Relational Forward Backward Algorithm for Multiple Queries

Marcel Gehrke, Tanya Braun, and Ralf Möller
Institute of Information Systems, University of Lübeck, Lübeck
{gehrke, braun, moeller}@ifis.uni-luebeck.de

Abstract

The lifted dynamic junction tree algorithm (LDJT) efficiently answers *filtering* and *prediction* queries for probabilistic relational temporal models by building and then reusing a first-order cluster representation of a knowledge base for multiple queries and time steps. Specifically, this paper contributes (i) a relational forward backward algorithm with LDJT, (ii) *smoothing* for hindsight queries, and (iii) different approaches to instantiate a first-order cluster representation during a backward pass. Further, our relational forward backward algorithm makes hindsight queries with huge lags feasible. LDJT answers multiple temporal queries faster than the static lifted junction tree algorithm on an unrolled model, which performs *smoothing* during message passing.

1 Introduction

Areas like healthcare or logistics and cross-sectional aspects such as IT security involve probabilistic data with relational and temporal aspects and need efficient exact inference algorithms. These domains involve many objects in relation to each other with changes over time and uncertainties about object existence, attribute value assignments, or relations between objects. More specifically, IT security involves network dependencies (relational) for many components (objects), streams of attacks over time (temporal), and uncertainties. Therefore, in this paper, we study the problem of exact inference in relational temporal probabilistic models.

First-order probabilistic inference leverages the relational aspect of a static model. For models with known domain size, it exploits symmetries in a relational static model by combining instances to reason with representatives, known as lifting (Poole 2003). Poole (2003) introduces parametric factor graphs as relational models and proposes lifted variable elimination (LVE) as an exact inference algorithm on relational models. Taghipour et al. (2013), besides others, extend LVE to its current form. To benefit from the ideas of the junction tree algorithm (Lauritzen and Spiegelhalter 1988) and LVE, Braun and Möller (2018) present the lifted junction tree algorithm (LJT) that efficiently performs exact lifted inference on relational models given a set of queries.

Now, we take a look at approches that perform inference at discrete time steps on relational temporal models.

Most approaches for relational temporal models perform approximate inference. Additionally to being approximate, these approaches involve unnecessary groundings or are not designed to handle multiple queries efficiently. Ahmadi et al. (2013) propose lifted belief propagation for dynamic Markov logic networks (DMLNs). Geier and Biundo (2011) and Papai, Kautz, and Stefankovic (2012) present online inference algorithms for DMLNs. Vlasselaer et al. (2016) introduce an exact approach for relational temporal models, but perform inference on a ground knowledge base.

We (2018a) present parameterised probabilistic dynamic models (PDMs) to represent probabilistic relational temporal behaviour and propose the lifted dynamic junction tree algorithm (LDJT) to efficiently answer multiple *filtering* and *prediction* queries exactly. LDJT combines the advantages of the interface algorithm (Murphy 2002) and LJT (Braun and Möller 2018). This paper extends LDJT and specifically contributes (i) a relational forward backward algorithm by introducing an *inter* first-order junction tree (FO jtree) backward pass, (ii) performing *smoothing* for hindsight queries, and (iii) different FO jtree instantiation approaches during a backward pass.

Even though a backward pass is crucial for many AI problems and *smoothing* is a key inference problem, to the best of our knowledge there is no approach with a backward pass which solves *smoothing* efficiently for relational temporal models. Additionally, a backward pass is required for problems such as learning. Therefore, we extend LDJT, leveraging the well-studied LVE and LJT algorithms. Further, our relational forward backward algorithm allows for instantiating FO jtrees by leveraging LDJT's forward pass, which makes hindsight queries with huge lags feasible.

In the following, we begin by recapitulating PDMs and present LDJT, an efficient reasoning algorithm for PDMs. Afterwards, we extend LDJT with an *inter* FO jtree backward pass and discuss different approaches to instantiate an FO jtree during a backward pass. Lastly, we evaluate LDJT against LJT and conclude by looking at extensions.

2 Parameterised Probabilistic Models

We shortly present parameterised probabilistic models (PMs) for relational static models (Braun and Möller 2018) and extend PMs to the temporal case, resulting in PDMs (Gehrke, Braun, and Möller 2018a).

