleuven.be, pedro.zuidberg-dos-martires@oru.se

Abstract

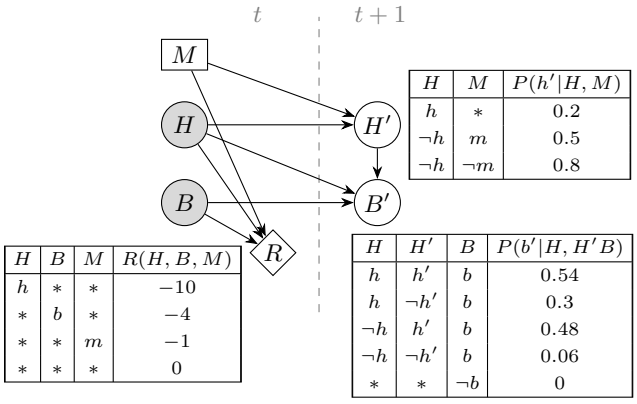
Decision making under uncertainty in dynamic environments is a fundamental AI problem in which agents need to determine which decisions (or actions) to make at each time step to maximise their expected utility. Dynamic decision networks (DDNs) are an extension of dynamic Bayesian networks with decisions and utilities, and can be used to compactly represent Markov decision processes (MDPs). We propose a novel algorithm called `mapl-cirup` that leverages knowledge compilation techniques developed for (dynamic) Bayesian networks to perform inference and gradient-based learning in DDNs. Specifically, we knowledge-compile the Bellman update present in DDNs into dynamic decision circuits and evaluate them within an (algebraic) model counting framework. In contrast to other exact symbolic MDP approaches, we obtain differentiable circuits that enable gradient-based parameter learning.

1 Introduction

Bayesian networks (BNs) have been widely adopted to model real-world processes under uncertainty (Koller and Friedman 2009; Russell and Norvig 2020). Unfortunately, inference in BNs is computationally hard (#P-hard) (Roth 1996). To deal with this hardness, state-of-the-art techniques apply knowledge compilation (KC), exploiting conditional independencies as well as local structures (Chavira and Darwiche 2008). More recently, KC has also been successfully applied for inference in dynamic models (Vlasselaer et al. 2016), exploiting not only local, but also repeated structure over time.

Dynamic decision networks (DDNs) combine dynamic (Dean and Kanazawa 1989; Murphy 2002) and decision-theoretic (Howard and Matheson 1984; Bhattacharjya and Shachter 2007) Bayesian networks into a single modelling language (Kanazawa and Dean 1989; Russell and Norvig 2020), capable of representing, for instance, Markov decision processes (MDPs) (Bellman 1957).

Example 1 (Monkey DDN, Figure 1)**.** *A monkey tries to hit (H) you with suspicious mud. You can decide to move (M) or not. If you get hit, you might smell bad (B), but the monkey celebrates and is less likely to hit you in the next time step. Your decision and current state hence influence the next state (primed variables). If the monkey misses, it gets angrier and focuses more on its task, increasing the probability to hit you. Moving decreases the probability of being hit. There is a reward (R) associated with every state and decision.*



H	M	$P(h'\|H,M)$
h	$*$	0.2
$\neg h$	m	0.5
$\neg h$	$\neg m$	0.8

H	B	M	$R(H,B,M)$
h	$*$	$*$	-10
$*$	b	$*$	-4
$*$	$*$	m	-1
$*$	$*$	$*$	0

H	H'	B	$P(b'\|H,H'B)$
h	h'	b	0.54
h	$\neg h'$	b	0.3
$\neg h$	h'	b	0.48
$\neg h$	$\neg h'$	b	0.06
$*$	$*$	$\neg b$	0

Figure 1: DDN for Example 1. DDNs provide a graphical representation of the dependencies between variables at time t and $t+1$, the latter represented by using a primed symbol such as B'. For DDNs representing MDPs, the current state is fully observed, *i.e.*, all non primed state variables are evidence (gray nodes). The '$*$' in the tables represent unspecified truth values. The reward table represents an additive reward function (Russell and Norvig 2020, Chapter 16.4.2).

Derkinderen and De Raedt (2020) explored a KC approach to model and solve a single-stage maximisation problem under uncertainty. Crucially, they leveraged the algebraic model counting framework (Kimmig, Van den Broeck, and De Raedt 2017) where compiling once allows for solving a variety of tasks on top of the same compiled diagram. In this work we investigate how to apply a similar approach to the dynamic setting, solving dynamic decision networks. To illustrate the benefit of applying the algebraic model counting framework, we then use the same compiled diagram of the decision task for parameter learning.

Our main contribution is the introduction of `mapl-cirup` (`Ma`rkov `pl`anning with `cir`cuit `up`dates), read as 'maple syrup'. `mapl-cirup` is a variation of the classic value iteration algorithm that uses KC to exploit

dependencies and repeated temporal structures. We adapt the idea by Chavira and Darwiche (2008) to encode BNs as weighted logic formulas and knowledge-compile them into arithmetic circuits for efficient inference. Specifically, we introduce dynamic decision circuits (DDCs) and use them as a compilation target for DDNs. This reduces planning to probabilistic inference in DDCs. As a second contribution we show that the resulting differentiable representation can be used for gradient-based parameter learning in MDPs.

2 Preliminaries

2.1 Model Counting and Knowledge Compilation

A prominent technique to perform probabilistic inference with discrete random variables is reducing it to weighted model counting (WMC) (Darwiche 2009). That is, encoding a specific probabilistic inference problem as a weighted propositional logic formula. Computing the weight of the formula is then equivalent to computing the target probability. Chavira and Darwiche (2008), for instance, used this technique to perform inference in BNs. Similar to these works, we only focus on Boolean variables, and encode multi-valued variables using a combination of Boolean ones.

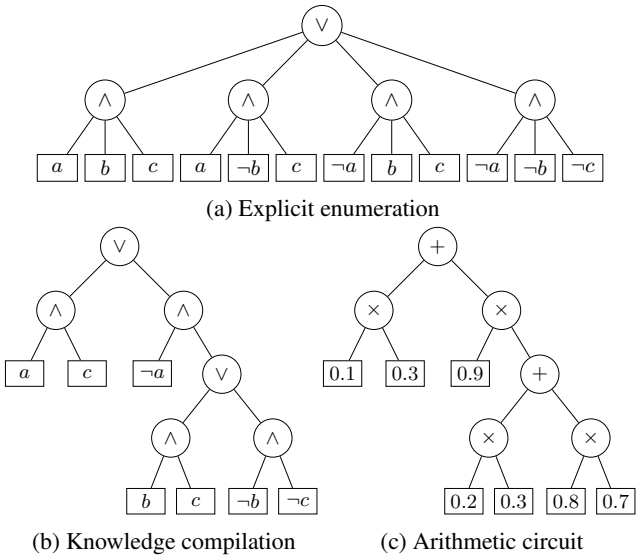
Example 2 (Weighted Model Counting)**.** *Consider the theory $T = C \leftrightarrow A \vee B$, where we use weight function $w : \{a \mapsto 0.1, \neg a \mapsto 1 - 0.1, b \mapsto 0.2, \neg b \mapsto 1 - 0.2, c \mapsto 0.3, \neg c \mapsto 1 - 0.3\}$. The weighted model count of T is then:*

$$\begin{aligned} WMC(T, w) &= w(a)w(b)w(c) + w(a)w(\neg b)w(c) + \\ &\quad w(\neg a)w(b)w(c) + w(\neg a)w(\neg b)w(\neg c) \\ &= 0.006 + 0.024 + 0.054 + 0.504 = 0.588 \end{aligned}$$

In case the weights have a probabilistic interpretation, the weight 0.588 is the probability of the theory being satisfied.

