Exploiting Local and Repeated Structure in Dynamic Bayesian Networks

Jonas Vlasselaer, Wannes Meert, Guy Van den Broeck, Luc De Raedt

Departement of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A - bus 2402, 3001 Heverlee, Belgium
{firstname.lastname}@cs.kuleuven.be

Abstract

We introduce the *structural interface algorithm* for exact probabilistic inference in dynamic Bayesian networks. It unifies state-of-the-art techniques for inference in static and dynamic networks, by combining principles of *knowledge compilation* with the *interface algorithm*. The resulting algorithm not only exploits the repeated structure in the network, but also the local structure, including determinism, parameter equality and context-specific independence. Empirically, we show that the structural interface algorithm speeds up inference in the presence of local structure, and scales to larger and more complex networks.

Keywords: Probabilistic Graphical Models, Dynamic Bayesian Networks, Probabilistic Inference, Knowledge Compilation

1. Introduction

Bayesian Networks (BNs) are powerful and popular tools for reasoning about uncertainty [1]. Although BNs where originally developed for static domains, they have been extended towards dynamic domains to cope with time-related or sequential data [2, 3]. These Dynamic Bayesian Networks (DBNs) generalize hidden Markov models and Kalman filters, and are widely used in applications such as speech recognition, bio-sequence analysis, health monitoring, machine monitoring, robotics and games.

Inference methods for *static* BNs, including junction trees and variable elimination, exploit *conditional independencies* (CI) by using a factorized representation of the probability distribution. More recent techniques, including knowledge compilation [4], also exploit *local structure* (LS) in the network. This type of structure can induce additional independencies, and is present when the conditional probability tables contain deterministic dependencies or equal parameters. It is well-known that, in the presence of LS, knowledge compilation often outperforms traditional methods [4].

Inference in *dynamic* models can be performed by unrolling the network and using static inference techniques on the resulting network. This approach,

Approach	CI	LS	RS
1. Traditional BN algorithm on the unrolled network	✓		
2. Knowledge compilation on the unrolled network	✓	✓	
3. Interface algorithm	✓		✓
4. Structural interface algorithm	✓	✓	✓

Table 1: Properties exploited by DBN inference algorithms.

however, performs poorly when the number of time steps increases. Therefore, special purpose algorithms have been devised, such as the *interface algorithm* [3], which extends the forward-backward algorithm for hidden Markov models towards general DBNs. In addition to CI, these algorithms exploit the *repeated structure* (RS) obtained from duplicating the network along the time dimension.

The key contribution of the present paper is that we show how to use *knowledge compilation* techniques for efficient exact inference in DBNs. The resulting *structural interface algorithm* speeds up inference by exploiting CI, RS as well as LS (see Table 1). We investigate the trade-offs of compiling the complex transition model of a DBN into a circuit representation. We evaluate our algorithm on three classes of benchmark DBNs, and show that the structural interface algorithm outperforms the classical interface algorithm in the presence of LS. As a result, we can tackle dynamic models that are considerably more complex than what is currently possible with exact solvers.

The paper is organized as follows. Sections 2 and 3 provide the necessary background, on inference for dynamic networks, and static networks with local structure. Next, in Section 4 we describe the structural interface algorithm. Finally, Section 5 compares the different DBN inference techniques empirically.

2. Inference in Dynamic Bayesian Networks

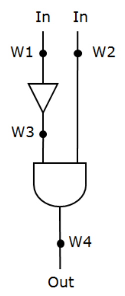
We first review Dynamic Bayesian Networks and existing work on exact probabilistic inference with these representations. Upper-case letters (Y) denote random variables and lower case letters (y) denote their instantiations. Bold letters represent sets of variables ($\mathbf{Y}$) and their instantiations ($\mathbf{y}$).

2.1. Representation and Tasks

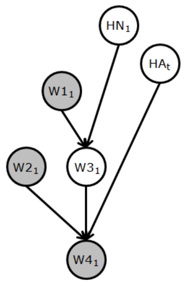
A Dynamic Bayesian Network (DBN) [2, 3] is a directed acyclic graphical model that represents a stochastic process. It models a probability distribution over a semi-infinite collection of random variables $\mathbf{Z}_1, \mathbf{Z}_2, \mathbf{Z}_3, \ldots$, where $\mathbf{Z}_t$ are the variables at time t and $\mathbf{Z}_{1:T}$ denotes all variables up until time T. A Dynamic Bayesian Network is defined by two networks: B_1, which specifies the prior or initial state distribution $\Pr(\mathbf{Z}_1)$, and $B_{\rightarrow}$, a two-slice temporal BN (2TBN) that specifies the transition model $\Pr(\mathbf{Z}_t|\mathbf{Z}_{t-1})$. Together, they represent the distribution

$$\Pr(\mathbf{Z}_{1:T}) = \Pr(\mathbf{Z}_1) \prod_{t=2}^{T} \Pr(\mathbf{Z}_t|\mathbf{Z}_{t-1}) \tag{1}$$

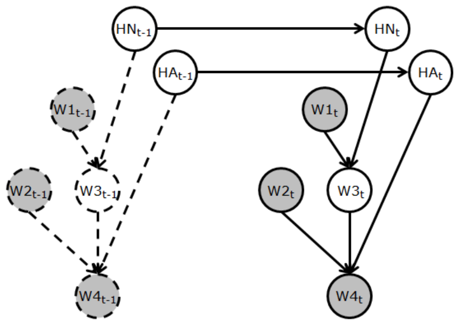
The initial network B_1 is a regular Bayesian network, which factorizes the distribution over its N variables as $\Pr(\mathbf{Z}_1) = \prod_{i=1}^{N} \Pr(Z_1^i | \mathbf{Pa}(Z_1^i))$, where Z_t^i is the ith variable at time t and $\mathbf{Pa}(Z_t^i)$ are the parents of Z_t^i in the network. The transition model $B_{\rightarrow}$ is not a regular Bayesian network as only the nodes in the second slice (for time t) of the 2TBN[1] have an associated conditional probability distribution. Thus, the transition model factorizes as $\Pr(\mathbf{Z}_t | \mathbf{Z}_{t-1}) = \prod_{i=1}^{N} \Pr(Z_t^i | \mathbf{Pa}(Z_t^i))$, where $\mathbf{Pa}(Z_t^i)$ can contain variables from either $\mathbf{Z}_t$ or $\mathbf{Z}_{t-1}$.