2.1 Parameterised Probabilistic Models

PMs combine first-order logic with probabilistic models, representing first-order constructs using logical variables (logvars) as parameters. As an example, we set up a PM to model if a server is compromised. We use logvars to represent users with certain privileges and model different attacks on different user groups, which can also infect each other. A boolean random variable (randvar) holds if it is infected.

Definition 1. Let $\mathbf{L}$ be a set of logvar names, Φ a set of factor names, and $\mathbf{R}$ a set of 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 logvars $X^1, ..., X^n \in \mathbf{L}$. 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 for restricting 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 $gr(Y|C)$ denotes the set of instances of Y with all logvars in Y grounded w.r.t. constraint C.

From $\mathbf{R} = \{IA, IU\}$, for infected admin respectively user, and $\mathbf{L} = \{X, Y\}$ with $\mathcal{D}(X) = \{x_1, x_2, x_3\}$ and $\mathcal{D}(Y) = \{y_1, y_2\}$, we build the boolean PRVs $IA(Y)$ and $IU(X)$. With $C = (X, \{x_1, x_2\})$, $gr(IU(X)|C) = \{IU(x_1), IU(x_2)\}$. $gr(IU(X)|\top)$ also contains $IU(x_3)$.

Definition 2. We denote a parametric factor (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, C a constraint on $\mathbf{X}$, and $\mathcal{A} = (A^1, ..., A^n)$ a sequence of 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 identical for all grounded instances of $\mathcal{A}$. The complete specification for ϕ is a list of all input-output values. A PM $G := \{g^i\}_{i=0}^{n-1}$ is a set of parfactors and semantically represents the full joint probability distribution $P_G = \frac{1}{Z} \prod_{f \in gr(G)} f$ with Z as normalisation constant.

Adding boolean PRVs $UserAttack$, $AdminAttack$, IS, for infected server, and $Infects(X, Y)$, we build the PM $G_{ex} = \{g^i\}_{i=0}^4$, with $g^0 = \phi^0(UserAttack, IU(X))|\top$, $g^1 = \phi^1(AdminAttack, IA(Y))|\top$, $g^2 = \phi^2(IU(X), IA(Y), Infects(X, Y))|\top$, $g^3 = \phi^3(IS, IU(X))|\top$, and $g^4 = \phi^4(IS, IA(Y))|\top$. g^2 has eight, the others four input-output pairs (omitted). Constraints are $\top$, i.e., the ϕ's are defined for all domain values. Figure 1 depicts G^{ex} as a parfactor graph.

The semantics of a model is given by grounding and building a full joint distribution. In general, a query asks 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 query term Q (ground PRV), and events $\mathbf{E} = \{E^i = e^i\}_i$ (ground PRVs with fixed range values), the expression $P(Q|\mathbf{E})$ denotes a *query* w.r.t. P_G.

2.2 Parameterised Probabilistic Dynamic Models

We define PDMs based on the first-order Markov assumption. Further, the underlying process is stationary.
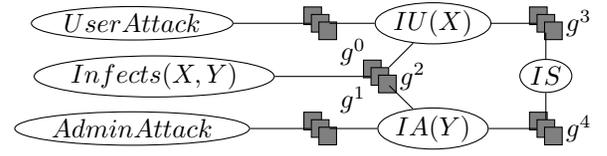

Figure 1: Parfactor graph for G^{ex}

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 representing $\mathbf{A}_{t-1}$ and $\mathbf{A}_t$ where $\mathbf{A}_\pi$ is a set of PRVs from time slice π.

Figure 2 shows $G^{ex}_\rightarrow$ consisting of G^{ex} for time step $t - 1$ and t with *inter*-slice parfactors for the behaviour over time. In this example, the parfactors g^A and g^U are the *inter*-slice parfactors, modelling the temporal behavior.

Definition 5. Given a PDM G, a query term Q (ground PRV), and events $\mathbf{E}_{0:t} = \{E^i_t = e^i_t\}_{i,t}$ (ground PRVs with fixed range values), $P(Q_t|\mathbf{E}_{0:t})$ denotes a *query* w.r.t. P_G.

The problem of answering a marginal distribution query $P(A^i_\pi|\mathbf{E}_{0:t})$ w.r.t. the model is called *prediction* for $\pi > t$, *filtering* for $\pi = t$, and *smoothing* for $\pi < t$.

3 Lifted Dynamic Junction Tree Algorithm

We recapitulate LJT (Braun and Möller 2018) to answer queries for PMs and LDJT (Gehrke, Braun, and Möller 2018a) to answer *filtering* and *prediction* queries for PDMs.

