Lifted Dynamic Junction Tree Algorithm [⋆]

Marcel Gehrke, Tanya Braun, and Ralf Möller

Institute of Information Systems, Universität zu Lübeck, Lübeck
{gehrke, braun, moeller}@ifis.uni-luebeck.de

Abstract. Probabilistic models involving relational and temporal aspects need exact and efficient inference algorithms. Existing approaches are approximative, include unnecessary grounding, or do not consider the relational and temporal aspects of the models. One approach for efficient reasoning on relational static models given multiple queries is the lifted junction tree algorithm. In addition, for propositional temporal models, the interface algorithm allows for efficient reasoning. To leverage the advantages of the two algorithms for relational temporal models, we present the lifted dynamic junction tree algorithm, an exact algorithm to answer multiple queries efficiently for probabilistic relational temporal models with known domains by reusing computations for multiple queries and multiple time steps. First experiments show computational savings while appropriately accounting for relational and temporal aspects of models.

1 Introduction

Areas like healthcare and logistics involve probabilistic data with relational and temporal aspects and need efficient exact inference algorithms, e.g, as indicated by Vlasselaer et al. [23]. These areas involve many objects in relation to each other with changing information over time and uncertainties about objects, objects attributes, or relations. More specifically, healthcare systems involve electronic health records (the relational part) for many patients (the objects), streams of measurements over time (the temporal part), and uncertainties due to, e.g., missing or incomplete information, for example caused by data integration of records from different hospitals. By performing model counting, probabilistic databases (PDBs) can answer huge queries, which embed most of the model behaviour, for relational temporal models with uncertainties [5,6]. However, we build more expressive and compact models including behaviour (offline) enabling efficient answering of smaller queries (online). To be more precise, for query answering we perform deductive reasoning by computing marginal distributions at discrete time steps. In this paper we study the problem of exact inference, in form of filtering and prediction, in large temporal models that exhibit symmetries.

For exact inference on propositional temporal models, a naive approach is to unroll the temporal model for a given number of time steps and use any exact inference algorithm for static, i.e., non-temporal, models. In the worst case, once

[⋆] This research originated from the Big Data project being part of Joint Lab 1, funded by Cisco Systems Germany, at the centre COPICOH, University of Lübeck the number of time steps changes, one has to unroll the model and infer again. To prevent the complete unrolling, Murphy proposes the interface algorithm [13].

One reasoning approach for leveraging the relational aspect of a static model is first-order probabilistic inference. For models with known domain size, it exploits symmetries in a model by combining instances to reason with representatives, known as lifting [16]. Poole introduces parametric factor graphs as relational models and proposes lifted variable elimination (LVE) as an exact inference algorithm on relational models [16]. Further, de Salvo Braz [18], Milch et al. [11], and Taghipour et al. [20] extend LVE into its current form. Lauritzen and Spiegelhalter [9] introduce the junction tree algorithm. To benefit from the ideas of the junction tree algorithm and LVE, Braun and Möller [2] present the lifted junction tree algorithm (LJT) that efficiently performs exact first-order probabilistic inference on relational models given a set of queries.

We aim at an exact and efficient inference algorithm for a set of queries that handles both the relational and the temporal aspect. To this end, we present the lifted dynamic junction tree algorithm (LDJT) that combines the advantages of the interface algorithm and LJT. Specifically, this paper contributes (i) a definition for parameterised probabilistic dynamic models (PDMs) as a representation for relational temporal models, and (ii) a formal description of LDJT, a reasoning algorithm for PDMs, a set of queries, and a set of observations (evidence).

Related work for inference on relational temporal models mostly consists of approximative approaches. Additionally, to being approximative, these approaches involve unnecessary groundings or are only designed to handle single queries efficiently. Ahmadi et al. propose a lifted (loopy) belief propagation [1]. From a factor graph, they build a compressed factor graph and apply lifted belief propagation with the idea of the factored frontier algorithm [12], which is an approximate counterpart to the interface algorithm [13]. Thon et al. introduce CPT-L, a probabilistic model for sequences of relational state descriptions with a partially lifted inference algorithm [21]. Geier and Biundo present an online interface algorithm for dynamic markov logic networks (DMLNs) [7], similar to the work of Papai et al. [15]. Both approaches slice DMLNs to run well-studied static MLN [17] inference algorithms on each slice individually. Further, the interface algorithm also slices the model to utilise static approaches. Two ways of performing online inference using particle filtering are described in [10,14].

Vlasselaer et al. introduce an exact approach for relational temporal models involving computing probabilities of each possible interface assignment [22].

LDJT has several benefits: The lifting approach exploits symmetries in the model to reduce the number of instances to perform inference on. For answering multiple queries, the junction tree idea enhances efficiency by clustering a model into submodels sufficient for answering a particular query. Further, the interface idea drastically reduces the size of the model and allows adding time steps dynamically, by an efficient separation of time steps. Furthermore, the junction tree structure of the model is reused for all time steps $t > 0$.

The remainder of this paper has the following structure: We begin by introducing PDMs to represent relational temporal models. Followed by, LDJT an efficient reasoning algorithm for PDMs and evaluate LDJT compared to LJT and a ground interface approach. We conclude by looking at possible extensions.

2 Parameterised Probabilistic Dynamic Models

We introduce parameterised probabilistic models (PMs), which is mainly based on [3], as a representation for relational static models. Afterwards, we extend PMs to the temporal case, resulting in PDMs for relational temporal models.

2.1 Parameterised Probabilistic Models

PMs combine first-order logic with probabilistic models, representing first-order constructs using logical variables (logvars) as parameters.