Unfortunately, performing WMC on propositional logic formulas is a computational hard task, #P-hard to be precise (Valiant 1979). We can see this intuitively in Example 2 where we explicitly enumerate all the satisfiable states. Of course it is infeasible in practice to enumerate all satisfiable states one by one. So instead, a state-of-the-art technique to solving this is through knowledge compilation (KC) (Darwiche 2002) — an approach also deployed by Chavira and Darwiche (2008). The core idea is simple: take a propositional logic formula and compile it into a representation where WMC, and consequently probabilistic inference, can be performed in polytime (with respect to the size of the target representation). A key advantage of this approach is that the compilation is weight agnostic. This means that multiple WMC problems can be addressed using a single compilation step, as long as those problem instances share the same logic formula. This amortises the cost of (offline) compilation, which is still computationally hard, over the multiple (online) query steps that follow. This is especially useful for repeated inference.

Example 3 (KC)**.** *The logic theory from Example 2 can be represented as a directed acyclic graph (DAG) trivially enumerating all the possible models (Figure 2a). However, using knowledge compilation one can represent the same theory in a more compact way (Figure 2b), while preserving the efficient WMC computation. Substituting the $\wedge$ and $\vee$ connectives with multiplication and addition operations respectively, and the literals with the according weights, we obtain the arithmetic circuit for $WMC(T, w)$ (Figure 2c).*



(a) Explicit enumeration

(b) Knowledge compilation

(c) Arithmetic circuit

Figure 2: From the logic theory in Example 2 to the circuit, via knowledge compilation. Evaluating the circuit in Figure 2c yields the weighted model count.

2.2 Algebraic Model Counting

A limitation of WMC is its restriction to the semiring of positive real valued numbers, for instance probabilities. Algebraic model counting (Kimmig, Van den Broeck, and De Raedt 2017) alleviates this issue by defining meaningful counting problems on (knowledge-compiled) logic formulas using an arbitrary commutative semiring.

Definition 1. *A* **commutative semiring** *is an algebraic structure $(\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$ such that 1) $\mathcal{A}$ is a set of domain elements; 2) addition $\oplus$ and multiplication $\otimes$ are binary operations $\mathcal{A} \times \mathcal{A} \to \mathcal{A}$ that are both associative and commutative; 3) $\otimes$ distributes over $\oplus$; 4) $e^{\oplus} \in \mathcal{A}$ is the neutral element of $\oplus$; 5) $e^{\otimes} \in \mathcal{A}$ is the neutral element of $\otimes$; and 6) $e^{\oplus}$ is an annihilator for $\otimes$.*

Definition 2 (Algebraic Model Counting)**.** *Given a commutative semiring $\mathcal{S} = (\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$, a propositional logic theory T, and a labelling function $\alpha : \mathcal{L} \mapsto \mathcal{A}$, mapping literals $\mathcal{L}$ of the variables in T to values from the semiring domain $\mathcal{A}$, the task of algebraic model counting (AMC) is to compute:*

$$AMC(\mathcal{S}, T, \alpha) = \bigoplus_{m \in \mathcal{M}(T)} \bigotimes_{\ell \in m} \alpha(\ell), \tag{1}$$

where $\otimes_{\ell \in m} \alpha(\ell)$ is the label of a model m represented as a set of literals ℓ true in m, and $\mathcal{M}(T)$ are the models of T.

A plethora of problems in artificial intelligence can be formulated as algebraic model counting problems (Kimmig,

Van den Broeck, and De Raedt 2017). For instance, computing gradients using the gradient semiring (Manhaeve et al. 2018), which is useful for learning tasks. Derkinderen and De Raedt (2020) adapted the AMC framework to perform decision making under uncertainty by formulating the computation of maximum expected utilities (MEUs) as algebraic model counts. First, they define an appropriate labelling function for the set of literals $\mathcal{L}$ of a logic formula:

$$\{\alpha(\ell) = (p_\ell, eu_\ell, D_\ell) \mid \ell \in \mathcal{L}\} \subset \mathcal{A}, \tag{2}$$

where p_ℓ is the probability of ℓ, eu_ℓ its expected utility, and $D_\ell = \{\ell\}$ if ℓ is a decision literal, or $D_\ell = \emptyset$ otherwise. Second, they adapt the expected utility semiring to incorporate maximisation. This results in the algebraic structure $\mathcal{S}_{meu} = (\mathcal{A}, \oplus, \otimes, e^{\oplus}, e^{\otimes})$ with

$$a_1 \oplus a_2 = \begin{cases} max(a_1, a_2) & \text{if } D_1 \neq D_2 \\ (p_1 + p_2, eu_1 + eu_2, D_1) & \text{otherwise} \end{cases}$$

$$a_1 \otimes a_2 = (p_1 \cdot p_2, p_1 \cdot eu_2 + p_2 \cdot eu_1, D_1 \cup D_2)$$

$$e^{\oplus} = (0, 0, \mathcal{D}) \quad \text{and} \quad e^{\otimes} = (1, 0, \emptyset)$$

where $a_1, a_2 \in \mathcal{A}$ and $\mathcal{D} \subset \mathcal{L}$ being the set of all decision variables and their negation. Furthermore, $max(a_1, a_2)$ returns the element (a_1 or a_2) with the highest expected utility (eu_1/p_1 or eu_2/p_2, where the divisor is a technical detail related to normalizing under the presence of constraints). Note that a semiring contains two operations, while decision making under uncertainty involves three: max, sum, and product. This considerably complicates inference but is solved in $\mathcal{S}_{meu}$ by the input-dependent $\oplus$-operation. Consequently, $\mathcal{S}_{meu}$ is not a semiring in general, and must only be used while constraining the variable ordering within the AMC computations. This is supported by several knowledge compilation tools. For more details we refer to Derkinderen and De Raedt (2020).

2.3 Representing Markov Decision Processes

While Derkinderen and De Raedt (2020) were able to solve decision making problems, their technique does not consider temporal or sequential dependencies between decisions. Such scenarios are usually modelled using MDPs (Puterman 2009). MDPs are formal models for dynamic decision problems in a fully observable and stochastic environment, with additive rewards. They consist of a set of (explicit) states $\mathbf{S}$, a set of actions $\mathbf{A}$, a reward function $R(s, a)$, and a probabilistic transition function $T(s, a, s') = P(s' \mid s, a)$ expressing the probability of ending up in state s' given the current state s and the action a performed in it.

Solving an MDP, *i.e.*, finding the best action for each state, corresponds to solving the following recursive equation for every state $s \in \mathbf{S}$

$$U(s) = \max_{a \in \mathbf{A}} \left\{ R(s, a) + \gamma \sum_{s' \in \mathbf{S}} P(s'|s, a) U(s') \right\} \tag{3}$$

This is the renowned *Bellman equation*, which can be solved using the *value iteration* algorithm (Bellman 1957). Value iteration typically starts with $U(s) = 0$, $\forall s \in \mathbf{S}$, and iteratively updates the value of U (also called the utility) according to Equation 3. A single value iteration step is referred to as a *Bellman update*. Value iteration is guaranteed to converge to the optimal solution for a discount factor $\gamma \in [0, 1)$. That is, for each state, the action yielding the maximum expected reward.