3.1 Lifted Junction Tree Algorithm

LJT provides efficient means to answer queries $P(Q^i|\mathbf{E})$, with $Q^i \in \mathbf{Q}$ a set of query terms, 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 $Q^i \in \mathbf{Q}$.

We first define an FO jtree, with parameterised clusters (parclusters) as nodes, and then go through each step.

Definition 6. A parcluster $\mathbf{C}$ is defined by $\forall \mathbf{L} : \mathbf{A}|C$. $\mathbf{L}$ is a set of logvars, $\mathbf{A}$ 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) $C^\phi \subseteq C$ holds. We call the set of assigned parfactors a local model G^i.

An FO jtree for a PM G is $J = (\mathbf{V}, \mathbf{P})$ where J is a cycle-free graph, the nodes $\mathbf{V}$ denote a set of parclusters, and $\mathbf{P}$ is a set of edges between parclusters. An FO jtree must satisfy the 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 $\mathbf{S}^{ij}$ of edge $i - j$ is given by $\mathbf{C}^i \cap \mathbf{C}^j$ containing shared PRVs.

LJT constructs an FO jtree using a first-order decomposition tree, enters evidence in the FO jtree, and passes messages through an *inbound* and an *outbound* pass. To compute a message, LJT eliminates all non-separator PRVs from the
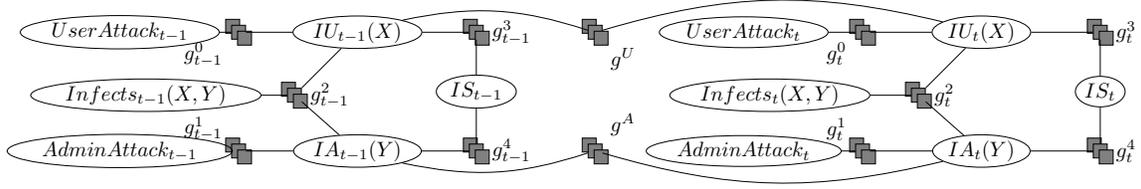
Figure 2: $G_\rightarrow^{ex}$ the two-slice temporal parfactor graph for model G^{ex}

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.

Figure 3 shows an FO jtree of G^{ex} with the local models of the parclusters and the separators as labels of edges. During the *inbound* phase of message passing, LJT sends messages from $\mathbf{C}^1$ and $\mathbf{C}^3$ to $\mathbf{C}^2$ and during the *outbound* phase from $\mathbf{C}^2$ to $\mathbf{C}^1$ and $\mathbf{C}^3$. If we want to know whether $UserAttack$ holds, we query for $P(UserAttack)$ for which LJT can use parcluster $\mathbf{C}^1$. LJT sums out $IU(X)$ from $\mathbf{C}^1$'s local model G^1, $\{g^0\}$, combined with the received messages, here, one message from $\mathbf{C}^2$.

3.2 LDJT: Overview

LDJT efficiently answers queries $P(Q_\pi^i|\mathbf{E}_{0:t})$, with $Q_\pi^i \in \mathbf{Q_t}$ and $\mathbf{Q_t} \in \{\mathbf{Q}_t\}_{t=0}^T$, given a PDM G and evidence $\{\mathbf{E}_t\}_{t=0}^T$, by performing the following steps: (i) Construct two FO jtrees J_0 and J_t with *in-* and *out-clusters* from G. (ii) For $t = 0$, using J_0 to enter $\mathbf{E}_0$, pass messages, answer each query term $Q_\pi^i \in \mathbf{Q}_0$, and preserve the state in message α_0. (iii) For $t > 0$, instantiate J_t for the current time step t, recover the previous state from message α_{t-1}, enter $\mathbf{E}_t$ in J_t, pass messages, answer each query term $Q_\pi^i \in \mathbf{Q}_t$, and preserve the state in message α_t.

We begin with LDJT's FO jtree construction. The FO jtrees contain a minimal set of PRVs to m-separate the FO jtrees. M-separation means that information about these PRVs renders FO jtrees independent from each other. Afterwards, we present how LDJT connects FO jtrees for reasoning to solve the *filtering* and *prediction* problems efficiently.