Definition 1. *We define a basic block with* $\mathbf{L}$ *as a set of logvar names,* Φ *as set of factor names, and* $\mathbf{R}$ *a set of random variable (randvar) names. A parameterised randvar (PRV)* $A = P(X^1, ..., X^n)$ *represents a set of randvars behaving identically by combining a randvar* $P \in \mathbf{R}$ *with* $X^1, ..., X^n \in \mathbf{L}$*. If* $n = 0$*, the PRV is parameterless. The domain of a logvar* L *is denoted by* $\mathcal{D}(L)$*. The term* $range(A)$ *provides possible values of a PRV* A*. Constraint* $(\mathbf{X}, C_{\mathbf{X}})$ *allows to restrict logvars to certain domain values and is a tuple with a sequence of logvars* $\mathbf{X} = (X^1, ..., X^n)$ *and a set* $C_{\mathbf{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X^i)$*. The symbol* $\top$ *denotes that no restrictions apply and may be omitted. The term* $lv(Y)$ *refers to the logvars in some element* Y*. The term* $rv(Y)$ *refers to the randvars in* Y*. The term* $gr(Y)$ *denotes the set of instances of* Y *with all logvars in* Y *grounded w.r.t. constraints.*

Now, we illustrate PMs with an example, which has the goal to remotely infer the condition of patients with regards to water retaining. To determine the condition of patients, we use the change of their weights. Further, we are interested in the condition of people the patient is living with, giving us indications to improve the inferred conditions. In case patients are living together and both are gaining weight, they probably overeat and do not retain water. If no new weights are submitted, we are interested whether the scale broke or the patient stopped submitting weights. In case only one patient stops to send weights, it is likely that the patient stopped deliberately. If patients living together stop to submit weights at the same time, it is more likely that their scale broke. Hence, we can improve the accuracy of inference by accounting for patients living together.

To model the example, we use the randvar names C, LT, S, and W for Condition, LivingTogether, ScaleWorks, and Weight, respectively, and the logvar names X and X'. From the names, we build PRVs $C(X)$, $LT(X, X')$, $S(X)$, and $W(X)$. The domain of X and X' is the set $\{alice, bob, eve\}$. The range of $C(X)$ is $\{normal, deviation, retains\ water, stopped\}$, $LT(X, X')$ and $S(X)$ have range $\{true, false\}$, and of $W(X)$ has range $\{steady, falling, rising, null\}$. Now, we define parametric factors (parfactors), to set PRVs into relation to each other.

Definition 2. *We define a parfactor* g *with* $\forall \mathbf{X} : \phi(\mathcal{A})\ |C$*.* $\mathbf{X} \subseteq \mathbf{L}$ *being a set of logvars over which the factor generalises and* $\mathcal{A} = (A^1, ..., A^n)$ *a sequence of*

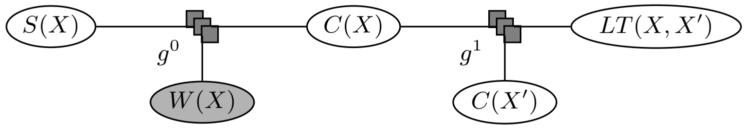


Fig. 1. Parfactor graph for model G^{ex} with observable nodes in grey

PRVs. We omit $(\forall \mathbf{X} :)$ *if* $\mathbf{X} = lv(\mathcal{A})$*. A function* $\phi : \times_{i=1}^{n} range(A^i) \mapsto \mathbb{R}^+$ *with name* $\phi \in \Phi$ *is identically defined for all grounded instances of* $\mathcal{A}$*. A list of all input-output values is the complete specification for* ϕ*. The output value is called potential.* C *is a constraint on* $\mathbf{X}$*. A PM (model)* $G := \{g^i\}_{i=0}^{n-1}$ *is a set of parfactors and represents the full joint probability distribution* $P(G) = \frac{1}{Z} \prod_{f \in gr(G)} \phi(\mathcal{A}_f)$ *where* Z *is a normalisation constant.*

Now, we build the model G^{ex} of our example with the parfactors:

$$g^0 = \phi^0(C(X), S(X), W(X))|\top \text{ and } g^1 = \phi^1(C(X), C(X'), LT(X, X'))|\kappa^1$$

We omit the concrete mappings of ϕ^0 and ϕ^1. Parfactor g^0 has the constraint $\top$, meaning it holds for *alice*, *bob*, and *eve*. The constraint κ^1 of g^1 ensures that $X \neq X'$ holds. Fig. 1 depicts G^{ex} as a parfactor graph and shows PRVs as nodes, which are connected via undirected edges to nodes of parfactors in which they appear. We can observe the weight of patients. The remaining PRVs are latent.

The semantics of a model is given by grounding and building a full joint distribution. In general, queries ask for a probability distribution of a randvar using a model's full joint distribution and given fixed events as evidence.

Definition 3. *Given a PM* G*, a ground PRV* Q *and grounded PRVs with fixed range values* $\mathbf{E}$ *the expression* $P(Q|\mathbf{E})$ *denotes a query w.r.t.* $P(G)$*.*

In our example, a query is $P(C(bob)|W(bob) = steady)$, asking for the probability distribution of *bob*'s condition given information about his weight.

2.2 Parameterised Probabilistic Dynamic Models

To define PDMs, we use PMs and the idea of how Bayesian networks (BNs) give rise to dynamic Bayesian networks (DBNs). We define PDMs based on the first-order Markov assumption, i.e., a time slice t only depends on the previous time slice $t-1$. Further, the underlining process is stationary, i.e., the model's behaviour does not change over time.

Definition 4. *A PDM is a pair of PMs* $(G_0, G_\rightarrow)$ *where* G_0 *is a PM representing the first time step and* $G_\rightarrow$ *is a two-slice temporal parameterised model (2TPM) representing* $\mathbf{A}_{t-1}$ *and* $\mathbf{A}_t$ *with* $\mathbf{A}_t$ *a set of PRVs from time slice* t*.*

Fig. 2 shows how the model G^{ex} behaves over time. $G^{ex}_{\rightarrow}$ consists of G^{ex} for time step $t-1$ and for time step t with inter-slice parfactors for the behaviour over time. In this example, the latent PRVs depend on each other from one time slice to the next, which is represented with the parfactors g^{LT}, g^C, and g^S.