Graphically, MDPs can be represented using dynamic decision networks (DDNs) —a dynamic and decision-theoretic extension of standard BNs (cf. Figure 1). Just as in BNs, DDNs use circular nodes to represent random variables. Additionally, DDNs also contain reward nodes (diamond shaped) and decision nodes (rectangular shaped). Note that in DDNs 'actions' are called 'decisions'.

Contrary to MDPs, where states (s and s') are modelled explicitly, DDNs instead model the random variables that constitute the state. This is illustrated in Figure 1. An advantage of representing MDPs as DDNs, *i.e.*, using random variables instead of explicit states, is a (potential) exponential reduction in the representation size (Russell and Norvig 2020, Section 17.1.4). This is analogous to the space efficiency of Bayesian networks over flattened state encodings. Representing the state as a Bayesian network, as done in Figure 1 at time $t + 1$, is equivalent to using factored MDPs (Boutilier, Dearden, and Goldszmidt 2000). We will heavily exploit this structure in the next section.

3 Dynamic Decision Circuits

The main advantage of KC lies in the possibility of amortising the computationally hard compilation step over multiple circuit evaluations. Assuming that the structure of a DDN does not change over time, the idea is to compile only once the logic formula representing the transition from one time step to the next, and to reuse the resulting compiled circuit to solve the underlying DDN (*i.e.*, exploit the repeated structure). This idea is inspired by the work of Vlasselaer et al. (2016), who compile the transition function of dynamic Bayesian networks into a transition circuit. This allows them to perform efficient filtering.

Instead of applying knowledge compilation to the filtering process, however, we introduce a novel encoding to knowledge-compile the Bellman equation (cf. Equation 3). More concretely, we want to compile the structure underlying the transition in a DDN into a circuit such that evaluating it using the $\mathcal{S}_{meu}$ algebraic structure (cf. Section 2.2) corresponds to a Bellman update. Evaluating the circuit involves three operations (max, sum, and product), in contrast to only two (sum and product) in the work of Vlasselaer et al. (2016).

In order to represent a Bellman update, every term in Equation 3 should have a corresponding graphic element in its DDN. Comparing Equation 3 and the DDN in Figure 1, we can see that this is indeed the case for the terms $R(s, a)$ and $P(s' \mid s, a)$. However, there does not exist a graphical equivalent for the accumulated expected reward coming from the future ($U(s')$). Therefore, we augment the DDNs with a utility node U. In Figure 3 we perform this augmentation for the DDN in Figure 1.

3.1 Encoding

Below we describe how to encode DDNs as propositional logic formulas, assuming Boolean random variables. This

can easily be extended to multi-valued random variables using the encoding from Chavira and Darwiche (2008). Similarly, our encoding also consists of Boolean indicator variables λ and Boolean parameter variables θ.

First, we generate *indicator clauses*. Intuitively, they encode the values a variable can assume, and that it must assume exactly one of them. For non-Boolean variables Y with domain $\{y_1, \ldots, y_n\}$, we have

$$\left(\lambda_{y_1} \vee \ldots \vee \lambda_{y_n}\right) \bigwedge_{\text{for } i<j} \left(\neg\lambda_{y_i} \vee \neg\lambda_{y_j}\right) \tag{4}$$

When a variable is instead Boolean, we simply encode it with λ_x such that the truth value corresponds to that of x.

Second, we introduce *parameter clauses*. Each *parameter variable* θ corresponds to a value in the probability, reward, or utility tables of a DDN.

- Probabilities $P(x_i'|pa_1, \ldots, pa_n)$

$$\bigvee_{pa_1,\ldots,pa_n} \left(\lambda_{pa_1} \wedge \ldots \wedge \lambda_{pa_n} \wedge \theta_{x_i'|pa_1,\ldots,pa_n}\right) \leftrightarrow \lambda_{x_i'} \tag{5}$$

- Rewards $R(x_1, \ldots, x_n) = r_i$

$$\lambda_{x_1} \wedge \ldots \wedge \lambda_{x_n} \wedge \lambda_{r_i} \leftrightarrow \theta_{r_i} \tag{6}$$

- Decisions D_i

$$\lambda_{d_i} \leftrightarrow \theta_{d_i} \tag{7}$$

- Utilities $U(x_1', \ldots, x_n') = u_i$

$$\lambda_{x_1'} \wedge \ldots \wedge \lambda_{x_n'} \wedge \lambda_{u_i} \leftrightarrow \theta_{u_i} \tag{8}$$

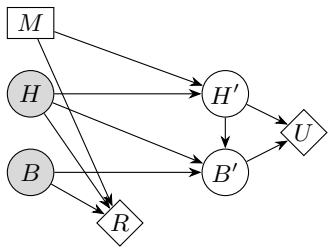
Example 4 (Encoding of the Monkey DDN)**.** *We encode every element of the DDN in Figure 3 as described above. The decision variable (M): $\lambda_m \leftrightarrow \theta_m$.*
The reward (R):

$\lambda_{r_1} \vee \lambda_{r_2} \vee \lambda_{r_3}$,
$\neg\lambda_{r_1} \vee \neg\lambda_{r_2}$, $\neg\lambda_{r_1} \vee \neg\lambda_{r_3}$, $\neg\lambda_{r_2} \vee \neg\lambda_{r_3}$,
$\lambda_h \wedge \lambda_{r_1} \leftrightarrow \theta_{r_1}$, $\lambda_b \wedge \lambda_{r_2} \leftrightarrow \theta_{r_2}$, $\lambda_m \wedge \lambda_{r_3} \leftrightarrow \theta_{r_3}$

The transition for H':

$$\begin{aligned}\lambda_{h'} \leftrightarrow &(\lambda_h \wedge \theta_{h'|h}) \vee (\neg\lambda_h \wedge \lambda_m \wedge \theta_{h'|\neg h,m}) \\ &\vee (\neg\lambda_h \wedge \neg\lambda_m \wedge \theta_{h'|\neg h,\neg m})\end{aligned}$$

Note that $\theta_{h'|h,m}$ and $\theta_{h'|h,\neg m}$ were merged together into $\theta_{h'|h}$, effectively exploiting context-specific independence.



H'	B'	$U(H', B')$
h'	b'	u_1
h'	$\neg b'$	u_2
$\neg h'$	b'	u_3
$\neg h'$	$\neg b'$	u_4

Figure 3: Dynamic decision network (DDN) for Example 1 augmented with the utility node U. For each state, which corresponds to a specific instantiation of the primed variables, we associate a *utility parameter* u_i.