3.3 LDJT: FO Jtree Construction for PDMs

LDJT constructs FO jtrees for G_0 and $G_\rightarrow$, both with an incoming and outgoing interface. To be able to construct the interfaces in the FO jtrees, LDJT uses a PDM G to identify the interface PRVs $\mathbf{I}_t$ for a time slice t.

Definition 7. The forward interface is defined as
$\mathbf{I}_{t-1} = \{A_{t-1}^i \mid \exists \phi(\mathcal{A})|C \in G : A_{t-1}^i \in \mathcal{A} \land \exists A_t^j \in \mathcal{A}\}$.
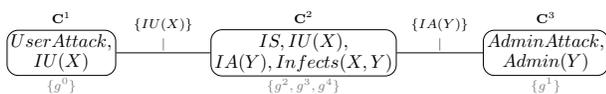

Figure 3: FO jtree for G^{ex} (local models as labels)

PRVs $User_{t-1}(X)$ and $Admin_{t-1}(Y)$ from $G_\rightarrow^{ex}$, shown in Fig. 2, have successors in the next time slice, making up $\mathbf{I}_{t-1}$. To ensure interface PRVs $\mathbf{I}$ ending up in a single parcluster, LDJT adds a parfactor g^I over the interface to the model. Thus, LDJT adds a parfactor g_0^I over $\mathbf{I}_0$ to G_0, builds an FO jtree J_0, and labels the parcluster with g_0^I from J_0 as *in-* and *out-cluster*. For $G_\rightarrow$, LDJT removes all non-interface PRVs from time slice $t - 1$, adds parfactors g_{t-1}^I and g_t^I, constructs J_t, and labels the parcluster containing g_{t-1}^I as *in-cluster* and the parcluster containing g_t^I as *out-cluster*.

The interface PRVs are a minimal required set to m-separate the FO jtrees. LDJT uses these PRVs as separator to connect the *out-cluster* of J_{t-1} with the *in-cluster* of J_t, allowing to reuse the structure of J_t for all $t > 0$.

3.4 LDJT: Reasoning with PDMs

Since J_0 and J_t are static, LDJT uses LJT as a subroutine by passing on a constructed FO jtree, queries, and evidence for step t to handle evidence entering, message passing, and query answering using the FO jtree. Further, for proceeding to the next time step, LDJT calculates an α_t message over the interface PRVs using the *out-cluster* to preserve the information about the current state. Afterwards, LDJT increases t by one, instantiates J_t, and adds α_{t-1} to the *in-cluster* of J_t. During the *inbound* and *outbound* message passing, α_{t-1} is distributed through J_t. Thereby, LDJT performs an *inter* FO jtree forward pass to proceed in time.

Figure 4 depicts the passing on of the current state from time step three to four. To capture the state at $t = 3$, LDJT sums out the non-interface PRVs IS_3 and $IA_2(Y)$ from $\mathbf{C}_3^3$'s local model and the received messages and saves the result in message α_3. After increasing t by one, LDJT adds α_3 to $\mathbf{C}_4^2$, the *in-cluster* of J_4. α_3 is then distributed by message passing and accounted for during calculating α_4.

4 Relational Forward Backward Algorithm

We begin by introducing a relational forward backward pass. Further, we propose instantiation approaches for a backward pass and combine them to answer *hindsight* queries.

4.1 Relational Forward Backward Algorithm

Using the forward pass, each FO jtree contains evidence from the initial time step up to its time step. The *inter* FO jtree backward pass propagates information to previous time steps to answer queries $P(A_\pi^i|\mathbf{E}_{0:t})$ with $\pi < t$.

To perform a backward pass, LDJT uses the *in-cluster* of the FO jtree J_t to calculate a β_t message over the interface
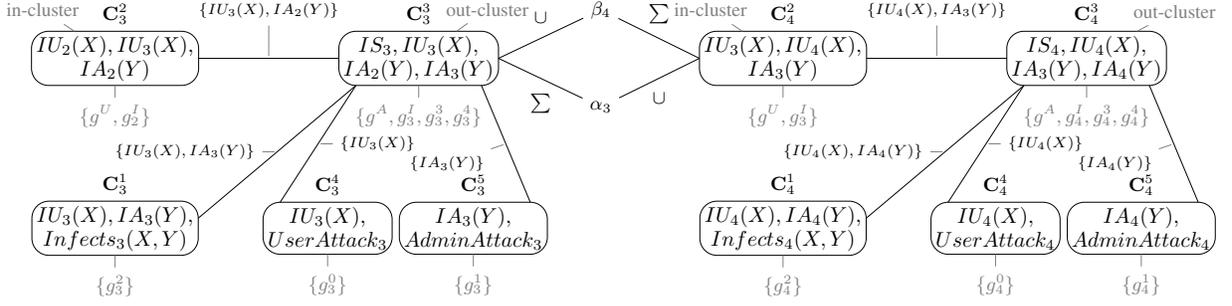
Figure 4: Forward and backward pass of LDJT (local models and in- and out-cluster labeling in grey)