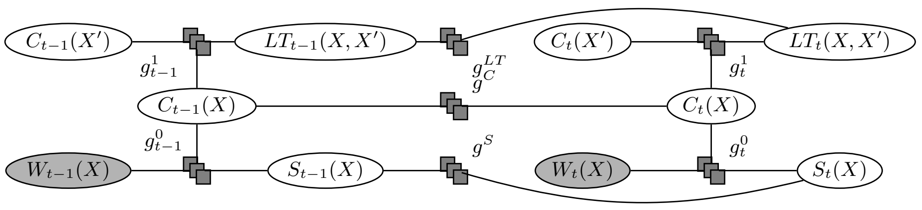


Fig. 2. $G^{ex}_{\rightarrow}$ the two-slice temporal parfactor graph for model G^{ex}

Definition 5. *Given a PDM G, a ground PRV Q_t and grounded PRVs with fixed range values $\mathbf{E}_{\{0:t\}}$ the expression $P(Q_t|\mathbf{E}_{\{0:t\}})$ denotes a query w.r.t. $P(G)$.*

The problem of answering queries for the current time step is called filtering and for a time step in the future it is called prediction.

3 Lifted Dynamic Junction Tree Algorithm

We start by recapping LJT to provide means to answer queries for PMs, mainly based on [3], and the interface algorithm, an approach to perform inference for propositional temporal models, mainly based on [13]. Afterwards, we present LDJT, consisting of a first-order junction tree (FO jtree) construction for a PDM and an efficient reasoning algorithm to perform filtering and prediction.

3.1 Lifted Junction Tree Algorithm

LJT provides efficient means to answer a set of queries $\{P(Q^i|\mathbf{E})\}_{i=1}^k$ given a PM G and evidence $\mathbf{E}$, by performing the following steps: (i) Construct an FO jtree J for G. (ii) Enter $\mathbf{E}$ in J. (iii) Pass messages. (iv) Compute answer for each query Q^i. We first define an FO jtree and then go through each step. For an FO jtree, we need parameterised clusters (parclusters), the nodes of an FO jtree.

Definition 6. *A parcluster $\mathbf{C}$ is defined by $\forall \mathbf{L} : \mathbf{A}|C$. $\mathbf{L}$ is a set of logvars, $\mathbf{A}$ is a set of PRVs with $lv(\mathbf{A}) \subseteq \mathbf{L}$, and C a constraint on $\mathbf{L}$. We omit $(\forall \mathbf{L} :)$ if $\mathbf{L} = lv(\mathbf{A})$. A parcluster $\mathbf{C}^i$ can have parfactors $\phi(\mathcal{A}^\phi)|C^\phi$ assigned given that (i) $\mathcal{A}^\phi \subseteq \mathbf{A}$, (ii) $lv(\mathcal{A}^\phi) \subseteq \mathbf{L}$, and (iii) $C^\phi \subseteq C$ hold. We call the set of assigned parfactors a local model G^i.*
An FO jtree for a model G is $J = (\mathbf{V}, \mathbf{E})$ where J is a cycle-free graph, the nodes $\mathbf{V}$ denote a set of parcluster, and the set of edges $\mathbf{E}$ the paths between parclusters. An FO jtree must satisfy the following three properties: (i) A parcluster $\mathbf{C}^i$ is a set of PRVs from G. (ii) For each parfactor $\phi(\mathcal{A})|C$ in G, $\mathcal{A}$ must appear in some parcluster $\mathbf{C}^i$. (iii) If a PRV from G appears in two parclusters $\mathbf{C}^i$ and $\mathbf{C}^j$, it must also appear in every parcluster $\mathbf{C}^k$ on the path connecting nodes i and j in J. The separator $\mathbf{S}^{ij}$ containing shared PRVs of edge $i-j$ in J is given by $\mathbf{C}^i \cap \mathbf{C}^j$. The FO jtree is minimal if by removing a PRV from any parcluster, the FO jtree stops being an FO jtree.

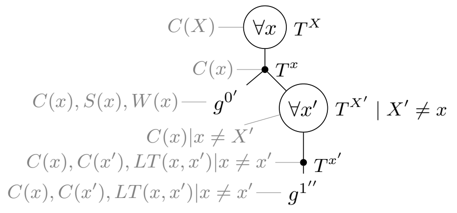


Fig. 3. FO dtree for model G^{ex}

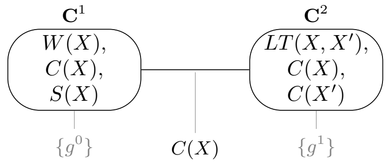


Fig. 4. Minimised FO jtree for model G^{ex}

LJT constructs an FO jtree using a first-order decomposition tree (FO dtree). Analogous to the ground case [4], LJT can use the clusters of an FO dtree to construct an FO jtree. For the details on construction of an FO dtree from a PM, the formal definition, and properties of an FO dtree, refer to Taghipour et al. [19]. For our approach, important FO dtree characteristics are: (i) each leaf node contains exactly one parfactor, (ii) the cluster for a leaf node l consists of the randvars of the corresponding parfactor in l, $rv(l)$, and (iii) a (par)cluster in an (FO) jtree corresponds to a cluster of an (FO) dtree.

Fig. 3 shows the FO dtree for the model G^{ex}, with the clusters of the FO dtree for every node in grey. An FO jtree directly constructed from clusters of an FO dtree is non-minimal. To minimise an FO jtree, LJT merges neighbouring parclusters if one parcluster is the subset of the other. The parfactors at the leaf nodes from an FO dtree determine the local models for parclusters.

LJT enters evidence in the FO jtree and passes messages through an *inbound* and an *outbound* pass, to distribute local information of the nodes through the FO jtree. To compute a message, LJT eliminates all non-seperator PRVs from the parcluster's local model and received messages. After message passing, LJT answers queries. For each query, LJT finds a parcluster containing the query term and sums out all non-query terms in its local model and received messages.

Fig. 4 shows the minimised FO jtree corresponding to the FO dtree from Fig. 3. One possibility to obtain the FO jtree is to merge the clusters of T^X into T^x and then T^x into the leaf with $g^{0'}$ and the remaining clusters of the FO dtree into the leaf with $g^{1''}$. By merging the clusters of the FO dtree, LJT acquires a minimised FO jtree. Here, each parfactor from the PM makes up the local model of a parcluster, the ideal case to answer queries. Throughout the paper, we also have FO jtrees with more parfactors in a local model of a parcluster, resulting in less messages during message passing but higher query answering efforts.