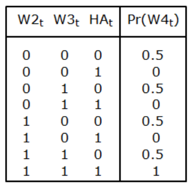
(a) An electronic digital circuit.



(b) The prior distribution.



(c) The transition model (2TBN).



W2t	W3t	HAt	Pr(W4t)
0	0	0	0.5
0	0	1	0
0	1	0	0.5
0	1	1	0
1	0	0	0.5
1	0	1	0
1	1	0	0.5
1	1	1	1

(d) The CPT for W4$_t$.

Figure 1: A digital circuit containing a logical *NOT-gate* (with *wire 1* as input and *wire 3* as output) and a logical *AND-gate* (with *wire 2* and *wire 3* as inputs and *wire 4* as output) and its corresponding DBN (shaded nodes are observed). The 1.5TBN is obtained by removing all dashed arcs and nodes from the 2TBN.

We use as running example the task of finding failing components in digital electronic circuits (see Figure 1a). This problem can be easily modeled as a DBN, where each of the variables in $\mathbf{Z}_t$ either represents the state of a wire (e.g. *high* or *low*) or the state of a logical component (e.g. *healthy* or *faulty*) (see Figure 1b) [5]. The transition model (Figure 1c) defines the dynamics of the components' state over time.

[1]We focus on first-order Markov chains where the transition model is specified by a 2TBN. However, our results generalize to kTBNs and $(k-1)^{\text{th}}$-order Markov processes.

The goal of (marginal) inference in a DBN is to compute $\Pr(X_t^i|\mathbf{e}_{1:\tau})$, the probability of a hidden variable X^i at time t, given a sequence of observations $\mathbf{e}_{1:\tau}$ up until time τ. If $(t = \tau)$ this is called filtering, if $(t > \tau)$ prediction, and if $(t < \tau)$ smoothing. For our example, one is typically interested in the health states HA_t (for the AND-gate) and HN_t (for the NOT-gate) at time t, given a sequence of observed electrical inputs and outputs up to and including t, i.e. the task of filtering. This corresponds to computing $\Pr(HA_t|\mathbf{w1}_{1:t}, \mathbf{w2}_{1:t}, \mathbf{w4}_{1:t})$ and $\Pr(HN_t|\mathbf{w1}_{1:t}, \mathbf{w2}_{1:t}, \mathbf{w4}_{1:t})$.

2.2. Unrolling the Network

The semantics of a DBN, for a finite number of time steps T, is defined by unrolling the transition model (2TBN) for all time slices (Equation 1). Such an unrolled network (see Figure 2) is equivalent to a static Bayesian network and allows one to perform inference with any standard algorithm for BNs [6, 5].

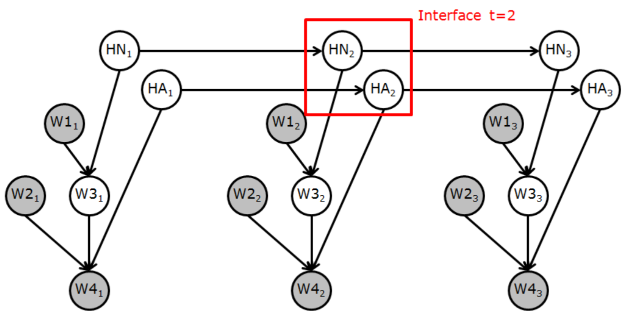


Figure 2: The unrolled network for three time slices.

Despite the wide range of existing algorithms for BNs, naively unrolling the network for T time slices has multiple drawbacks: (1) the time complexity of inference depends on heuristics and is not guaranteed to scale linearly with T, (2) it requires $O(T)$ memory, and (3) the number T is often unknown which implies that adding a new time step requires inference in the complete network. While a standard BN algorithm in combination with a sensible heuristic allows one to overcome (1), e.g. heuristics based on a "slice-by-slice" ordering [7], more specific algorithms are required to overcome (2) and (3).

2.3. Exploiting Repeated Structure

Explicitly unrolling the 2TBN introduces a repeated structure in the network. This structure is exploited by specific inference algorithms for DBNs to overcome all the above limitations of unrolling.

A key property of DBNs is that the hidden variables $\mathbf{X}_t$ d-separate the past from the future, that is, knowing their values makes the future independent of the past. Often, a subset $\mathbf{I}_t$ of $\mathbf{X}_t$ also suffices to d-separate the past from the future. This set $\mathbf{I}_t$, referred to as the *interface*[2] [3], consists of the nodes from time slice t that have an outgoing arc to nodes in time slice $t+1$ (see Figure 2). The interface allows one to define the transition model by means of a 1.5TBN rather than a 2TBN. This 1.5TBN is obtained by removing all non-interface variables and all arcs in the first time slice of the 2TBN (see Figure 1c) [3].

Exploiting the repeated structure in a DBN reduces the inference task in DBNs to repeatedly performing inference in the 1.5TBN. This is achieved by means of a forward (and backward) pass, similar to the forward-backward algorithm for hidden Markov models [8]. The forward pass involves computing the joint probability distributions $\Pr(\mathbf{I}_t|\mathbf{e}_{1:t})$ for every time step t. These distributions, referred to as the *forward messages*, can be computed recursively as follows [3]:

$$\Pr(\mathbf{I}_t|\mathbf{e}_{1:t}) = \sum_{\mathbf{I}_{t-1}} \Pr(\mathbf{I}_t|\mathbf{I}_{t-1}, \mathbf{e}_t) \Pr(\mathbf{I}_{t-1}|\mathbf{e}_{1:t-1}) \tag{2}$$

The factor $\Pr(\mathbf{I}_t|\mathbf{I}_{t-1}, \mathbf{e}_t)$ can be computed as $\sum_{\mathbf{Z}_t \setminus \mathbf{I}_t} \Pr(\mathbf{Z}_t|\mathbf{I}_{t-1}, \mathbf{e}_t)$ on the 1.5TBN without the need to unroll the network. The standard implementation of the *interface algorithm*[3] [3] computes this factor using *junction trees* where it is enforced that all nodes in $\mathbf{I}_{t-1}$ and in $\mathbf{I}_t$ each form a clique. Based on the forward messages, one can compute marginal probabilities as follows:

$$\Pr(Z_t^i|\mathbf{e}_{1:t}) = \sum_{\mathbf{I}_{t-1}} \Pr(Z_t^i|\mathbf{I}_{t-1}, \mathbf{e}_t) \Pr(\mathbf{I}_{t-1}|\mathbf{e}_{1:t-1})$$

Although the size of the joint distribution in the forward message grows exponentially with the size of the interface $\mathbf{I}_t$, the interface algorithm overcomes all drawbacks of naively unrolling the network. It scales linearly with the number of time slices, only needs to keep in memory the last forward message and the 1.5TBN, and it allows for a time step to be added without the need to recompute the forward messages for previous time steps.