PRVs and sends β_t to the *out-cluster* of J_{t-1}. Therefore, LDJT has to ignore the α_{t-1} message, received from the *out-cluster* of J_{t-1}. After LDJT calculates β_t by summing out all non-interface PRVs, it decreases t by one. Finally, LDJT instantiates the J_{t-1} for the new time step $t-1$ with the β_t message in the *out-cluster*. Figure 4 also depicts LDJT's backward pass. LDJT uses the *in-cluster* of J_4 to calculate β_4. After decreasing t by one, LDJT adds β_4 to the *out-cluster* of J_3. β_4 is then distributed and accounted for in β_3.

Algorithm 1 outlines the relational forward backward algorithm. LDJT constructs FO jtrees J_0 and J_t and the set of interface PRVs using *DFO-JTREE* as described in Section 3.3. Afterwards, LDJT answers all queries by performing a routine of entering evidence, message passing, query answering for the current time step, and proceeding in time.

For query answering, LDJT identifies the query type, namely *filtering*, *prediction*, and *hindsight*. To perform *filtering*, LDJT passes the query and the current FO jtree to LJT to answer the query. LDJT applies the forward pass until the time step of the query is reached to answer the query for *prediction* queries. To answer *hindsight* queries, LDJT applies the backward pass until the time step of the query is reached and answers the query. Further, LDJT uses LJT for message passing to account for α and β messages.

Let us now illustrate how LDJT answers *hindsight* queries, which is only possible due to the relational forward backward algorithm. Assuming the server is compromised at time step 1983, we would like to know whether $IA(y_1)$ infected $IU(x_1)$ at time step 1973 and whether $IU(x_1)$ holds at timestep 1978. LDJT answers the marginal distribution queries $P(\mathbf{Q}_{1983}|\mathbf{E}_{0:1983})$, with $\mathbf{E}_{1983}$ consisting of $\{IS_{1983} = true\}$ and the set of query terms $\mathbf{Q}_{1983}$ consisting of at least $\{IU_{1978}(x_1), Infects_{1973}(x_1, y_1)\}$.

LDJT enters the evidence $\{IS_{1983} = true\}$ in J_{1983} and passes messages. To answer the queries, LDJT performs a backward pass and first calculates β_{1983} by summing out $IU_{1983}(X)$ from $\mathbf{C}^2_{1983}$'s local model and received messages without α_{1982}. LDJT adds the β_{1983} message to $\mathbf{C}^3_{1982}$'s local model and passes messages in J_{1982} using LJT. In such a manner LDJT proceeds until it reaches time step 1978 and thus propagated the information to J_{1978}.

Having J_{1978}, LDJT answers the marginal distribution query $P(IU_{1978}(x_1)|\mathbf{E}_{0:1983})$. To answer the query, LJT sums out $UserAttack_{1978}$ and $IU_{1978}(X)$ where $X \neq x_1$

from $\mathbf{C}^4_{1978}$'s local model and the received message from $\mathbf{C}^3_{1978}$. To answer the other marginal distribution query $P(Infects_{1973}(x_1, y_1)|\mathbf{E}_{0:1983})$, LDJT performs additional backward passes until it reaches time step 1973 and then uses LJT to answer $P(Infects_{1973}(x_1, y_1)|\mathbf{E}_{0:1983})$.