Additionally, Fig. 4 shows the local models of the parclusters and the separator PRV $C(X)$ as label of the edge. Thus, we have two parclusters, $\mathbf{C}^1$ and $\mathbf{C}^2$, in the minimised FO jtree. Before LJT answers queries, it passes messages to account for evidence. During the *inbound* phase LJT sends messages from $\mathbf{C}^1$ to $\mathbf{C}^2$ and from $\mathbf{C}^2$ to $\mathbf{C}^1$. If we want to know whether the scale from *alice* works, we have to query for $P(S(alice))$ for which LJT can use parcluster $\mathbf{C}^1$. LJT sums out $C(X)$, $W(X)$, and $S(X)$ where $X \neq alice$ from $\mathbf{C}^1$'s local model G^{ex^1} combined with the received messages, here, the one message from $\mathbf{C}^2$.

3.2 Interface Algorithm

The interface algorithm for DBNs allows to efficiently pass on the current state of the model from one time slice to the next, while being able to make use of a static junction tree algorithm for BNs. The interface algorithm defines the set of nodes with outgoing edges to the next time slice as an interface for temporal d-separation. The interface has to be in one cluster of the associated jtree. While proceeding to the next time step, the interface algorithm reuses the structure of the jtree and only passes in information from the outgoing interface cluster.

A DBN is defined using two BNs: B_0 is a BN which defines the prior and $B_{\rightarrow}$ is a two-slice temporal bayesian network (2TBN), which models the temporal behaviour. The interface algorithm uses that the set of nodes with outgoing edges, I_t, to the next time slice from $B_{\rightarrow}$ is sufficient to d-separate the past from the future. The interface algorithm builds a jtree J_0 for B_0 and ensures during the creation that I_0 ends up in a cluster of the jtree. The cluster containing I_0 is labeled *in-* and *out-cluster*. Then, the algorithm turns $B_{\rightarrow}$ into a 1.5TBN, H_t (H for half), by removing all non interface nodes N_{t-1} and their edges from the first slice of $B_{\rightarrow}$. Now, it constructs a jtree J_t for H_t and ensures that I_{t-1} and I_t each end up in clusters of the jtree. The cluster containing I_{t-1} is labeled *in-cluster* and the cluster containing I_t is labeled *out-cluster*. The idea is to pass messages α_{t-1} over I_{t-1} from the *out-cluster* of J_{t-1} to the *in-cluster* of J_t.

Fig. 5 shows how the interface algorithm uses the *in-* and *out-clusters* of the jtrees J_0 and J_t for the first three time steps to pass on the current state in each time step. To reason for $t = 0$, the interface algorithm uses J_0. First, a junction tree algorithm enters evidence in J_0 if available, passes messages, and answers queries. The interface algorithm then computes a message using the *out-cluster* of J_0, by summing out all non-interface variables, to pass the message on via the separator to J_1. For all $t > 0$, the interface algorithm instantiates J_t for that time step. Afterwards, the interface algorithm recovers the state of the model by adding the message from the *out-cluster* of J_{t-1} to the *in-cluster* of J_t. For $t = 1$ the interface algorithm instantiates J_1 and then adds the message containing I_0 to J_1's *in-cluster*. After the interface algorithm recovered the previous state by adding the message, it behaves as it did for $t = 0$. A junction tree algorithm enters evidence in J_1 if available, passes messages, and answers queries. During message passing, information from I_0 is distributed through the jtree J_1 and hence present in the *out-cluster* to compute the message over I_1.

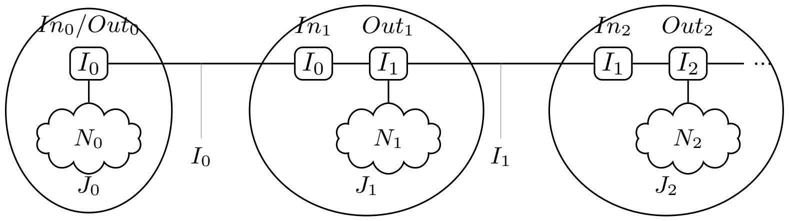


Fig. 5. Combination of jtrees using interface algorithm

3.3 LDJT: Overview

LDJT efficiently answers sets of queries $\{P(Q_t^i|\mathbf{E}_t)\}_{i=1}^k$, $Q_t^i \in \{\mathbf{Q}_t\}_{t=0}^T$, given a PDM G and evidence $\{\mathbf{E}_t\}_{t=0}^T$, by performing the following steps:

(i) Offline construction of the two FO jtrees J_0 and J_t with *in-* and *out-clusters*
(ii) For $t = 0$, using J_0 to enter $\mathbf{E}_0$, pass messages, answer $\mathbf{Q}_0$, and preserve the state in message α_0
(iii) For $t > 0$ instantiate J_t, recover the previous state from message α_{t-1}, enter $\mathbf{E}_t$ in J_t, pass messages, answer $\mathbf{Q}_t$, and preserve the state in message α_t

LDJT solves the filtering and prediction problems efficiently by reusing a compact structure for multiple queries and time steps. Further, LDJT only requires the current evidence and the state of the interface from the previous time step for queries. Next, we show how LDJT constructs the FO jtrees J_0 and J_t with *in-* and *out-clusters* and then, how LDJT uses the FO jtrees for reasoning.

3.4 LDJT: FO Jtree Construction for PDMs

The steps of FO jtree constructions are shown in Alg.1. LDJT constructs an FO jtree for G_0 and for $G_\rightarrow$, both with an incoming and outgoing interface. Therefore, LDJT first identifies the interface PRVs $\mathbf{I}_t$ for a time slice t. We define $\mathbf{I}_{t-1}$ as follows:

Definition 7. *The forward interface is defined as* $\mathbf{I}_{t-1} = \{A_{t-1}^i \mid \exists \phi(\mathcal{A})|C \in G : A_{t-1}^i \in \mathcal{A} \wedge \exists A_t^j \in \mathcal{A}\}$, *i.e., the PRVs which have successors in the next slice. The set of non-interface PRVs is* $\mathbf{N}_t = \mathbf{A}_t \setminus \mathbf{I}_t$.