The transition for B':

$$\begin{aligned}\lambda_{b'} \leftrightarrow &(\lambda_h \wedge \lambda_{h'} \wedge \theta_{b'|h,h'}) \vee (\neg\lambda_h \wedge \neg\lambda_{h'} \wedge \theta_{b'|\neg h,\neg h'}) \\ &\vee (\lambda_h \wedge \neg\lambda_{h'} \wedge \theta_{b'|h,\neg h'}) \vee (\neg\lambda_h \wedge \lambda_{h'} \wedge \theta_{b'|\neg h,h'}) \\ &\vee (\lambda_b \wedge \theta_{b'|b}) \vee (\neg\lambda_b \wedge \theta_{b'|\neg b})\end{aligned}$$

Finally, the utility variable (U):

$\lambda_{u_1} \vee \lambda_{u_2} \vee \lambda_{u_3} \vee \lambda_{u_4}$,
$\neg\lambda_{u_1} \vee \neg\lambda_{u_2}$, $\neg\lambda_{u_1} \vee \neg\lambda_{u_3}$, $\neg\lambda_{u_1} \vee \neg\lambda_{u_4}$,
$\neg\lambda_{u_2} \vee \neg\lambda_{u_3}$, $\neg\lambda_{u_2} \vee \neg\lambda_{u_4}$, $\neg\lambda_{u_3} \vee \neg\lambda_{u_4}$,
$\lambda_{h'} \wedge \lambda_{b'} \wedge \lambda_{u_1} \leftrightarrow \theta_{u_1}$,
$\lambda_{h'} \wedge \neg\lambda_{b'} \wedge \lambda_{u_2} \leftrightarrow \theta_{u_2}$,
$\neg\lambda_{h'} \wedge \lambda_{b'} \wedge \lambda_{u_3} \leftrightarrow \theta_{u_3}$,
$\neg\lambda_{h'} \wedge \neg\lambda_{b'} \wedge \lambda_{u_4} \leftrightarrow \theta_{u_4}$

3.2 Labelling

We now introduce the labelling function that maps literals to elements of $\mathcal{S}_{meu}$ (cf. Section 2.2). Concretely, we define a triple label $(p_\ell, eu_\ell, \mathcal{D}_\ell)$ for each literal ℓ, representing the probability, the expected utility, and the set of decisions, respectively. This enables us to solve DDNs using AMC.

First, all indicator variables are labelled as:

$$\alpha(\lambda_\ell) = \alpha(\neg\lambda_\ell) = (1, 0, \emptyset), \tag{9}$$

which corresponds to the neutral element $e^{\otimes}$ from $\mathcal{S}_{meu}$.

Second, we label the parameter variables.

- Probabilities with $p = P(x_i'|pa_1, \ldots, pa_n)$:

$$\alpha(\theta_{x_i'|pa_1,\ldots,pa_n}) = (p, 0, \emptyset) \tag{10}$$
$$\alpha(\neg\theta_{x_i'|pa_1,\ldots,pa_n}) = (1-p, 0, \emptyset) \tag{11}$$

- Rewards, where $R(x_1, \ldots, x_n) = r_i$:

$$\alpha(\theta_{r_i}) = (1, R(x_1, \ldots, x_n), \emptyset) \tag{12}$$
$$\alpha(\neg\theta_{r_i}) = (1, 0, \emptyset) \tag{13}$$

- Decisions:

$$\alpha(\theta_d) = (1, 0, \{d\}), \quad \alpha(\neg\theta_d) = (1, 0, \{\neg d\}) \tag{14}$$

- Utilities, where $U(x_1', \ldots, x_n') = u_i$:

$$\alpha(\theta_{u_i}) = (1, U(x_1', \ldots, x_n'), \emptyset) \tag{15}$$
$$\alpha(\neg\theta_{u_i}) = (1, 0, \emptyset) \tag{16}$$

Note that Equation 15 reflects the recursive nature of the Bellman equation. In the context of value iteration we initialise the label

$$\alpha(\theta_{u_i}) = (1, 0, \emptyset) \tag{17}$$

Example 5 (Labelling of the Monkey encoding)**.** *We provide the labelling function for the encoding in Example 4. All the indicator variables are set as specified by Equation 9.*
The decision parameter is:

$\alpha(\theta_m) = (1, 0, \{d\}), \quad \alpha(\neg\theta_m) = (1, 0, \{\neg d\})$

The reward parameters are:

$\alpha(\theta_{r_1}) = (1, -10, \emptyset), \quad \alpha(\theta_{r_2}) = (1, -4, \emptyset),$
$\alpha(\theta_{r_3}) = (1, -1, \emptyset),$
$\alpha(\neg\theta_{r_1}) = \alpha(\neg\theta_{r_2}) = \alpha(\neg\theta_{r_3}) = (1, 0, \emptyset)$

The probabilities for H':

$$\alpha(\theta_{h'|h}) = (0.2, 0, \emptyset), \quad \alpha(\neg\theta_{h'|h}) = (0.8, 0, \emptyset),$$
$$\alpha(\theta_{h'|\neg h,m}) = (0.5, 0, \emptyset), \quad \alpha(\neg\theta_{h'|\neg h,m}) = (0.5, 0, \emptyset),$$
$$\alpha(\theta_{h'|\neg h,\neg m}) = (0.8, 0, \emptyset), \quad \alpha(\neg\theta_{h'|\neg h,\neg m}) = (0.2, 0, \emptyset)$$

The labelling function for B' can be produced in a similar way. The utility parameters are initially set as specified by Equation 17.

3.3 Compiling

Given the encoding of a DDN as a propositional logic formula, and given the proper labelling function, we can proceed to compiling the circuit. To this end, we can use an off-the-shelf knowledge compiler, label the leaves in the compiled structure, and evaluate the circuit using AMC:

$$U(\mathbf{x}) = AMC(\mathcal{S}_{meu}, \Delta, \alpha_{|\mathbf{x}}), \tag{18}$$

which yields the maximum expected utility for the current time step. Here we denote the compiled circuit by Δ, and use the subscript $|\mathbf{x}$ to indicate instantiating the state variables $\mathbf{X}$ to the values $\mathbf{x}$. In the circuit this translates to changing the indicator variables' weights. For instance in the monkey example, to condition on $H = h$, we set $\alpha(\neg\lambda_h) = (0, 0, \emptyset)$.

Definition 3 (Decision Circuit, by Bhattacharjya and Shachter (2007))**.** *A decision circuit is a rooted, directed, acyclic graph whose leaf nodes are labelled with variables or constants, and whose other nodes are either summation, multiplication, or maximisation.*

Definition 4 (Dynamic Decision Circuit)**.** *A dynamic decision circuit (DDC) is a decision circuit labelled with a recursive labelling function, which associates each leaf with a value that can depend on the evaluation of the circuit itself.*

With Equation 18 we can compute the maximum expected utility only if we know the utility coming from the future, *i.e.*, $U(s')$. In the next section we show how to perform this AMC call recursively.

4 `mapl-cirup`

We explained how to obtain a (dynamic decision) circuit Δ from a DDN, and how to evaluate it —via algebraic model counting— with the labelling function α. Next, we provide a variation of the classic value iteration algorithm in which the Bellman update is substituted with an AMC call on Δ.