Theorem 1. *LDJT's backward pass is correct.*

Proof. Each FO jtree contains evidence up to the time step the FO jtree is instantiated for. During a backward pass, LDJT distributes information, including evidence, from the current FO jtree J_t backwards. Therefore, LDJT performs an *inter* FO jtrees backward message pass over the interface separator. The β_t message is correct, since calculating the β_t message, the *in-cluster* received all messages from its neighbours and ignores the α_{t-1} message, which originated from the designated receiver. The β_t message, which LDJT adds to the *out-cluster* of J_{t-1}, is then accounted for during the message pass inside J_{t-1}, as well as the α_{t-1} message. Following this approach, every FO jtree included in the backward pass contains all information, as the α message encodes all past information and the β message encodes all information from the future. Thus, it suffices to apply the backward pass until LDJT reaches the desired time step and does not need to apply the backward pass until $t = 0$. $\square$

The forward and backward pass instantiate FO jtrees from the corresponding structure given a time step. However, since LDJT already instantiates FO jtrees during a forward pass, it has different approaches to instantiate FO jtrees during a backward pass. The first approach is to preserve all instantiated FO jtrees from the forward pass. The second approach is to instantiate FO jtrees on-demand using evidence and α messages, which is only possible by leveraging the m-separation of the FO jtrees and the forward pass.

Preserving FO Jtree Instantiations Preserving all instantiated FO jtrees, including computed messages, is time-efficient since the approach reuses already performed computations. Thereby, during an *intra* FO jtree message pass, LDJT only needs to account for the β message and not also the α message. The main drawback is the memory consumption, as each FO jtree needs to be stored.

On-Demand FO Jtree Instantiation To instantiate an FO jtree, LDJT enters evidence, α and β messages, and repeats a complete message pass. Instantiating FO jtrees using

Algorithm 1 LDJT Alg. for PDM $(G_0, G_\rightarrow)$, Queries $\{\mathbf{Q}\}_{t=0}^T$, Evidence $\{\mathbf{E}\}_{t=0}^T$

> **procedure** LDJT$(G_0, G_\rightarrow, \{\mathbf{Q}\}_{t=0}^T, \{\mathbf{E}\}_{t=0}^T)$
>> $t := 0$
>> $(J_0, J_t, \mathbf{I}_t) := $ DFO-JTREE$(G_0, G_\rightarrow)$
>> **while** $t \neq T + 1$ **do**
>>> $J_t := $ LJT.EnterEvidence$(J_t, \mathbf{E}_t)$
>>> $J_t := $ LJT.PassMessages(J_t)
>>> **for** $q_\pi \in \mathbf{Q}_t$ **do**
>>>> AnswerQuery$(J_0, J_t, q_\pi, \mathbf{I}_t, \alpha, t)$
>>> $(J_t, t, \alpha[t-1]) := $ ForwardPass$(J_0, J_t, t, \mathbf{I}_t)$

> **procedure** ANSWERQUERY$(J_0, J_t, q_\pi, \mathbf{I}_t, \alpha, t)$
>> **while** $t \neq \pi$ **do**
>>> **if** $t > \pi$ **then**
>>>> $(J_t, t) := $ BackwardPass$(J_0, J_t, \mathbf{I}_t, \alpha[t-1], t)$
>>> **else**
>>>> $(J_t, t, _) := $ ForwardPass$(J_0, J_t, \mathbf{I}_t, t)$
>> LJT.PassMessages(J_t)
>> **print** LJT.AnswerQuery(J_t, q_π)

> **function** FORWARDPASS$(J_0, J_t, \mathbf{I}_t, t)$
>> $\alpha_t := \sum_{J_t(\text{out-cluster}) \setminus \mathbf{I}_t} J_t(\text{out-cluster})$
>> $t := t + 1$
>> $J_t(\text{in-cluster}) := \alpha_{t-1} \cup J_t(\text{in-cluster})$
>> **return** (J_t, t, α_{t-1})

> **function** BACKWARDPASS$(J_0, J_t, \mathbf{I}_t, \alpha_{t-1}, t)$
>> $\beta_t := \sum_{J_t(\text{in-cluster}) \setminus \mathbf{I}_t} (J_t(\text{in-cluster}) \setminus \alpha_{t-1})$
>> $t := t - 1$
>> $J_t(\text{out-cluster}) := \beta_{t+1} \cup J_t(\text{out-cluster})$
>> **return** (J_t, t)

---

evidence and α messages, by leveraging the m-separation of FO jtrees, is space efficient. The main drawback are repeated computations, due to a complete message pass.

4.2 Discussion

We discuss how LDJT efficiently combines the instantiation approaches and show how LDJT reuses calculations for multiple *hindsight* and *prediction* queries from one time step.