To ensure interface PRVs $\mathbf{I}$ ending up in a single parcluster, LDJT adds a parfactor g^I over the interface to the model with uniform potentials in the mappings, e.g., mapping all input values to 1. $\mathbf{I}_0$ has to be assigned to a local model of a parcluster of J_0. To ensure that $\mathbf{I}_0$ ends up in a single parcluster, LDJT adds a parfactor g_0^I over $\mathbf{I}_0$ to G_0. When LDJT constructs the FO jtree,

Algorithm 1 FO Jtree Construction for a PDM $(G_0, G_\rightarrow)$

```
function DFO-JTREE($G_0, G_\rightarrow$)
    $\mathbf{I}_t$ := Set of interface PRVs for time slice $t$
    $g_0^I$ := Parfactor for $\mathbf{I}_0$
    $G_0$ := $g_0^I \cup G_0$
    $J_0$ := Construct minimized FO jtree for $G_0$
    $g_{t-1}^I$ := Parfactor for $\mathbf{I}_{t-1}$
    $g_t^I$ := Parfactor for $\mathbf{I}_t$
    $H_t$ := $\{\phi(\mathcal{A})|C \in G_\rightarrow \mid \forall A \in \mathcal{A} : A \notin \mathbf{N}_{t-1}\}$
    $G_t$ := $(g_{t-1}^I \cup g_t^I \cup H_t)$
    $J_t$ := Construct minimized FO jtree for $G_t$
return $(J_0, J_t, \mathbf{I}_t)$
```

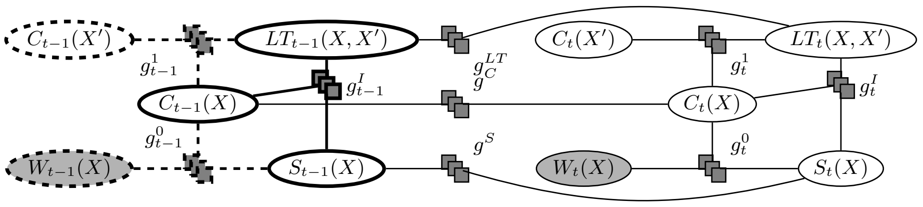


Fig. 6. Two-slice temporal parfactor graph for model $G^{ex}_{\rightarrow}$ with interface parfactors

it first constructs an FO dtree. In the FO dtree, the parfactor ends up as a leaf and the cluster of each leaf contains the randvars of the corresponding parfactor. Further, while minimising the FO jtree, the parfactor is assigned to a local model of a parcluster. Thereby, by adding the interface parfactor to the model ensures that the resulting FO jtree has an incoming and outgoing interface. LDJT labels the parcluster with g^I_0 from J_0 as *in-* and *out-cluster*.

For $G^{ex}_{\rightarrow}$, which is shown in Fig. 2, PRVs $C_{t-1}(X)$, $S_{t-1}(X)$, and $LT_{t-1}(X, X')$ have children in the next time slice, making up $\mathbf{I}_{t-1}$. Fig. 6 shows G^{ex}_0 with g^I_0, by setting $t - 1 = 0$ and removing all other PRVs and parfactors that do not belong to $t - 1$, leaving us with the PRVs, parfactors, and edges with thicker lines. Fig. 7 shows a corresponding FO jtree, with *in-* and *out-cluster* labelling. In the FO jtree, parcluster $\mathbf{C}^2_0$ is the only candidate to be the *in-* and *out-cluster*.

Having J_0, LDJT constructs J_t for the remaining time steps. During inference, time slice $t - 1$ will encode the past and the actual inference is performed on t, allowing us to transform the 2TPM into a 1.5-slice TPM H_t.

Definition 8. *$H_t = \{\phi(\mathcal{A})|C \in G_{\rightarrow} \mid \forall A \in \mathcal{A} : A \notin \mathbf{N}_{t-1}\}$, i.e., eliminating all non-interface PRVs, their parfactors, and edges from the first time slice in $G_{\rightarrow}$.*

LDJT needs to ensure that in the resulting FO jtree from $G_{\rightarrow}$, $\mathbf{I}_{t-1}$ and $\mathbf{I}_t$ each end up in parclusters. Hence, LDJT adds parfactors g^I_{t-1} and g^I_t to $G_{\rightarrow}$. LDJT constructs J_t from H_t with g^I_{t-1} and g^I_t and labels the parcluster containing g^I_{t-1} as *in-cluster* and the parcluster containing g^I_t as *out-cluster*.

Fig. 6 shows $G^{ex}_{\rightarrow}$ with g^I_{t-1} and g^I_t added. By removing all nodes, parfactors, and edges with dashed lines the result is a 1.5-slice TPM of $G^{ex}_{\rightarrow}$ with g^I_{t-1} and g^I_t added. Fig. 8 shows a corresponding FO jtree. To label the *in-cluster*, LDJT

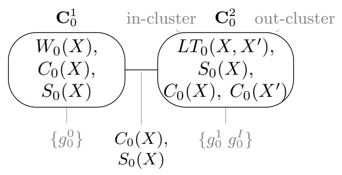
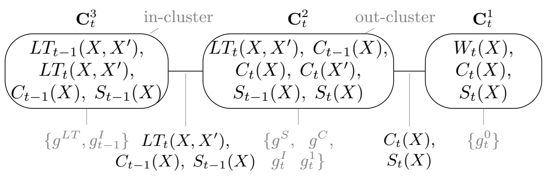