The forward message allows one to correctly compute $\Pr(Z_t^i|\mathbf{e}_{1:\tau})$ with $t \geqslant \tau$, but not when $t < \tau$ (i.e., the smoothing task). Then, one also defines a *backward interface* to compute *backward messages*. For the sake of simplicity, we omit the backward pass, as it is similar to the forward pass [3].

Another approach, known as *constant-space* algorithms [7], extend the *variable elimination* algorithm for Bayasian networks to efficiently perform inference in dynamic networks. Concretely, these algorithms utilize "slice-by-slice" elimination orders and dynamically generate the conditional probability tables of the network. Hence, inference scales linearly with the number of time-slices T while the required memory is constant and independent of T.

For notational convenience, we omit the observations $\mathbf{e}_{1:t}$ in the remainder of this text and refer to the forward message as $\Pr(\mathbf{I}_t)$. Its different entries (possible variable instantiations) are denoted by ($\mathbf{i}_t^1$, $\mathbf{i}_t^2$, ... $\mathbf{i}_t^M$). In case all variables are binary, we have $M = 2^{|\mathbf{I}|}$.

[2]One distinguishes between forward an backward interfaces [3]. We focus on the former.
[3]In the Bayes Net Toolbox for Matlab, available at `https://github.com/bayesnet/bnt`

3. Local Structure in Static Bayesian Networks

Most inference algorithms for BNs, such as *junction trees*, only exploit conditional independences and have time complexities that are exponential in the treewidth of the BN. Algorithms based on *knowledge compilation*, however, are also capable of exploiting local structure, allowing one to conduct inference more efficiently. We first introduce different types of local structure and show how these can be exploited.

3.1. Local Structure

Bayesian networks often exhibit abundant local structure in the form of determinism and context-specific independence (CSI) [9]. Determinism is introduced by 0 and 1 *parameters* in the network while CSI is often the result of *equal parameters*. Exploiting local structure can lead to exponential speed gains and allows one to perform inference in networks of high treewidth, where this is otherwise impossible [4].

The Conditional Probability Table (CPT) for *wire 4* in our running example (see Figure 1d) contains the different types of local structure. When the logical component is *healthy* ($HA_t = \top$), the 1 *parameter* indicates that $W4_t$ is deterministically true (*high*) if all wires at the input of the component are true (*high*). The 0 *parameters* indicate that $W4_t$ is deterministically false (*low*) in all other cases. When the logical component is *faulty* ($HA_t = \bot$), the *equal parameters* (0.5) give rise to context-specific independence since the state of $W4_t$ does not depend anymore on the state of the wires at the input of the component, i.e. $\Pr(W4_t | W2_t, W3_t, HA_t = \bot) = \Pr(W4_t | HA_t = \bot)$

3.2. Knowledge Compilation

Knowledge compilation is a technique capable of exploiting different types of local structure [4]. The approach we take can be summarized as performing three steps: (1) conversion of the BN into a logical knowledge base and weighted model counting problem, (2) compiling the knowledge base into a more tractable target representation, and (3) performing inference in the target representation.

3.2.1. Conversion to Weighted Model Counting

In the first step, the BN is encoded into a knowledge base (KB) (i.e., a sentence in propositional Boolean logic) whose satisfying assignments are called *models*. An associated weight function w, which maps each propositional variable to a real number, allows one to reduce the task of probabilistic inference to *weighted model counting* [10]. The weight of a model is given by the product of the weights of all variables consistent with the model. The sum of all models then corresponds to the probability of evidence in the BN. Computing the marginal probability of a variable instantiation comes down to summing and normalizing the weights of all models consistent with the instantiation.

We illustrate the conversion step on our running example by means of the encoding proposed by Fierens et al. [11]. The propositional formula for the CPT shown in Figure 1d contains one *parameter variable* ($w(P_{Faulty|\neg HA,t}) = 0.5$) and six *indicator variables* ($w(\cdot) = 1$):

$$\begin{aligned} Normal_t &\Leftrightarrow W2_t \wedge W3_t \wedge HA_t \\ Faulty_t &\Leftrightarrow \neg HA_t \wedge P_{Faulty|\neg HA,t} \\ W4_t &\Leftrightarrow Normal_t \vee Faulty_t \end{aligned}$$

The first formula encodes the last entry of the CPT, associated with the indicator variable $Normal_t$. We can safely omit the corresponding parameter variable since it represents a probability of one and does not change the weighted model count. All entries in the CPT that have an equal probability of 0.5 are compactly encoded into the second formula. With these entries we associate the indicator variable $Faulty_t$. The third formula expresses when $W4_t$ is true. All entries in the CPT with a 0 parameter can be dropped as they give rise to models with a weight of 0. A model for this formula is, for example, given by $(W4_t, \neg Normal_t, Faulty_t, \neg HA_t, P_{Faulty|\neg HA,t}, \neg W2_t, W3_t)$ which has a corresponding weight of $1 \cdot 1 \cdot 1 \cdot 1 \cdot 0.5 \cdot 1 \cdot 1 = 0.5$.

In general, the knowledge base KB for a BN can be obtained by encoding each row of each CPT as a propositional formula and conjoining these formulas. This requires an indicator variable for each value z of a random variable Z and a parameter variable for each CPT parameter $\theta_{z|\mathbf{u}}$. The encoding of Fierens et al. [11] assumes that all variables are Boolean. In the general case, any other encoding can be used. For details, we refer to Darwiche [5].