Combining Instantiation Approaches   Having reoccurring *hindsight* queries, LDJT knows the maximum lag, for which it has to perform backward passes, also called fixed lag *smoothing*. With a known fixed lag, a combination of our two approaches is advantageous. Assuming the fixed *smoothing* lag is 10, LDJT can preserve the last 10 FO jtrees and instantiate additional FO jtrees on-demand. If an on-demand *hindsight* query has a lag of 20, LDJT instantiates the FO jtrees starting with J_{t-11}. Thereby, LDJT preserves a certain number of FO jtrees instantiated for fast query answering, and in case a *hindsight* query is even further in the past, reconstructs FO jtrees using evidence and α messages.

Further, LDJT cannot keep all FO jtrees in memory due to

the memory consumption for a huge number of time steps. Thus, by only storing the α messages, which is a fraction compared to all messages and parfactors for each time step, LDJT preserves the required information to perform hindsight queries even for the first time step. Hence, combining the approaches is a necessary compromise between time and space efficiency.

Reusing Computations for One Time Step   LDJT also reuses computations to answer multiple queries for one time step. For example, a robot with a stream of location data would like to know where he was 2 and 4 time steps ago. Here, LDJT reuses the calculations performed during the *hindsight* query with a lag of 2, namely it starts the backward pass for the query with lag 4 at J_{t-2}.

To reuse computations, the first option is that the *hindsight* queries are sorted based on the time difference to the current time step. Here, LDJT preserves the FO jtree from the last *hindsight* query and performs additional backward passes. The second option is to preserve the calculated β messages for the current time step and instantiate the FO jtree closest to the currently queried time step. Analogously, LDJT reuses computations for *prediction* queries. Under the presence of *prediction* queries, LDJT uses the computed α_t to proceed to the next time step. However, based on the problem, given new evidence for a new time step all other α and β messages that LDJT calculated for the previous time step are invalid.

5   Evaluation

Now, we show that LDJT can answer queries efficiently, namely in linear time, even if no FO jtrees are preserved. For the evaluation, we use the example model G^{ex} with the set of evidence being empty. To answer queries, LDJT performs on-demand FO jtree instantiation. Additionally, we compare the runtime for multiple maximum time steps against LJT provided with the unrolled model and unrolled FO jtree. We do not compare LDJT against the ground case, as from a runtime complexity perspective of LDJT, there is no difference in either performing *smoothing* with lag 10 or *prediction* with 10 time steps into the future. Thus, the previous results that LDJT outperforms the ground case also hold for the relational forward backward algorithm.

We define our set of queries for each time step as: $\{IS_t, IU_t(x_1), IA_t(y_1)\}$ with lag 0, 2, 5, and 10. Thus, for each time step, these 12 queries are executed, e.g., 1200 queries for 100 time steps. Now, we evaluate the runtimes of LDJT and LJT providing 16 GB of RAM using the set of queries.

Figure 5 shows the runtime in seconds in log scale for each maximum time step up to $10,000$ time steps. We can see that the runtime of LDJT (diamond) to answer the questions is linear to the maximum number of time steps. Thus, LDJT more or less needs a constant time to answer queries once it has instantiated the corresponding FO jtree. Additionally, the time to perform either a forward or backward pass is more or less constant. For LDJT, the runtimes for each operation are independent from the current time step, since the structure of the model stays the same over time. Providing the unrolled model to LJT (circle) yields results for the first 8 time steps with a reduced set of queries. The
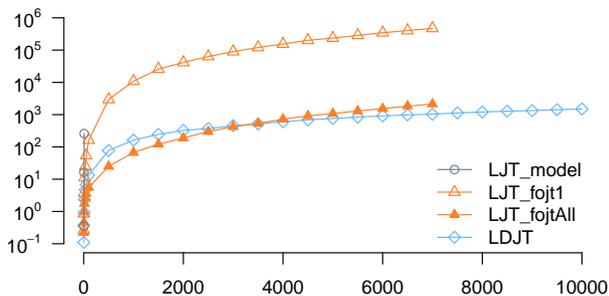
Figure 5: Runtimes [seconds], x-axis: maximum time steps

FO jtree construction of LJT is not optimised for the temporal case, such as creating an FO jtree similar to an unrolled version of LDJT's FO jtree. Therefore, the number of PRVs in a parcluster increases with additional time steps in LJT. In our example, the maximum number of PRVs in a parcluster for 4 time steps is 14 and for 8 time steps is 27 and the complexity of LVE is exponential to the maximum number of PRVs (Taghipour et al. 2013).