Fig. 7. FO jtree J_0 for G^{ex}_0



$\mathbf{C}^3_t$
in-cluster
$\mathbf{C}^2_t$
out-cluster
$\mathbf{C}^1_t$
$LT_{t-1}(X,X')$, $LT_t(X,X')$, $C_{t-1}(X)$, $S_{t-1}(X)$
$LT_t(X,X')$, $C_{t-1}(X)$, $C_t(X)$, $C_t(X')$, $S_{t-1}(X)$, $S_t(X)$
$W_t(X)$, $C_t(X)$, $S_t(X)$
$\{g^{LT}, g^I_{t-1}\}$
$LT_t(X,X')$, $C_{t-1}(X)$, $S_{t-1}(X)$
$\{g^S,\ g^C,\ g^I_t\ g^1_t\}$
$C_t(X)$, $S_t(X)$
$\{g^0_t\}$

Fig. 8. FO jtree J_t for $G^{ex}_{\rightarrow}$

searches for a parcluster with g^I_{t-1} in its local model, which $\mathbf{C}^3_t$ has, and labels the parcluster as *in-cluster*. LDJT does the same for g^I_t and labels $\mathbf{C}^2_t$ as *out-cluster*.

3.5 LDJT: Reasoning with FO Jtrees from PDMs

Alg. 2 provides the steps for answering queries. Since J_0 and J_t are static, LDJT uses LJT as a subroutine passing on an already constructed FO jtree, queries, and evidence for step t and lets LJT handle evidence entering, message passing, and query answering. For the first time step $t = 0$, LDJT uses J_0, takes the queries and evidence for $t = 0$ and uses LJT to answer the queries. After all queries are answered, LDJT needs to preserve the current state to pass it on to the next time slice. Therefore, LDJT uses the *out-cluster* parcluster, sums out all non-interface PRVs from that parcluster, and saves the result in message α_0, which holds the state of the PRVs that have an impact on the next time slice and thus, encodes the current state. Afterwards, LDJT increases t by one.

For time steps $t > 0$, LDJT uses J_t and first recovers the state of the previous time step by adding α_{t-1} to the *in-cluster* of J_t. LDJT then uses LJT to perform filtering. During message passing of LJT, information from the previous state is distributed through the FO jtree. After query answering, LDJT sums out all non-interface PRVs from the *out-cluster* of J_t, saves the result in message α_t, and increases t. Using the interface clusters, the FO jtrees are m-separated from one time step to the next and LDJT can use J_t once constructed for all $t > 0$.

Fig. 9 depicts how LDJT uses the interface to pass on the current state from time step three to four. First, LDJT enters evidence for $t = 3$ using LJT. Afterwards, LJT distributes local information by message passing. To capture the state at $t = 3$, LDJT needs to sum out the non-interface PRVs $C_2(X)$, $C_3(X')$, and $S_2(X)$ from $\mathbf{C}^2_3$ and save the result in message α_3. Thus, LDJT sums out the non-interface PRVs of the parfactors g^S, g^C, g^I_3, g^1_3 and the received messages m^{12}_3 and m^{32}_3. After increasing t by one, LDJT adds α_3 to the *in-cluster* of J_4, $\mathbf{C}^3_4$. α_3 is then distributed by message passing and accounted for in α_4.

Algorithm 2 LDJT Alg. for PDM $(G_0, G_\rightarrow)$, Queries $\{\mathbf{Q}\}^T_{t=0}$, Evidence $\{\mathbf{E}\}^T_{t=0}$

```
procedure LDJT($G_0, G_\rightarrow, \{\mathbf{Q}\}^T_{t=0}, \{\mathbf{E}\}^T_{t=0}$)
    $t := 0$
    $(J_0, J_t, \mathbf{I}_t) :=$ DFO-JTREE($G_0, G_\rightarrow$)
    while $t \neq T$ do
        if $t = 0$ then
            $J_0 :=$ LJT($J_0, \mathbf{Q}_0, \mathbf{E}_0$)    $\triangleright$ Including query answering, no FO jtree const.
            $\alpha_0 := \sum_{J_0(\text{out-cluster})\setminus\mathbf{I}_0} J_0(\text{out-cluster})$
            $t := t + 1$
        else
            $J_t(\text{in-cluster}) := \alpha_{t-1} \cup J_t(\text{in-cluster})$
            $J_t :=$ LJT($J_t, \mathbf{Q}_t, \mathbf{E}_t$)    $\triangleright$ Including query answering, no FO jtree const.
            $\alpha_t := \sum_{J_t(\text{out-cluster})\setminus\mathbf{I}_t} J_t(\text{out-cluster})$
            $t := t + 1$
```

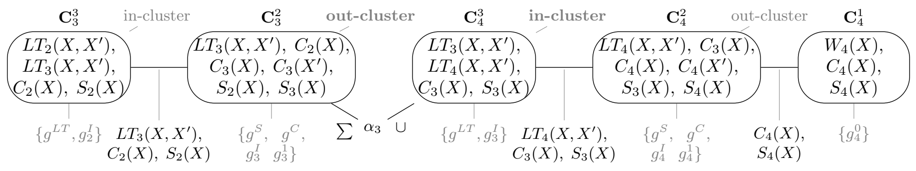

$\mathbf{C}_3^3$
in-cluster
$LT_2(X,X')$, $LT_3(X,X')$, $C_2(X)$, $S_2(X)$
$\{g^{LT}, g_2^I\}$
$LT_3(X,X')$, $C_2(X)$, $S_2(X)$
$\mathbf{C}_3^2$
out-cluster
$LT_3(X,X')$, $C_2(X)$, $C_3(X)$, $C_3(X')$, $S_2(X)$, $S_3(X)$
$\{g^S, g^C, g_3^I, g_3^1\}$
$\sum$ α_3 $\cup$
$\mathbf{C}_4^3$
in-cluster
$LT_3(X,X')$, $LT_4(X,X')$, $C_3(X)$, $S_3(X)$
$\{g^{LT}, g_3^I\}$
$LT_4(X,X')$, $C_3(X)$, $S_3(X)$
$\mathbf{C}_4^2$
out-cluster
$LT_4(X,X')$, $C_3(X)$, $C_4(X)$, $C_4(X')$, $S_3(X)$, $S_4(X)$
$\{g^S, g^C, g_4^I, g_4^1\}$
$C_4(X)$, $S_4(X)$
$\mathbf{C}_4^1$
$W_4(X)$, $C_4(X)$, $S_4(X)$
$\{g_4^0\}$