4.1 Bellman Update Using Circuits

Algorithm 1 shows `mapl-cirup`'s value iteration approach. Initially, the value of each state is set to zero in Line 4. Then, it iteratively performs Bellman updates until convergence (Line 5 to 11). The Bellman update itself (Equation 3) is replaced by an AMC call in Line 7, corresponding to Equation 18. Similar to classic value iteration, this update is performed for each state $s \in \mathbf{S}$, *i.e.*, for each possible instantiation $\mathbf{x}$ (Line 6). The newly computed values $U'(\mathbf{x})$ are then used in the next update step by updating the utility labels $\alpha(\theta_{\mathbf{u}})$ in Line 8. Figure 4 depicts intuitively how `mapl-cirup` differs from the classic value iteration

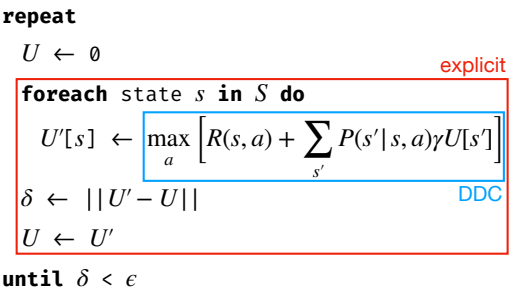


Figure 4: The classic value iteration algorithm as implemented by `mapl-cirup`.

algorithm for planning in MDPs: the loop over the states is explicit, but the Bellman update is encoded as a DDC. Importantly, the compilation of the circuit is amortised over $|\text{VI}|2^{|\mathbf{X}|}$ steps. Where, $|\text{VI}|$ is the number of iterations required to converge (to the optimal policy), and $|\mathbf{X}|$ is the number of state variables in the dynamic decision network (DDN) given in input.

Theorem 1 (Correctness of `mapl-cirup`)**.** *Algorithm 1 correctly computes the optimal solution for the problem encoded via the DDC Δ and labelling function α.*

Proof. (Sketch) The proof follows from the correctness of `mapl-cirup`'s components. Since the convergence of value iteration has already been proven, we must only prove that we correctly use the AMC framework to perform a Bellman update. We use $\mathcal{S}_{meu}$ as it was defined originally (Derkinderen and De Raedt 2020), leaving only the labelling function and the encoding to be analysed. The labelling function is a mere application of the function described along $\mathcal{S}_{meu}$. We map the DDN parameters to the appropriate labels, and update the parameters U similarly to the Bellman update (Equation 3) while the correspondence between states and utility parameters is guaranteed by the encoding. The remainder of the encoding is an adaptation of the encoding by Chavira and Darwiche (2008). □

Algorithm 1: Value Iteration with DDCs

1: **inputs:** the DDC Δ , the labelling function α, the convergence error to terminate ϵ
2: **local variables:** U, U', vectors of utilities for states $\mathbf{x}$; δ the infinite norm of the change in the utilities
3: **procedure** `mapl-cirup`$(\Delta, \alpha, \epsilon)$
4: $\quad U \leftarrow \mathbf{0}$
5: $\quad$ **repeat**
6: $\quad\quad$ **for each** instantiation $\mathbf{x}$ **do**
7: $\quad\quad\quad U'(\mathbf{x}) \leftarrow AMC(\mathcal{S}_{meu}, \Delta, \alpha_{|\mathbf{x}})$
8: $\quad\quad \alpha(\theta_{\mathbf{u}}) \leftarrow U'$
9: $\quad\quad \delta \leftarrow ||U' - U||$
10: $\quad\quad U \leftarrow U'$
11: $\quad$ **until** $\delta < \epsilon$
12: $\quad$ **return** U

A computational drawback of `mapl-cirup` is the number of utility parameters U. Namely, we introduce one such parameter for each state, to store the state value that is used during the next update step (Figure 3, Section 3). This negatively affects the number of variables involved in the compiled representation Δ, as well as its size. Moreover, this encoding forces `mapl-cirup` to iterate over all the explicit states. Investigating a more compact representation for the utility function, for example similarly to Hoey et al. (1999), is therefore a promising direction for future work that would solve both problems.

A key benefit of `mapl-cirup` is that Δ is only compiled once, effectively exploiting the repeating temporal structure and amortizing the compilation cost over multiple iterations. Indeed, Δ remains constant throughout the whole value iteration process. This also means that it is useful to dedicate more computational resources to achieving a more compact representation Δ before starting the value iteration process.

4.2 Learning

As `mapl-cirup` follows the compile+evaluate paradigm, it provides access to the plethora of techniques developed within the algebraic model counting framework. Therefore we can re-use the circuit Δ to learn for example reward parameters in the following learning task. We are given a data set $\mathbf{E}$ of trajectories $\tau = \langle s_0, a_{0:k}, r_{0:k} \rangle$, each composed of the initial known state s_0, the $k+1$ consecutive actions $a_{0:k}$ taken from that state, and the rewards $r_{0:k}$ obtained after each of those actions. Additionally, we are given the corresponding DDN where each variable may have an unknown reward parameter, i.e., the reward is an additive function where (unknown) rewards are associated with state variables. The task consists of learning those unknown reward parameters, while the intermediate states are unobserved.

We tackle this task using a gradient-based approach, exploiting the algebraic framework to compute gradients on top of Δ. To this aim, we introduce a mean squared error loss function:

$$\frac{1}{|\mathbf{E}|} \sum_{\tau \in \mathbf{E}} \sum_{t=0}^{k} \left(ceu_{t;\theta}(s_0, a_{0:t}) - r_t \right)^2 \tag{19}$$

$$ceu_{t;\theta}(s_0, a_{0:t}) = \sum_{s_t} P(s_t | s_0, a_{0:t}) R_\theta(s_t, a_t) \tag{20}$$

where we parametrise R_θ with learnable parameters; and use $ceu_{t;\theta}(s_0, a_{0:t})$ as the expected utility at time t, given the current parameters θ, the initial state and actions leading up to t. By using this loss function, we minimise the difference between the expected utility ceu and the actually observed reward at time t, r_t. This loss function is readily computable from the already compiled Δ. Additionally, probability parameters can be simultaneously learned by integrating the work of Gutmann et al. (2008) on top of our method. To do that, we must account for the rewards and decisions, and therefore use the expected utility semiring on top of DDCs.

5 Related Work

Similar to `mapl-cirup`, SPUDD (Hoey et al. 1999) is a variation of the classic value iteration algorithm using knowledge compilation. It performs the Bellman update symbolically by replacing its elements with algebraic decision diagrams (ADDs) (Bahar et al. 1997). These ADDs exploit shared values to form compact representations, and support multiplication and addition between each other. During its value iteration process, SPUDD performs multiple compilation operations. `mapl-cirup`, on the other hand, compiles only once and reuses the diagram Δ multiple times. Moreover, we perform parameter learning, which is not tackled by SPUDD, on top of the same Δ.

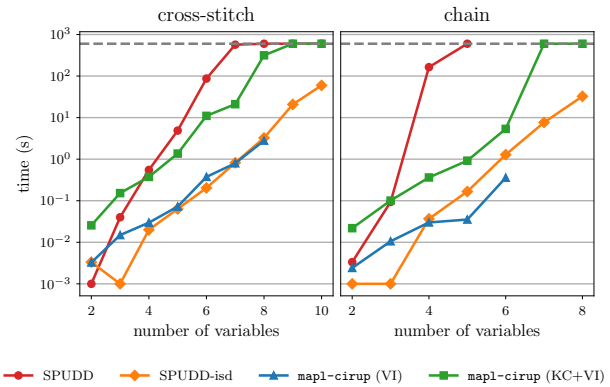


Figure 5: Comparison between `mapl-cirup` and SPUDD. It reports the value iteration time (VI) and the total time (KC+VI) including the compilation. SPUDD does not distinguish them because it interleaves compilation and VI.