But even with an optimised FO jtree (filled and unfilled triangle), LJT still does not handle the temporal aspects efficiently and always performs a message pass on the unrolled FO jtree, which is comparable to performing a *hindsight* query for the first time step and a *prediction* query for the very last time step with LDJT. Thus, the message pass normally involves unnecessary computations. Even if LJT performs only one message pass (filled triangle), i.e., all information for all time steps is directly provided, the overhead of performing a complete message pass is evident compared to LDJT. Performing for each time step one message pass (unfilled triangle), to obtain the same results as LDJT, shows that handling the temporal aspects efficiently is orders of magnitudes faster. Additionally, LJT needs to store all information for each time step, which is not feasible if the maximum number of time steps is huge due to the memory consumption. In our small example LJT could only unroll and store the information for about 7400 FO jtrees. Hence, a combination of our instantiation approaches is advantageous and necessary to answer *hindsight* queries with a huge lag.

Overall, Fig. 5 shows (i) how crucial proper handling of temporal aspects is and (ii) that LDJT answers the queries with on-demand instantiation in linear time. Evidence is not included in the evaluation as symmetric evidence only increases the groups to reason over, while asymmetric evidence completely grounds the model over time. This evaluation shows that LDJT has a linear behaviour given a lifted solution is possible. In general, in case LDJT cannot compute a lifted solution, it is equivalent to ground algorithms. However, LDJT outperforms the ground case, independent of the complexity of the inter-slice parfactors, as long as a lifted solution is possible. Further, we (2018b) show that when lifting an algorithm, one also needs to ensure preconditions of lifting. LDJT prevents unnecessary groundings by ensuring preconditions of lifting while proceeding in time (Gehrke, Braun, and Möller 2018b) to obtain a lifted solution if possible.

6 Conclusion

We present the first relational forward backward algorithm, LDJT, to efficiently answer *hindsight*, *filtering*, and *prediction* queries for relational temporal models. LDJT answers multiple queries by reusing a compact FO jtree structure for multiple queries. The ensured temporal m-separation of FO jtrees allows for reusing computations and for reducing memory consumption, making a relational forward backward algorithm possible and answering *hindsight* queries with huge lags feasible. First results show that LDJT's runtime is linear to the number of time steps and significantly outperforms LJT. Overall, answering *hindsight*, *filtering*, and *prediction* queries is crucial for all kinds of AI problems, which becomes manageable in combination with lifting.

We currently work on calculating the most probable explanation as well as supporting decision making. Furthermore, we now can look into learning relational temporal models and how to preserve a lifted representation over time.

References

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

Braun, T., and Möller, R. 2018. Parameterised Queries and Lifted Query Answering. In *Proc. of IJCAI*, 4980–4986.

Gehrke, M.; Braun, T.; and Möller, R. 2018a. Lifted Dynamic Junction Tree Algorithm. In *Proc. of the 23rd Int. Conf. on Conceptual Structures*, 55–69. Springer.

Gehrke, M.; Braun, T.; and Möller, R. 2018b. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proc. of AI 2018*, 556–562. Springer.

Geier, T., and Biundo, S. 2011. Approximate Online Inference for Dynamic Markov Logic Networks. In *Proc. of the 23rd ICTAI*, 764–768. IEEE.

Lauritzen, S. L., and Spiegelhalter, D. J. 1988. Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems. *Journal of the Royal Statistical Society. Series B (Methodological)* 157–224.

Murphy, K. P. 2002. *Dynamic Bayesian Networks: Representation, Inference and Learning*. Ph.D. Dissertation, UCB.

Papai, T.; Kautz, H.; and Stefankovic, D. 2012. Slice Normalized Dynamic Markov Logic Networks. In *Proc. of NIPS*, 1907–1915.

Poole, D. 2003. First-order probabilistic inference. In *Proc. of IJCAI*, volume 3, 985–991.

Taghipour, N.; Fierens, D.; Davis, J.; and Blockeel, H. 2013. Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. *JAIR* 47(1):393–439.

Vlasselaer, J.; Van den Broeck, G.; Kimmig, A.; Meert, W.; and De Raedt, L. 2016. TP-Compilation for Inference in Probabilistic Logic Programs. *JAR* 78:15–32.