Fig. 9. LDJT passes on the current state to the next time step, J_3 shown without $\mathbf{C}_3^1$

Given a stream of evidence, a stream of queries, which can be the same for each time step, and a PDM, LDJT performs filtering and prediction. To perform filtering, LDJT enters the current evidence, e.g., the patients weight, into the current FO jtree, which already accounts for the past, and answers queries, e.g. what is the condition of a patient. Further, LDJT performs prediction, for example given the evidence so far what is the condition of the patient in 10 time steps. To perform prediction, LDJT has to enter the current evidence, e.g. the patients weight, in the current FO jtree and passes on this information through all FO jtrees until LDJT reaches the time step the query is designated for and then answers the queries. LDJT efficiently solves the problem of performing filtering and prediction for PDMs by using a compact structure, which is also reused for all $t > 0$. Further, LDJT only calculates one additional message per time step for temporal m-separation.

Theorem 1. *LDJT is correct regarding filtering and prediction*

Proof. The interface PRVs m-separate the time slices for a given PDM. Using interface parfactors, LDJT ensures that the FO jtrees for the initial time step and the copy pattern $G_{\rightarrow}$ have an *in-cluster* and an *out-cluster*. The interface parfactors have uniform potentials in the mappings, therefore, have no impact on message passing or calculating answers to queries, besides a scaling factor. Further, the interface message α_t is equivalent to having the PDM unrolled for t time steps with evidence entered for each time step and calculating a query over the interface. To perform filtering for $t + 1$, LDJT uses LJT to distribute the information contained in α_t, which accounts for all evidence until time step t, and the entered evidence for time step $t + 1$ in J_{t+1} during the *inbound* and *outbound* phase of message passing. Hence, all parclusters of J_{t+1} receive information accounting for all evidence until time step $t+1$. Therefore, LDJT can use J_{t+1} to perform filtering for $t+1$ and prediction can be reformulated as filtering without new evidence added.

4 Evaluation

For different maximum time steps and domain sizes, we evaluate the largest (par)cluster and the number of (par)clusters in the (FO) jtrees for LDJT, LJT based on an unrolled model, and a ground interface approach, named JT. The number of (par)clusters n in an (FO) jtree determines the number of messages

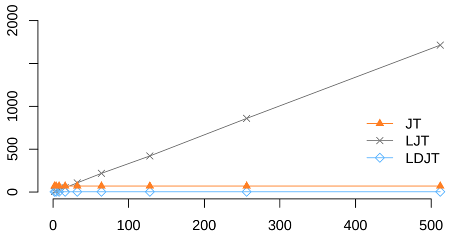


Fig. 10. (Par)clusters (y-axis) for different time steps (x-axis)

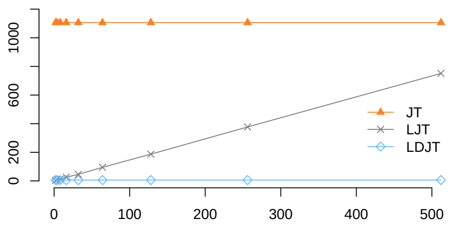


Fig. 11. (P)RVs in largest (par)cluster (y-axis) for different time steps (x-axis)

calculated during message passing. In general, message passing consists of calculating $2 \cdot (n-1)$ messages. Given new evidence, e.g., for a new time step, we need to calculate and parse new messages. The number of (P)RVs m in the largest (par)cluster, indicates how many variables we need to sum out, namely in the worst case we need to sum out $m-1$ (P)RVs for each message. Further, the largest (par)cluster and the number of (par)clusters can also be used to determine the complexity of VE and LVE a priori.

For G^{ex} Fig. 10 depicts the number of (par)clusters and Fig. 11 shows the number of (P)RVs in the largest (par)cluster for LDJT, LJT, and JT, with a domain size $\mathcal{D}(X) = \mathcal{D}(X') = 32$, for different maximum time steps. In these figures, we see that with increasing maximum time steps, the size of the FO jtree and the number of PRVs in the largest parcluster increase for LJT, while they remain constant for LDJT and JT. With increasing time steps, the unrolled model becomes larger. Therefore, the size input model increases with the time steps for LJT, while the input model remains constant for LDJT and JT.

For different domain sizes, Fig. 12 shows the number of (par)clusters and Fig. 13 depicts the number of (P)RVs in the largest (par)cluster for LDJT, LJT for 32 time steps, and JT. In these figures, we can see that with increasing domain sizes, the size of the jtree and the number of RVs in the largest cluster increase for JT, while they remain constant for LDJT and LJT. Due to more groundings, the input model increases with the domain size for JT. LDJT and LJT combine the instances and handle them as one. Therefore, they remain constant.

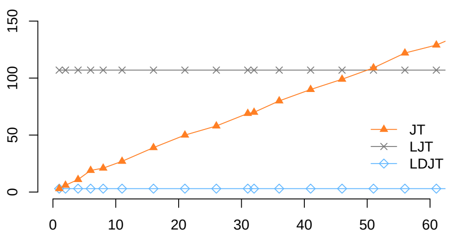


Fig. 12. (Par)clusters (y-axis) for different domain sizes (x-axis)

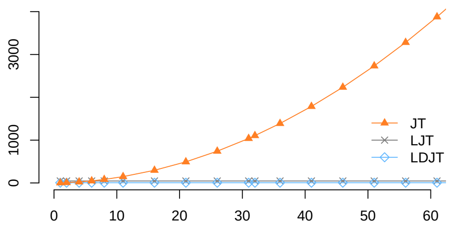


Fig. 13. (P)RVs in largest (par)cluster (y-axis) for different domain size (x-axis)

Having the numbers for different time steps and domain sizes for LDJT, LJT, and JT, let us now identify the calculations LDJT performs for each time step to compare the calculations against LJT and JT. During message passing, LDJT computes $2 \cdot (3 - 1) = 4$ messages. For each message, LDJT sums out at most $6 - 1 = 5$ PRVs. Actually, LDJT sums out 1 PRV for each *inbound* message and 3 and 4 PRVs for the *outbound* messages. Additionally, for each time step LDJT needs to calculate an α_t message, for which LDJT needs to sum out all non-interface PRVs from the *out-cluster* parcluster. The additional efforts are similar to answering one additional query. In our case, the *out-cluster* parcluster contains 6 and the interface 3 PRVs. Therefore, LDJT sums out 3 PRVs to calculate an α_t message and overall computes 5 messages for each time step.