3.2.2. Compilation and Inference

Once the network is encoded, the knowledge base KB is transformed into a more tractable representation which allows for efficient marginal inference. The language often used as target representation is d-DNNF (deterministic Decomposable Negation Normal Form). It is known to support weighted model counting in polynomial time [12] and generalizes other well-known languages such as OBDD and FBDD. The procedure consists of three steps:

1. Compile the knowledge base KB into a d-DNNF Δ [13].

$$\Delta = \textsc{Compile}(KB)$$

2. Incorporate evidence $\mathbf{e}$ by setting to zero the weight of any indicator variable that is not compatible with the evidence.

$$w'(Z) = \begin{cases} w(Z) & \neg Z \notin \mathbf{e} \\ 0 & \neg Z \in \mathbf{e} \end{cases}$$

3. Traverse the obtained d-DNNF to either:
(a) compute the weighted model count, which corresponds to the probability of the evidence in the BN, with an upward pass only:

$$\Pr(\mathbf{e}) = \textsc{Eval}_{\uparrow}(\Delta, w')$$

(b) compute the marginal probability $\Pr(Z|\mathbf{e})$, for all variables Z in parallel, with one upward and downward pass [5, Algorithm 34]:

$$\Pr(Z|\mathbf{e}) = \textsc{Eval}_{\uparrow\downarrow}(\Delta, w')$$

In the literature, one often converts the obtained target representation (d-DNNF) into an Arithmetic Circuit (AC) and traverses this circuit. Since this step is not strictly necessary, we omit it and use both terms interchangeably.

Compiling a knowledge base into a d-DNNF is computationally hard but has several advantages. Firstly, the size of the obtained circuit is not necessarily exponential in the treewidth. Secondly, the circuit can be reused in the presence of new evidence to compute marginal probabilities in polytime, without the need to recompile it. Thirdly, a d-DNNF allows a set of polytime transformations of which one is *conditioning*. This transformation, denoted $(\Delta|\mathbf{v})$, replaces the variables $\mathbf{V}$ in Δ by their assignment in $\mathbf{v}$ and propagates these values while preserving the properties of the target representation [12].

4. The Structural Interface Algorithm

We propose the *structural interface algorithm* for efficient inference in DBNs. It exploits conditional independence and repeated structure in the network in a way similar to the interface algorithm [3]. The use of knowledge compilation, however, allows us to additionally exploit local structure in the transition model.

We explore several approaches of integrating the interface algorithm with knowledge compilation. They have different memory requirements and trade offs between putting the burden on the compiler, a post-compilation (conditioning) step or the inference step.Table 2 summarizes the complexity of the different steps for each of the different interface encodings we present below.

	ENC1	ENC2	ENC3	ENC4
Compilation	$\mathcal{O}(2^{\omega_{1.5\text{TBN}}+2\cdot\|\mathbf{I}\|})$	$\mathcal{O}(2^{\omega_{1.5\text{TBN}}})$	$\mathcal{O}(2^{\omega_{1.5\text{TBN}}})$	$\mathcal{O}(2^{\omega_{1.5\text{TBN}}})$
Conditioning	n\a	$2 \cdot 2^{\|\mathbf{I}\|} \cdot \mathcal{O}(\|\Delta\|)$	n\a	$2^{\|\mathbf{I}\|} \cdot \mathcal{O}(\|\Delta\|)$
Evaluation	$2 \cdot \mathcal{O}(\|\Delta\|)$	$2 \cdot \mathcal{O}(\|\Delta\|)$	$2^{2\cdot\|\mathbf{I}\|} \cdot \mathcal{O}(\|\Delta\|)$	$2^{\|\mathbf{I}\|} \cdot \mathcal{O}(\|\Delta\|)$

Table 2: Complexity of each step for the different interface encodings. Parameter ω_N represents the treewidth of network N. Circuit Δ refers to the one constructed in the previous step. For Conditioning and Evaluation, we report the asymptotic complexity of one call ($\mathcal{O}(|\Delta|)$), multiplied by the number of required calls (e.g. 2).

4.1. Exploiting Local Structure in the Transition Model

Our approach performs inference in a DBN by recursively computing the forward message (see Equation 2) but uses knowledge compilation, rather than junction trees, to compute the factor $\Pr(\mathbf{I}_t|\mathbf{I}_{t-1}, \mathbf{e}_t)$ on the 1.5TBN. This does not only involve encoding, then compiling, the 1.5TBN, but also require to represent the joint distributions $\Pr(\mathbf{I}_{t-1})$ and $\Pr(\mathbf{I}_t)$ in the compiled circuit.

The 1.5TBN is encoded by means of a knowledge base $KB_{1.5}$ (cf. Section 3.2.1). Each CPT (for variables in the second slice) is turned into a corresponding set of formulas. Now, we identify several approaches to represent $\Pr(\mathbf{I}_{t-1})$ and $\Pr(\mathbf{I}_t)$ and to compute the forward message on a circuit representation.

4.1.1. Compiling the Interface into the Circuit (ENC1)

A joint distribution $\Pr(\mathbf{I})$ can be naturally encoded into a knowledge base $KB_{\mathbf{I}}$ as discussed in Section 3.2.1. This requires $2^{|\mathbf{I}|}$ formulas and indicator variables to be added, all in one-to-one correspondence to the rows of $\Pr(\mathbf{I})$. For our running example, with variables HN_t and HA_t in the interface, $KB_{\mathbf{I}_t}$ is given by the following 4 formulas (and similar for $KB_{\mathbf{I}_{t-1}}$):

$$\begin{array}{lll}
State_t^1 \Leftrightarrow HN_t \wedge HA_t & & (\text{for } \mathbf{i}_t^1) \\
State_t^2 \Leftrightarrow HN_t \wedge \neg HA_t & & (\text{for } \mathbf{i}_t^2) \\
State_t^3 \Leftrightarrow \neg HN_t \wedge HA_t & & (\text{for } \mathbf{i}_t^3) \\
State_t^4 \Leftrightarrow \neg HN_t \wedge \neg HA_t & & (\text{for } \mathbf{i}_t^4)
\end{array}$$

This allow us to compute the forward message as follows:

$$(\Pr(\mathbf{i}_t^1), \ldots, \Pr(\mathbf{i}_t^n)) = \textsc{Eval}_{\uparrow\downarrow}(\textsc{Compile}(KB_{\mathbf{I}_{t-1}} \wedge KB_{1.5} \wedge KB_{\mathbf{I}_t}), w)$$

where w is updated with $w(State_{t-1}^j) = \Pr(\mathbf{i}_{t-1}^j)$. The **advantage** of this encoding is that, for each time step, only two passes trough the circuit are needed to compute the forward message (i.e., one call to $\textsc{Eval}_{\uparrow\downarrow}$). The **disadvantage** is that the number of required formulas to encode $\Pr(\mathbf{I})$ scales exponentially in $|\mathbf{I}|$ (i.e. the number of interface variables).