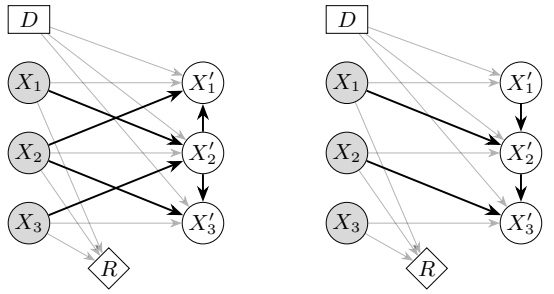
Although SPUDD was introduced over two decades ago, it is still considered the state-of-the-art approach for solving factored MDPs *exactly*, as mentioned in multiple recent publications on approximate methods (Hayes et al. 2021; Heß, Sundermann, and Thüm 2021; Moreira et al. 2021; Dudek, Shrotri, and Vardi 2022; Tan and Nejat 2022).

Vlasselaer et al. (2016) investigate how dynamic Bayesian networks (Murphy 2002) benefit from knowledge compilation techniques. However, they do neither investigate the decision-theoretic setting, nor perform learning. In an orthogonal way, Derkinderen and De Raedt (2020) use algebraic model counting with circuits in order to compute expected utilities and optimise decisions. Their work is restricted to non-temporal problems, while we consider a setting with decisions spanning over time.

Finally, *recurrent sum-product-max networks* (Tatavarti, Doshi, and Hayes 2021) have recently been introduced. These are circuits whose structure and parameters are learned directly from data and not compiled from a model. They learn from fully-observed trajectories where the total accumulated reward is provided as a signal, making it a different learning task than ours. In addition, to the best of our understanding, their approach does not perform an exact Bellman update. They update the utility value per-variable, instead of considering all the explicit states, making it more similar to a linear approximation (Guestrin et al. 2003).

		SPUDD		mapl-cirup		
model	$\lvert\mathbf{X}\rvert$	$\lvert\Delta\rvert$	VI [s]	$\lvert\Delta\rvert$	KC [s]	VI [s]
monkey	2	11 664	< 0.01	163	0.01	0.005
elevator	4	5 794	< 0.01	277	0.02	0.003
coffee	6	142 519	0.03	2542	0.6	0.054
factory	7	38 163	0.01	2932	0.93	0.105

Table 1: Comparison between mapl-cirup and SPUDD on the circuit size ($|\Delta|$), *i.e.*, the total number of nodes, the compilation time (KC), the time to find the best solution (VI). SPUDD reports time with a precision of 0.01s. It does not provide the knowledge compilation time.



(a) Cross-stitch (b) Chain

Figure 6: Dynamic decision networks with structure (in bold) and $n = 3$ variables.

6 Experiments

We performed our experimental evaluation with an Intel CPU E3-1225v3 @3.20 GHz and 32 GB of memory. All experiments ran 10 times and we report the average run time. We omit the variance when negligible. The hyperparameters for mapl-cirup were as follows: discount factor $\gamma = 0.9$ and tolerance $\epsilon = 0.1$. As a timeout, we use 600s of total run time (indicated by a dashed line on the figures). On the implementation side we used the PySDD package (Darwiche et al. 2018). In order to mitigate some of the performance issues with Python3 we JIT-compiled the circuit for the Bellman update using Numba (Lam, Pitrou, and Seibert 2015). As a comparison we include SPUDD (version 3.6.2, written in C++), with the same hyperparameters. [1]

(Q1) How does mapl-cirup perform in general? To address this question, we evaluate mapl-cirup on three MDP instances of different sizes from the SPUDD repository: elevator, coffee, and factory. We additionally include the monkey instance of Example 1, and design two DDN families that are parametric in the number of state variables $|\mathbf{X}|$ as to better illustrate the scalability. The first family has a cross-stitch-like structure, while the second forms a chain of dependencies. These are depicted in Figure 6. We devised them also to investigate how the structure imposed by intra-state dependencies can be exploited by mapl-cirup and SPUDD. In particular, the chain-like structure represents an extreme case, because the cascade of dependencies leads to an exponential blow-up (see Appendix A). On the other hand, the cross-stitch-like structure represents a more realistic scenario, where only some of the state variables depend on others. As a baseline and to verify correctness, we compare to the SPUDD algorithm. For the parametric instances we additionally include SPUDD-isd (Boutilier, Dearden, and Goldszmidt 2000), which allows SPUDD to more natively and compactly encode those instances (cf. Appendix, Section A). The results reported in Table 1 and Figure 5 indicate that mapl-cirup is able to solve dynamic decisions problems up to reasonable sizes. Both mapl-cirup and SPUDD suffer from the exponential explosion inherent to the hardness of decision making problems. Due to the utility function representation, this impacts mapl-cirup more than SPUDD. mapl-cirup produces a much smaller circuit ($|\Delta|$) than SPUDD since it only knowledge-compiles a single time step and reuses the same circuit Δ over time. In contrast, SPUDD manipulates the circuits, compiling new ones at each iteration resulting in a larger total node count.

[1] https://github.com/ML-KULeuven/mapl-cirup

	cross-stitch			chain		
$\lvert\mathbf{X}\rvert$	$\lvert\Delta\rvert$	$\lvert$VI$\rvert$	$\lvert\Delta_{\approx}\rvert$	$\lvert\Delta\rvert$	$\lvert$VI$\rvert$	$\lvert\Delta_{\approx}\rvert$
2	327	20	352	259	17	205
3	815	45	554	632	39	461
4	2 220	21	1 244	1 851	37	648
5	4 230	37	1 277	3 872	19	756
6	12 606	52	1 738	11 005	51	999
7	17 365	49	2 364	–	–	1 265
8	39 623	53	2 753	–	–	2 223
9	–	–	3 242	–	–	1 481
10	–	–	2 989	–	–	1 816

Table 2: Comparison of mapl-cirup's circuit size, with ($|\Delta_{\approx}|$) and without ($|\Delta|$) the linearly approximated utility function. Included is the number of iterations until convergence ($|$VI$|$), and the number of variables ($|\mathbf{X}|$).

(Q2) How does the exponential representation of the utility function influence performance? The main drawback of mapl-cirup is how it currently represents the utility function explicitly inside the circuit itself (cf. Section 4.1). Therefore, we explored the effect of removing the exponential encoding of the utility function by means of a well-known linear approximation (Guestrin et al. 2003), which reduces the number of utility nodes in the circuit from $2^{|\mathbf{X}|}$ to $|\mathbf{X}|$ (cf. Appendix, Section A). We report the circuit sizes $|\Delta_{\approx}|$ in Table 2 and the run times in Figure 7. These results show that indeed the utility function encoding impacts the size of Δ and thereby the run time. It also demonstrates that improvements are possible, and that these can be compatible with our compilation approach that exploits structure and enables differentiable learning (cf. Q3).

(Q3) Is the loss function a good indicator for learning the rewards? We consider two evaluation metrics: the average relative error on i) the reward parameters, and ii) the reward of each state. The latter metric is important for

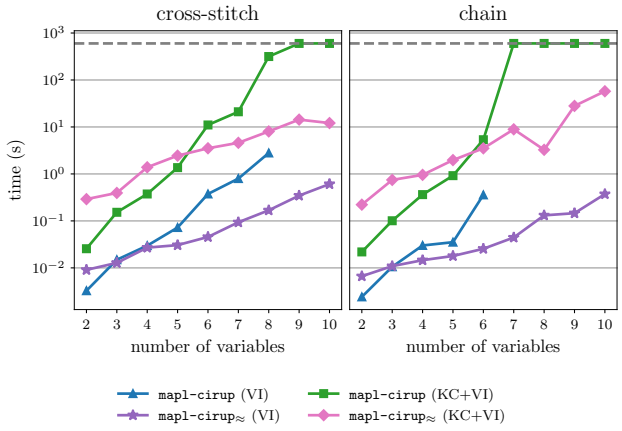


Figure 7: Comparison with and without a linearly approximated utility function. We report the value iteration time (VI) and the total time (KC+VI) including the compilation.