For a domain size of 32, JT has 70 clusters with maximal 1106 RVs in each of them. Hence, for every time step JT computes 138 messages and for every message needs to sum out 1105 RVs in the worst case. For the PDM unrolled for 64 time steps, LJT has 218 paclusters with at most 95 PRVs. Thus, LJT computes 434 messages and for every message needs to sum out 94 PRVs in the worst case. For 64 time steps, LDJT computes 320 messages, each with only a small friction of summing out operations. Further, only in case all evidence is known before for all 64 time steps, LJT computes 434 messages to be able to answer queries for all time steps. In case evidence is provided incrementally, LJT needs to perform message passing for each new evidence, which can result in 27776 messages. Therefore, accounting for the temporal and relational aspects of the model in LDJT significantly reduces the number of computations.

5 Conclusion

We present LDJT, a filtering and prediction algorithm for relational temporal models. LDJT answers multiple queries efficiently by reusing a compact FO jtree structure for multiple queries. Further, due to temporal m-separation, which is ensured by the *in-* and *out-clusters*, LDJT uses the same compact structure for all time steps $t > 0$. Furthermore, LDJT does not need to know the maximum number of time steps and allows for efficiently adding time steps dynamically. First results show that the number of computations LDJT saves compared to LJT and JT is significant.

We currently work on extending LDJT to also perform smoothing. Smoothing could also be helpful to deal with incrementally changing models. Other interesting future work includes a tailored automatic learning for PDMs, parallelisation as well as using local symmetries. Additionally, it would be interesting to include our work in PDBs, e.g., to handle correlated PDBs [8].

References

1. Ahmadi, B., Kersting, K., Mladenov, M., Natarajan, S.: Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training. Machine learning 92(1), 91–132 (2013)

2. Braun, T., Möller, R.: Lifted Junction Tree Algorithm. In: Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz). pp. 30–42. Springer (2016)
3. Braun, T., Möller, R.: Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm. In: Graph Structures for Knowledge Representation and Reasoning - 5th International Workshop (2017)
4. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009)
5. Dignös, A., Böhlen, M.H., Gamper, J.: Temporal Alignment. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. pp. 433–444. ACM (2012)
6. Dylla, M., Miliaraki, I., Theobald, M.: A Temporal-Probabilistic Database Model for Information Extraction. Proceedings of the VLDB Endowment 6(14), 1810–1821 (2013)
7. Geier, T., Biundo, S.: Approximate Online Inference for Dynamic Markov Logic Networks. In: Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence (ICTAI). pp. 764–768. IEEE (2011)
8. Kanagal, B., Deshpande, A.: Lineage Processing over Correlated Probabilistic Databases. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 675–686. ACM (2010)
9. Lauritzen, S.L., Spiegelhalter, D.J.: Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems. Journal of the Royal Statistical Society. Series B (Methodological) pp. 157–224 (1988)
10. Manfredotti, C.E.: Modeling and Inference with Relational Dynamic Bayesian Networks. Ph.D. thesis, Ph. D. Dissertation, University of Milano-Bicocca (2009)
11. Milch, B., Zettlemoyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted Probabilistic Inference with Counting Formulas. In: Proceedings of AAAI. vol. 8, pp. 1062–1068 (2008)
12. Murphy, K., Weiss, Y.: The Factored Frontier Algorithm for Approximate Inference in DBNs. In: Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence. pp. 378–385. Morgan Kaufmann Publishers Inc. (2001)
13. Murphy, K.P.: Dynamic Bayesian Networks: Representation, Inference and Learning. Ph.D. thesis, University of California, Berkeley (2002)
14. Nitti, D., De Laet, T., De Raedt, L.: A particle Filter for Hybrid Relational Domains. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). pp. 2764–2771. IEEE (2013)
15. Papai, T., Kautz, H., Stefankovic, D.: Slice Normalized Dynamic Markov Logic Networks. In: Proceedings of the Advances in Neural Information Processing Systems. pp. 1907–1915 (2012)
16. Poole, D.: First-order probabilistic inference. In: Proceedings of IJCAI. vol. 3, pp. 985–991 (2003)
17. Richardson, M., Domingos, P.: Markov Logic Networks. Machine learning 62(1), 107–136 (2006)
18. de Salvo Braz, R.: Lifted First-Order Probabilistic Inference. Ph.D. thesis, Ph. D. Dissertation, University of Illinois at Urbana Champaign (2007)
19. Taghipour, N., Davis, J., Blockeel, H.: First-order Decomposition Trees. In: Proceedings of the Advances in Neural Information Processing Systems. pp. 1052–1060 (2013)
20. Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. Journal of Artificial Intelligence Research 47(1), 393–439 (2013)

21. Thon, I., Landwehr, N., De Raedt, L.: Stochastic relational processes: Efficient inference and applications. Machine Learning 82(2), 239–272 (2011)
22. Vlasselaer, J., Van den Broeck, G., Kimmig, A., Meert, W., De Raedt, L.: TP-Compilation for Inference in Probabilistic Logic Programs. International Journal of Approximate Reasoning 78, 15–32 (2016)
23. Vlasselaer, J., Meert, W., Van den Broeck, G., De Raedt, L.: Efficient Probabilistic Inference for Dynamic Relational Models. In: Proceedings of the 13th AAAI Conference on Statistical Relational AI. pp. 131–132. AAAIWS'14-13, AAAI Press (2014)