4.1.2. Conditioning the Interface into the Circuit (ENC2)

The exponential aspect of $KB_{\mathbf{I}}$ has an adverse effect on the heuristics used by general-purpose compilation tools as it not only dwarfs $KB_{1.5}$ in size, but also represents a joint distribution without any local structure. A d-DNNF that is logically equivalent with the one obtained by $\textsc{Compile}(KB_{1.5} \wedge KB_{\mathbf{I}})$ can be obtained, however, by only compiling $KB_{1.5}$ with a general-purpose tool and adding $\Pr(\mathbf{I})$ to the resulting circuit by means of conditioning. Concretely, a joint distribution over all variables in $\mathbf{I}$ can be added to a compiled circuit Δ in the following way:

$$\textsc{Addi}(\Delta, \mathbf{I}) = \bigvee_{\mathbf{i}^j \in \mathbf{I}} (\Delta | \mathbf{i}^j) \wedge State^j \wedge \mathbf{i}^j \qquad (3)$$

The result of $\textsc{Addi}(\Delta, \mathbf{I})$ is a d-DNNF which allows us to compute the forward message as follows:

$$(\Pr(\mathbf{i}_t^1), \ldots \Pr(\mathbf{i}_t^n)) = \textsc{Eval}_{\uparrow\downarrow}(\textsc{Addi}(\textsc{Addi}(\textsc{Compile}(KB_{1.5}), \mathbf{I}_{t-1}), \mathbf{I}_t), w)$$

where w is updated with $w(State_{t-1}^j) = \Pr(\mathbf{i}_{t-1}^j)$. The **advantage** of incorporating $\Pr(\mathbf{I})$ directly into the d-DNNF is that the heuristic of the compiler does not have to deal with $KB_{\mathbf{I}}$ and can focus on better compiling the much smaller and more structured sentence $KB_{1.5}$. Furthermore, this approach allows one to share identical subcircuits, leading to an efficient computation of the forward message with only two passes trough the obtained circuit. The **disadvantage** is that the number of conditioning operations scales exponentially with $|\mathbf{I}|$.

4.1.3. Introducing the Interface as Evidence (ENC3)

We can compute the forward message using only $\Delta_{1.5}$, which is obtained by $\textsc{Compile}(KB_{1.5})$, without the need to explicitly encode $\Pr(\mathbf{I}_{t-1})$ and $\Pr(\mathbf{I}_t)$. This is done by repeatedly updating the weight function to incorporate each of the combinations of instantiations of $\Pr(\mathbf{I}_{t-1})$ and $\Pr(\mathbf{I}_t)$ as evidence (see Step 2, section 3.2.2). Concretely, the probability of the j-th instantiation in the forward message can be computed in the following way:

$$\Pr(\mathbf{i}_t^j) = \sum_{k=0}^{M} \textsc{Eval}_{\uparrow}(\textsc{Compile}(KB_{1.5}), w_{k\to j}) \cdot \Pr(\mathbf{i}_{t-1}^k)$$

where $w_{k\to j}$ incorporates the instantiations $\mathbf{i}_{t-1}^k$ and $\mathbf{i}_t^j$ and $M = 2^{|\mathbf{I}|}$ in case all interface variables are binary. Note that $\textsc{Compile}(KB_{1.5})$ only needs to be performed once. The **advantage** of bypassing an explicit encoding of the interfaces is that it lowers the memory requirements as the forward message is directly computed on the circuit $\Delta_{1.5}$. The **disadvantage** is that computing the forward message requires $2^{2\cdot|\mathbf{I}|}$ passes trough the circuit. Moreover, $2^{2\cdot|\mathbf{I}|} \cdot |\Delta_{1.5}|$ will be larger than $2 \cdot |\Delta|$ (the evaluation step of the previous two encodings) because identical subcircuits are note shared.

4.1.4. Encoding for the Structural Interface Algorithm (ENC4)

The approach of compiling $KB_{\mathbf{I}_{t-1}} \wedge KB_{1.5} \wedge KB_{\mathbf{I}_t}$ (ENC1) is similar to the interface algorithm where one adds edges to the moral graph between all nodes in $\mathbf{I}_{t-1}$ and $\mathbf{I}_t$ [3]. Since the compilation step is the most complex step in the knowledge compilation pipeline, and this approach potentially has to deal with a more complex knowledge base, we do not prefer this encoding.

For the structural interface algorithm, we propose an hybrid encoding that employs ENC2 as well as ENC3. Concretely, we explicitly introduce $\Pr(\mathbf{I}_{t-1})$ by conditioning while $\Pr(\mathbf{I}_t)$ is implicitly introduced as evidence. This allows us to compute the probability of the j-th instantiation in the forward message as follows:

$$\Pr(\mathbf{i}_t^j) = \textsc{Eval}_{\uparrow}(\textsc{Addi}(\textsc{Compile}(KB_{1.5}), \mathbf{I}_{t-1}), w_{\to j}) \tag{4}$$

where $w_{\to j}$ is updated with $w(\mathit{State}_{t-1}^j) = \Pr(\mathbf{i}_{t-1}^j)$ and incorporates the instantiation $\mathbf{i}_t^j$. For each time slice, $2^{|\mathbf{I}|}$ passes trough the circuit are required to compute the forward message. The **advantage** of this encoding is that it combines the advantages of ENC2 and ENC3. More precisely, the benefit of evaluating the circuit multiple times (ENC3) is that the cost of compilation is amortized over all queries. The benefit of conditioning (ENC2) is that subcircuits and computations are shared. By using the hybrid approach, we get some of both advantages, which we will empirically show to be a good trade-off.