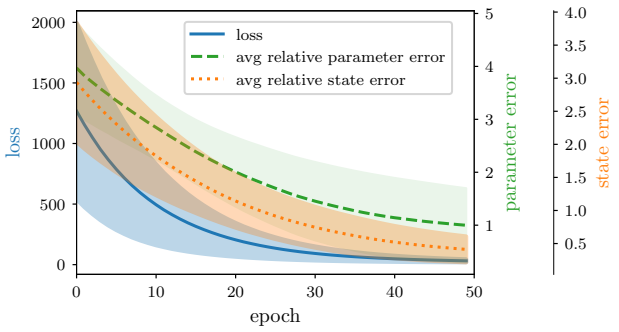


Figure 8: Results of learning reward parameters on the `coffee` example. It plots the average of each metric over 10 runs (line) and the standard deviations (coloured region).

planning, because the policy is computed on the total reward of each state. We use the previously outlined learning method, using the expected utility semiring to compute the loss function, and TensorFlow's automatic differentiation abilities (Abadi et al. 2015) to extract the gradients. The Adam optimiser (Kingma and Ba 2015) was used with learning rate $\alpha = 0.1$, $\hat{\epsilon} = 10^{-7}$, and the rest of the parameters set as default. Moreover, we initialised the reward parameters with values sampled uniformly from the interval of integers $[-30, 30]$. The dataset contains 100 trajectories ($|\mathbf{E}| = 100$) each of length 5 ($k = 5$). The training was performed on batches of size 10. We demonstrate the ability of the method by focusing on the `coffee` example, with additional reward parameters. To generate the dataset we sampled the `coffee` example enriched with extra reward parameters to make it more challenging, and initialised them with values sampled uniformly from the interval of integers $[1, 10]$. When learning, we assume that we do not know the distribution of the reward parameters. We ran the learning method 10 times, reporting the average and standard deviation in Figure 8. Two extra experiments are included in the Appendix (cf. Section B). These results show the parameter and state error decreasing along with the loss, affirmatively answering Q3 that the loss function acts as an approximate signal for learning the rewards. Specifically, we significantly decrease the relative state error from 2.94 to 0.41 on average. The relative parameter error is slightly larger, which is possible due to the additive nature of the reward function, and the additional freedom this yields to represent the state rewards.

7 Conclusions and Future Work

We introduced dynamic decision circuits, together with a value-iteration-based algorithm called `mapl-cirup` that reduces Markov planning to inference in DDCs. Thanks to the compile+evaluate paradigm, the compiled diagram can be used within the algebraic model counting framework to solve other tasks. We specifically identified and solved a learning task, where the reward parameters are learned from trajectories, in an offline reinforcement learning fashion. Our approach goes beyond planning for factored representations. It is the starting point to integrate exact inference methods with new approximate approaches such as policy gradient reinforcement learning.

As future work we consider investigating more efficient representations for the utility function, *e.g.*, by integrating the ADD operations used within SPUDD, since it alleviates the computational cost while keeping the method exact. Orthogonal to that, approximation methods (St-Aubin, Hoey, and Boutilier 2000) can be combined with `mapl-cirup`, leading again to improved run times and the possibility to scale to larger domains. In addition to the explored learning setting, exploiting the compiled, differentiable representation for other tasks, such as sensitivity analysis (Bhattacharjya and Shachter 2010), may be of interest.

Appendix

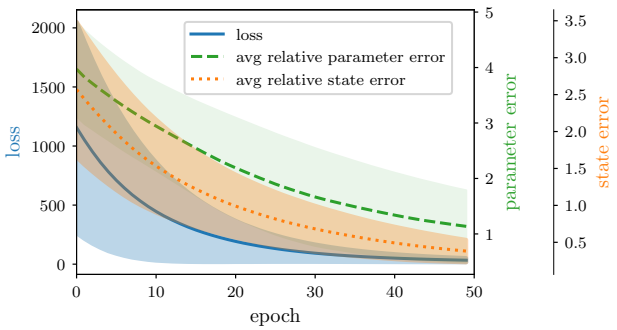
A Inference

Intra-state dependencies —also called 'correlated action effects' or 'synchronic constraints' (Boutilier, Dearden, and Goldszmidt 2000)— are dependencies between state variables within the same time slice. An example of this is given by the DDN in Figure 1. The probability distribution $P(B'|H, H')$ contains an intra-state dependency where a variable such as B' depends not only on variables from the previous time step, but also on variables from the current time step, *e.g.*, H'.

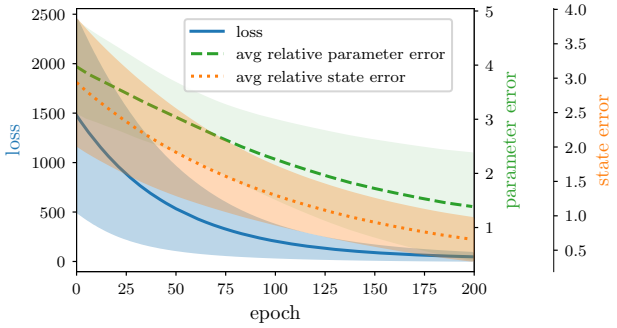
To encode the DDN of Figure 1 in a symbolic MDP solver that can not manage intra-state dependencies, they must first be removed by adding extra variables (Guestrin et al. 2003). This means splitting B' into $B'_{h'}$ and $B'_{\neg h'}$, where

$$P(B'_{h'}|H, M) = P(B'|H, H' = h) \text{ and}$$
$$P(B'_{\neg h'}|H, M) = P(B'|H, H' = \neg h),$$

for all instantiations of M. Flattening out a DDN with a rich intra-state structure may lead to an exponential blow-up, as we exemplify in Figure 10.



(a) Highly stochastic transition function



(b) Small dataset (10 trajectories)

Figure 9: Results of learning reward parameters on the `coffee` example. It plots the average of each metric over 10 runs (line) and the standard deviations (coloured region).

`mapl-cirup` does not suffer from this ailment as it leverages knowledge compilation to exploit exactly these complex dependencies. SPUDD, on the other hand, as it was introduced by Hoey et al. (1999), can not exploit these dependencies. In order to handle them, it requires a non-trivial extension, as described by Boutilier, Dearden, and Goldszmidt (2000), which is implemented in the official release of SPUDD. We therefore distinguish between SPUDD and SPUDD-isd, even though it is a single implementation. Where, the former corresponds to the original algorithm, and it requires the flattening described above to handle intra-state dependencies, while the latter includes the extension that can manage them.

In Q2 of the experimental section, the linear approximation we used is described by Guestrin et al. (2003) in the following way.

Definition 5. *A linear value function over a set of basis functions $H = \{h_1, \ldots, h_k\}$ is a function V that can be written as $V(\mathbf{x}) = \sum_{j=1}^{k} w_j h_j(\mathbf{x})$ for some weights $\mathbf{w} = (w_1, \ldots, w_k)'$.*

Normally these weights are learned, but we randomly initialised them, without focusing on the accuracy of the approximation, because it was out of the scope of our work. However, since in this way the approximation breaks the convergence property, we set an horizon limit of 50 iterations ($\sim$ the maximum number of iterations required by the exact algorithm to convergence) during our experiments.