4.2. Exploiting Repeated Structure in the Network

The use of knowledge compilation to compute the forward message does not only allow us to exploit the local, but also the repeated structure in the network. Since the structure of the transition model is time-invariant, there is no need to repeat the process of encoding and compiling the 1.5TBN and introducing $\Pr(\mathbf{I}_{t-1})$. This allows us to split Equation 4 in two parts:

$$\Delta_R = \textsc{Addi}(\textsc{Compile}(KB_{1.5}), \mathbf{I}_{t-1}), \tag{5}$$

which is performed only once, and

$$\Pr(\mathbf{i}_t^j) = \textsc{Eval}_{\uparrow}(\Delta_R, w) \tag{6}$$

which is performed for each $\mathbf{i}_t^j \in \mathbf{I}_t$ and for each $t < T$. Hence, the one-time cost of Equation 5 is amortized over $2^{|\mathbf{I}|} \cdot T$ queries. Note that for the standard interface algorithm, the one-time cost of compiling the transition model into a junction tree is amortized over T queries. This approach, however, does not exploit any of the local structure in the transition model.

4.3. Simplifying the Circuit

Computing the forward message by means of Equation 6 requires an update of the weight-function w before any new evaluation pass trough Δ_R. Some variables in the d-DNNF, however, are mapped to time-invariant weights that never change. They can be combined and replaced by a smaller set of new variables in case the following two conditions are met: (1) the variable is not observed and, (2) not queried.

In general, all of the parameter variables and a subset of the indicator variables meet these two conditions. For example, variable $W3_t$, which models the state of *wire 3* in our running example, is a purely internal variable and never queried or observed. Assume we have a d-DNNF which contains the following sub-formula and weight function:

$$W4_t \wedge (W3_t \wedge P_{W4_t|W3_t,t}) \qquad \text{with} \qquad \begin{cases} w(W3_t) & = a \\ w(P_{W4_t|W3_t,t}) & = b \end{cases}$$

This can be replaced by:

$$W4_t \wedge P_{new,t} \qquad \text{with} \qquad w(P_{new,t}) = a \cdot b$$

The effect of this transformation is that it reduces the number of unnecessary computations in each pass trough the circuit. If we would not employ this transformation, the multiplication $a \cdot b$ will be performed $T \cdot 2^{|\mathbf{I}|}$ times although the result will always be the same. This transformation can be performed in a deterministic manner by means of one bottom-up pass trough the d-DNNF. As it only needs to be computed once, i.e. before the evaluation step, the cost is amortized over $T \cdot 2^{|\mathbf{I}|}$ queries.

5. Experiments

Our experiments address the following four questions:

Q1 How do different algorithms scale with an increasing number of time steps?

Q2 How do both of the interface algorithms scale in the presence of local structure in the transition model?

Q3 How does the structural interface algorithm scale in case local structure is not fully exploited?

Q4 How do the different interface encodings compare?

We implemented our algorithm in ProbLog[4]. For compilation, we use both the `c2d`[5] and `dsharp`[6] compilers, and retain the smallest circuit. Experiments are run with a working memory of 8 GB and a timeout of 1 hour.

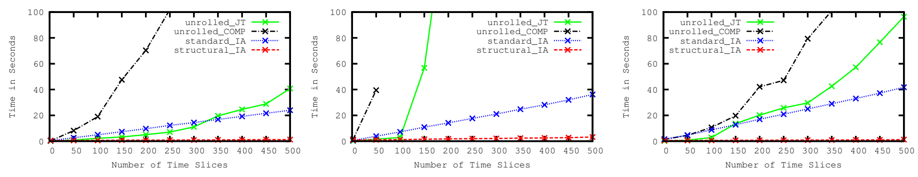
5.1. Models

We generate networks for the following three domains:

Digital Circuit 1. These networks model electronic digital circuits similar to the one used as running example in this text (and adopted from Darwiche [5]). A circuit contains logical AND-gates and OR-gates which all are randomly connected to each other (without forming loops). For a subset of logical gates, the input or output is observed and not connected to another gate. The interface contains all variables that model the health state of the component. Gates can share a health variable when, for example, they share a power line. We refer to the networks as `DC1-G-H`, with `G` the number of gates and `H` the number of health (interface) variables. The number of gates for which the input or output is observed is $2 \cdot \frac{G}{H}$. Observations are generated randomly. For each domain size, we randomly generate 3 networks and report average results.

Digital Circuit 2. These networks are a variant of the networks in `DC1` but now we have a separate health variable for each of the gates and the interface consists of one multi-valued variable. This variable aggregates all health variables and encodes, in an ordered way, which gate is most likely to be part of the failing gates. The introduction of the multi-valued variable facilitates the encoding of the interface, as compared to `DC1`, but offers an additional challenge for inference as it directly depends on each of the health variables. We refer to the networks as `DC2-G` with `G` the number of gates. For each domain size, we randomly generate 3 networks and report average results.

[4]Available at `http://dtai.cs.kuleuven.be/problog/`
[5]Available at `http://reasoning.cs.ucla.edu/c2d/`
[6]Available at `https://bitbucket.org/haz/dsharp`



(a) `DC1-20-5` (b) `DC2-12` (c) `MM-3-3`

Figure 3: Total inference time for an increasing number of time slices.

Mastermind. We model the mastermind game, similar to the BNs used in Chavira et al. [4]. Instead of modeling the game for a fixed number of rounds, however, we represent the game as a DBN with one time slice per round. The interface contains a variable for each of the pegs the game is played with. The interface thus models the belief of the colors set by the opponent for each of the pegs. We refer to the networks as `MM-C-P`, with `C` the number colors and `P` the number of pegs (interface variables).

5.2. Algorithms

We make use of the following four algorithms:

- `unrolled_JT` : The *junction tree algorithm* on the unrolled network for which we used SMILE[7].
- `unrolled_COMP` : Compiling the unrolled network, using the encoding introduced in section 3.2.1.
- `standard_IA` : The standard *interface algorithm*[8] where we experimented with the with `jtree_dbn_inf_engine` as well as with the `smoother_engine` but did not observe any significant difference.
- `structural_IA` : The *structural interface algorithm* where the interface is encoded using ENC4.