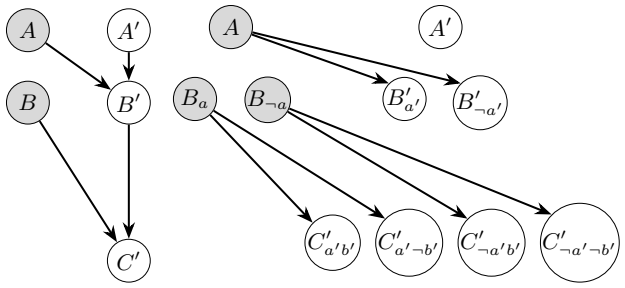


Figure 10: Intra-state chain structure (left) leading to an exponential explosion in its MDP formulation (right). Decisions, rewards, and some transitions are omitted for clarity.

B Learning

In order to further evaluate the capabilities of our learning task we executed two extra experiments. First, we changed the transition function of the `coffee` example to be less deterministic. The results are reported in Figure 9a. The quality of the learned parameters is only slightly influenced, but a higher variance is also noticeable. This is expected since we do not observe the intermediate states of a trajectory. Second, we perform the training on a smaller dataset with 10 trajectories instead of 100 (still of length $k = 5$), and batches of size 5 (instead of 10). The results are reported in Figure 9b. Interestingly, despite the fact that, of course, more epochs are required, and more variance is associated with the relative parameters error, even with such a small amount of data the learned parameters have a good quality, close to the ones learned from 100 examples.

Acknowledgments

This work was supported by the KU Leuven Research Fund (C14/18/062), the Research Foundation-Flanders (FWO, 1SA5520N), the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme, the EU H2020 ICT48 project "TAILOR" under contract #952215, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G. S.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Goodfellow, I.; Harp, A.; Irving, G.; Isard, M.; Jia, Y.; Jozefowicz, R.; Kaiser, L.; Kudlur, M.; Levenberg, J.; Mané, D.; Monga, R.; Moore, S.; Murray, D.; Olah, C.; Schuster, M.; Shlens, J.; Steiner, B.; Sutskever, I.; Talwar, K.; Tucker, P.; Vanhoucke, V.; Vasudevan, V.; Viégas, F.; Vinyals, O.; Warden, P.; Wattenberg, M.; Wicke, M.; Yu, Y.; and Zheng, X. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. Software available from tensorflow.org.

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1997. Algebraic Decision Diagrams and Their Applications. *Formal Methods Syst. Des.*, 10(2/3): 171–206.

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

Bhattacharjya, D.; and Shachter, R. D. 2007. Evaluating influence diagrams with decision circuits. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, 9–16.

Bhattacharjya, D.; and Shachter, R. D. 2010. Three new sensitivity analysis methods for influence diagrams. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence*, 56–64.

Boutilier, C.; Dearden, R.; and Goldszmidt, M. 2000. Stochastic dynamic programming with factored representations. *Artificial intelligence*, 121(1-2): 49–107.

Chavira, M.; and Darwiche, A. 2008. On Probabilistic Inference by Weighted Model Counting. *Artificial Intelligence*, 172(6): 772–799.

Darwiche, A. 2002. A Logical Approach to Factoring Belief Networks. In *Proceedings of the Eights International Conference on Principles of Knowledge Representation and Reasoning*, 409–420.

Darwiche, A. 2009. *Modeling and reasoning with Bayesian networks*. Cambridge university press.

Darwiche, A.; Marquis, P.; Suciu, D.; and Szeider, S. 2018. Recent trends in knowledge compilation (dagstuhl seminar 17381). In *Dagstuhl Reports*, volume 7.

Dean, T.; and Kanazawa, K. 1989. A Model for Reasoning about Persistence and Causation. *Computational Intelligence*, 5(2): 142–150.

Derkinderen, V.; and De Raedt, L. 2020. Algebraic circuits for decision theoretic inference and learning. In *ECAI 2020*, volume 325. IOS Press.

Dudek, J. M.; Shrotri, A. A.; and Vardi, M. Y. 2022. DPSampler: Exact Weighted Sampling Using Dynamic Programming. In *Proc. 31st IJCAI*.

Guestrin, C.; Koller, D.; Parr, R.; and Venkataraman, S. 2003. Efficient Solution Algorithms for Factored MDPs. *Journal of Artificial Intelligence Research*, 19: 399–468.

Gutmann, B.; Kimmig, A.; Kersting, K.; and De Raedt, L. 2008. Parameter Learning in Probabilistic Databases: A Least Squares Approach. In *Machine Learning and Knowledge Discovery in Databases. ECML/PKDD. Lecture Notes in Computer Science.*, volume 5211, 473–488. Springer.

Hayes, L.; Doshi, P.; Pawar, S.; and Tatavarti, H. T. 2021. State-Based Recurrent SPMNs for Decision-Theoretic Planning under Partial Observability. In *Proc. 30th IJCAI*.

Heß, T.; Sundermann, C.; and Thüm, T. 2021. On the scalability of building binary decision diagrams for current feature models. In *SPLC (A)*.

Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUDD: Stochastic Planning Using Decision Diagrams. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, 279–288.

Howard, R. A.; and Matheson, J. E. 1984. Influence Diagrams. *Readings on Principles and Applications of Decision Analysis*, 721–762.

Kanazawa, K.; and Dean, T. 1989. A Model for Projection and Action. In *Proc. 11th IJCAI*, 985–990.

Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2017. Algebraic Model Counting. *Journal of Applied Logic*, 22: 46–62.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR*.

Koller, D.; and Friedman, N. 2009. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press.

Lam, S. K.; Pitrou, A.; and Seibert, S. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 1–6.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2018. DeepProbLog: Neural probabilistic logic programming. *NeurIPS*, 31.

Moreira, D. A. M.; Delgado, K. V.; de Barros, L. N.; and Mauá, D. D. 2021. Efficient algorithms for Risk-Sensitive Markov Decision Processes with limited budget. *Int. J. Approx. Reason.*, 139.

Murphy, K. P. 2002. *Dynamic Bayesian Networks: Representation, Inference and Learning*. Ph.D. thesis, University of California, Berkeley.

Puterman, M. L. 2009. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley.

Roth, D. 1996. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2): 273–302.

Russell, S.; and Norvig, P. 2020. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition edition.

St-Aubin, R.; Hoey, J.; and Boutilier, C. 2000. APRICODD: Approximate Policy Construction Using Decision Diagrams. In *NeurIPS*, volume 13. MIT Press.

Tan, A. H.; and Nejat, G. 2022. Enhancing Robot Task Completion Through Environment and Task Inference: A Survey from the Mobile Robot Perspective. *J. Intell. Robotic Syst.*, 106(4).

Tatavarti, H.; Doshi, P.; and Hayes, L. 2021. Data-Driven Decision-Theoretic Planning Using Recurrent Sum-Product-Max Networks. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31.

Valiant, L. G. 1979. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3).

Vlasselaer, J.; Meert, W.; Van den Broeck, G.; and De Raedt, L. 2016. Exploiting Local and Repeated Structure in Dynamic Bayesian Networks. *Artificial Intelligence*, 232.