5.3. Results

We compare the four algorithms introduced above for an increasing number of time-slices. The results are depicted in Figure 3 and allow us to answers **(Q1)**. On each of the three domains, both of the interface algorithms scale linear with the number of time steps while this is not the case for inference in the unrolled network. This shows that, especially for a large number of time-slices, the general-purpose heuristics fail to find a good variable ordering. We do observe, however, that `unrolled_JT` is more efficient, compared to `standard_IA`, when the number of time-slices is rather small. The reason for this is that `standard_IA`

[7]Available at `http://genie.sis.pitt.edu/`
[8]Available at `https://github.com/bayesnet/bnt`

	1.5TBN				$KB_{1.5}$		d-DNNF Edges		d-DNNF Time		standard_IA
Model	Vars	Max Clust	Card	Avg Card	Vars	clauses	$\Delta_{1.5}$ #edges	Δ_R #edges	comp (s)	Rinf (s)	Tinf (s)
DC1-G-H							×1000	×1000			
20 - 5	34	16	2-2	2.0	146	373	12	3	0.1	0.002	1
30 - 5	46	21	2-2	2.0	216	597	31	6	0.2	0.003	4
40 - 5	58	$\geqslant$27	2-2	2.0	282	801	61	11	0.3	0.005	-
60 - 6	82	$\geqslant$29	2-2	2.0	416	1,214	372	57	2.0	0.03	-
70 - 7	94	$\geqslant$28	2-2	2.0	482	1,413	1,235	145	6.8	0.1	-
112 - 7	142	$\geqslant$29	2-2	2.0	770	2,343	6,870	959	42.3	0.7	-
80 - 8	106	$\geqslant$32	2-2	2.0	548	1,612	3,059	333	18.0	0.3	-
104 - 8	133	$\geqslant$29	2-2	2.0	704	2,105	9,886	1,090	61.7	1.2	-
90 - 9	118	$\geqslant$29	2-2	2.0	614	1,811	9,051	851	56.0	1.5	-
DC2-G											
12	30	16	2-13	2.7	157	615	26	7	0.3	0.003	1
16	38	20	2-17	2.8	209	963	95	16	0.7	0.007	9
20	46	$\geqslant$22	2-21	2.8	261	1,375	310	38	2.2	0.02	-
24	54	$\geqslant$26	2-25	2.9	313	1,851	643	106	5.0	0.05	-
28	62	$\geqslant$30	2-29	2.9	365	2,391	3,050	376	23.1	0.2	-
32	70	$\geqslant$34	2-33	2.9	417	2,995	7,300	668	57.1	0.4	-
MM-C-P											
3 - 3	59	11	2-3	2.2	147	447	62	2	0.2	0.001	1
6 - 3	59	11	2-6	2.6	210	699	519	24	1.3	0.02	2
9 - 3	59	11	2-9	3.1	273	1,032	1,944	88	4.9	0.1	3
4 - 4	99	$\geqslant$20	2-4	2.2	293	1,058	4,590	55	8.7	0.05	-
6 - 4	99	$\geqslant$20	2-6	2.5	357	1,326	27,656	361	55.2	1.2	-
8 - 4	99	$\geqslant$20	2-8	2.7	421	1,642	98,120	1,350	220.7	13.6	-
3 - 5	149	$\geqslant$25	2-3	2.1	417	1,769	13,234	75	23.6	0.07	-
4 - 5	149	$\geqslant$25	2-4	2.2	462	1,934	58,467	519	128.6	1.7	-

Table 3: Results for computing the forward message for 10 time-slices with `structural_IA` and `standard_IA`. *Max Clust* denotes the biggest cluster in the junction tree and *(Avg) Card* denotes the (average) cardinality of the variables in the transition model. *comp* includes the compilation time, conditioning time and the time to simplify the circuit (cf. Section 4.3). *Rinf* denotes the time needed with `structural_IA` to compute the forward message for one time slice. *Tinf* denotes the total inference needed by `standard_IA`.

has to deal with an extra constraint, being that all variables in the interface have to be in the same clique, which initially causes some overhead. Furthermore, `unrolled_JT` outperforms `unrolled_COMP` on each of the three domains despite the local structure present in the networks. Hence, we can state that that no guarantees can be provided when a general-purpose implementation is used to perform inference in the unrolled network.

We compare `standard_IA` and `structural_IA` for the task of computing the forward message for 10 time-slices. The results are depicted in Table 3 and serve as an answer to **Q2**. The structural interface algorithm, which exploits local structure, successfully performs inference on all of the networks while this is not the case for the standard interface algorithm. Furthermore, this table indicates that `structural_IA` works well in case the transition model is complex while the number of variables in the interface is rather limited. For example, `DC1-90-9` requires more compilation and evaluation time than `DC1-112-7`, although the latter contains more variables. This is explained by the exponential behaviour of the interface.

We explore the effect of exploiting local structure by the CNF encoding when compiling the network. The results are depicted in Table 4 and serve as an answer to **Q3**. Concretely, we consider a CNF encoding that does not exploit any local structure, a CNF encoding that only exploits determinism and a CNF encoding that exploits determinism as well as equal parameters. We observe

Model	No Local Structure Δ_R #edges	No Local Structure comp (s)	Only Det Δ_R #edges	Only Det comp (s)	Det & Equal Par Δ_R #edges	Det & Equal Par comp (s)	standard_IA Tinf (s)
DC1-G - H	×1000		×1000		×1000		
20 - 5	4,262	10.4	906	2.6	17	0.1	1
30 - 5	193,594	588.6	19,789	48.6	36	0.2	4
DC2-G							
12	-	-	1,289	2.5	26	0.1	1
16	-	-	2,039	4.1	95	0.4	9
MM-C-P							
3 - 3	1,096	3.2	15	0.3	38	0.1	1
6 - 3	-	-	-	-	441	4.5	2

Table 4: A comparison of different levels of exploiting local structure in the transition model. We use interface encoding ENC1 en do not simplify the circuit. Hence, *comp* only includes compilation time. *Tinf* denotes the total inference needed by `standard_IA` to compute the forward message for 10 time-slices.

Model	ENC1 Δ_R #edges	ENC1 comp (s)	ENC1 Rinf (s)	ENC2 Δ_R #edges	ENC2 comp (s)	ENC2 Rinf (s)	ENC3 Δ_R #edges	ENC3 comp (s)	ENC3 Rinf (s)	ENC4 Δ_R #edges	ENC4 comp (s)	ENC4 Rinf (s)
DC1-G-H	×1000			×1000			×1000			×1000		
20 - 5	5	0.2	0.004	15	0.2	0.01	1	0.09	0.02	3	0.1	0.002
30 - 5	7	0.2	0.006	18	0.3	0.02	4	0.2	0.02	6	0.2	0.003
40 - 5	13	0.4	0.01	24	0.5	0.02	10	0.3	0.05	11	0.3	0.005
60 - 6	63	2.3	0.05	116	2.5	0.1	54	1.9	0.8	57	2.0	0.03
70 - 7	171	8.4	0.1	389	9.3	0.4	136	6.7	6.6	145	6.8	0.1
112 - 7	975	45.6	0.8	1,222	44.9	1.0	950	42.2	48.7	959	42.3	0.7
80 - 8	416	24.1	0.4	1,386	28.2	1.4	314	17.9	56.4	333	18.0	0.3
104 - 8	1,172	73.7	1.0	2,160	71.8	2.0	1,070	62.0	212.8	1,090	61.7	1.2
90 - 9	1,140	86.3	1.0	5,317	99.6	5.7	807	55.7	-	851	56.0	1.5
DC2-G												
12	9	0.2	0.007	10	0.3	0.009	8	0.2	0.01	7	0.3	0.003
16	18	0.6	0.02	22	0.9	0.02	17	0.6	0.04	16	0.7	0.007
20	42	1.9	0.03	48	2.4	0.04	40	1.8	0.1	38	2.2	0.02
24	113	4.6	0.09	124	5.5	0.1	110	4.4	0.4	106	5.0	0.05
28	387	22.5	0.3	403	23.9	0.3	382	22.2	1.7	376	23.1	0.2
32	684	56.2	0.5	707	58.1	0.6	677	55.5	3.7	668	57.1	0.4
MM-C-P												
3 - 3	1	0.2	0.001	2	0.2	0.001	2	0.2	0.02	2	0.2	0.001
6 - 3	21	4.9	0.01	24	5.8	0.01	20	1.1	3.0	24	1.3	0.02
9 - 3	75	164.7	0.04	88	92.17	0.05	56	3.7	68.8	88	4.9	0.1
4 - 4	50	11.6	0.03	54	14.0	0.03	77	8.5	13.6	55	8.7	0.05
6 - 4	495	1024.5	0.4	360	275.3	0.2	396	53.3	-	361	55.2	1.2
8 - 4	-	-	-	-	-	-	1,312	198.5	-	1,350	220.7	13.6
3 - 5	6	11.5	0.03	74	27.0	0.04	171	22.7	27.9	75	23.6	0.07
4 - 5	1,401	929.4	4.7	515	228.0	0.3	879	125.0	-	519	128.6	1.7

Table 5: A comparison of the different encodings for the interface. *comp* includes the compilation time, conditioning time (if applicable) and the time to simplify the circuit (cf. Section 4.3). *Rinf* denotes the time needed to compute the forward message for one time slice.

that, in case no local structure is exploited, the transition model is much harder to compile and results in very large circuits. Moreover, `standard_IA` clearly outperforms `structural_IA` in case the latter does not exploit local structure. Only exploiting determinism significantly simplifies the compilation process but, for most networks, we can still benefit from also exploiting equal parameters.

We compare the four different interface encodings proposed in Section 4. The results are shown in Table 5 and let us answer **Q4**. We first observe that ENC4, i.e. the encoding we propose for the structural interface algorithm, is the only encoding that successfully performs inference in each of the networks. Second, the mastermind experiment illustrates that compiling the knowledge base is harder when using ENC1, as was suggested by the complexity indicated in Table 2. Third, the compilation step for ENC3 is the most efficient one, as it does not compile the interface. Computing the forward message, however, is in general much slower compared to the other encodings, as also indicated in Table 2. Fourth, although the d-DNNF for ENC3 does not encode the interface, its size is in general not smaller compared to the other encodings. The reason for this is that by explicitly encoding the interface we actually do not increase the number of models in the d-DNNF but rather add extra constraints on the models already present. Hence, explicitly encoding the interface might increase the total compilation time but significantly reduces the evaluation time.

6. Conclusions

In this paper, we proposed a new inference algorithm, the *structural interface algorithm*, for dynamic Bayesian networks based on knowledge compilation. This algorithm improves on the state-of-the-art because it (1) uses the repeated nature of the model, (2) exploits local structure, and (3) reduces the size of the resulting circuit. This approach can tackle dynamic models that are considerably more complex than what can currently be dealt with by exact inference techniques. We have experimentally shown this on two classes of problems, namely finding failures in an electronic circuit and performing filtering in the mastermind game.

Acknowledgements. This work was supported by IWT-SBO-100031 POM2. JV is supported by IWT. GVdB is supported by FWO. The authors like to thank Manfred Jaeger for supplying the mastermind models.

References

[1] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann Publishers Inc., 1988.

[2] T. Dean, K. Kanazawa, A Model for Reasoning About Persistence and Causation, Computational Intelligence 5 (2) (1989) 142–150.

[3] K. Murphy, Dynamic Bayesian Networks: Representation, Inference and Learning, Ph.D. thesis, UC Berkeley, Computer Science Division (2002).

[4] M. Chavira, A. Darwiche, M. Jaeger, Compiling relational Bayesian networks for exact inference, International Journal of Approximate Reasoning 42 (1-2) (2006) 4–20.

[5] A. Darwiche, Modeling and Reasoning with Bayesian Networks, Cambridge University Press, 2009.

[6] D. Koller, N. Friedman, Probabilistic graphical models: principles and techniques, MIT press, 2009.

[7] A. Darwiche, Constant-space reasoning in dynamic Bayesian networks, International Journal of Approximate Reasoning 26 (3) (2001) 161–178.

[8] L. R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, in: Proceedings of the IEEE, 1989, pp. 257–286.

[9] C. Boutilier, N. Friedman, M. Goldszmidt, D. Koller, Context-specific Independence in Bayesian Networks, in: Proceedings of the Twelfth International Conference on Uncertainty in Artificial Intelligence (UAI), 1996, pp. 115–123.

[10] M. Chavira, A. Darwiche, On probabilistic inference by weighted model counting, Artificial Intelligence 172 (6-7) (2008) 772–799.

[11] D. Fierens, G. Van den Broeck, J. Renkens, D. Shterionov, B. Gutmann, I. Thon, G. Janssens, L. De Raedt, Inference and learning in probabilistic logic programs using weighted Boolean formulas, Theory and Practice of Logic Programming 15 (3) (2015) 358–401.

[12] A. Darwiche, P. Marquis, A Knowledge Compilation Map, Journal of Artificial Intelligence Research 17 (2002) 229–264.

[13] A. Darwiche, New Advances in Compiling CNF into Decomposable Negation Normal Form., in: Proceedings of European Conference on Artificial Intelligence (ECAI), 2004, pp. 328–332.