UNIVERSITÄT ZU LÜBECK
INSTITUTE OF INFORMATION SYSTEMS

From the Institute of Information Systems
of the University of Lübeck
Director: Prof. Dr. rer. nat. habil. Ralf Möller

Taming Exact Inference in Temporal Probabilistic Relational Models

Dissertation
for Fulfilment of
Requirements
for the Doctoral Degree
of the University of Lübeck

from the Department of Computer Sciences

Submitted by

Marcel Gehrke
from Hamburg

Lübeck 2019

First referee	Prof. Dr. Ralf Möller
Second referee	Prof. Dr. Maciej Liśkiewicz
Date of oral examination	December 17, 2021

Approved for printing.

Abstract

In information technology settings, multiple sensors gather data over time. With many sensors constantly sending data, manually analysing data is infeasible. Therefore, such data needs to be analysed automatically with reference to a model, to support automatic decision making or to provide recommendations for suitable decisions. One approach to construct a respective model is using machine learning and deriving a probabilistic model with random variables encoding the structure in data. In a model, for each known object, random variables are learned including the relations between the objects, leading to huge models. Probabilistic *relational* models compactly encoded a known number of individuals, and performing inference on probabilistic *relational* models allows for efficiently answering queries with a known number of individuals. However, standard (static) inference approaches for probabilistic relational models do not account for temporal aspects of gathered data. Existing approaches for temporal probabilistic relational models are approximate, such that any error can stem either from the model or the approximate inference approach. In this dissertation we study the problem of *exact repeated inference* in *temporal* probabilistic relational models. In particular, we present the lifted dynamic junction tree algorithm to remedy these challenges w.r.t. *temporal* inference.

The lifted dynamic junction tree algorithm is an *exact* algorithm to efficiently answering *multiple queries* in *temporal probabilistic relational models*. The algorithm leverages the strengths of the lifted junction tree algorithm and the interface algorithm. The lifted dynamic junction tree algorithm ensures preconditions of lifting while proceeding in time to form a fully lifted forward backward algorithm. We also extend the query language of the algorithm with conjunctive queries and assignment queries as well as maximum expected utility queries to support decision making. Further, we make contributions w.r.t. lifted evidence. On the one hand, we introduce uncertain evidence, i.e., (lifted) events combined with probabilities. On the other hand, we tame the effect of evidence on a lifted representation as evidence can ground a temporal relational model over time. Therefore, we present an algorithm for retaining an approximated lifted representation with a small and bounded error.

We empirically evaluate each contribution of this dissertation, thoroughly testing how the algorithms perform while increasing the number of instances and the maximum number of time steps. Further, for each part we provide an extensive theoretical analysis. We also investigate the connection to the lifted junction tree algorithm and the interface algorithm as well as connections between lifting and handling temporal aspects.

Kurzfassung

In Informationssystemen erheben eine Vielzahl von Sensoren Daten. Mit vielen Sensoren, die durchgehend Daten senden, ist ein manuelles Analysieren von Daten unmöglich. Um viele Daten analysieren zu können, muss diese Analyse automatisch mit Bezug auf ein Modell erfolgen. Das Ergebnis der Analyse könnte eine Entscheidung sein, welche das System direkt umsetzt oder die Ausgabe einer Empfehlung, was eine vernünftige Entscheidung wäre. Eine Möglichkeit, ein entsprechendes Modell zu konstruieren ist, maschinelles Lernen zu benutzen und ein probabilistisches Modell herzuleiten, welches mittels Zufallsvariablen die Struktur der Daten widerspiegelt. In einem Modell werden Zufallsvariablen sowohl für eine bekannte Anzahl von Individuen, welche von den Sensoren beobachtet werden, als auch für die Beziehungen zwischen den Individuen gelernt. Effizientes beantworten von Anfragen in probabilistischen Modellen mit einer bekannten Anzahl von Individuen wird durch Inferenz auf probabilistische relationale Modellen ermöglicht. Bestehende (statische) Inferenzverfahren für probabilistische relationale Modelle behandeln jedoch zeitliche Aspekte der gesammelten Daten nicht effizient. Bestehende Inferenzverfahren für temporale probabilistische relationalen Modellen sind approximativ, so dass ein möglicher Fehler entweder durch Approximationen beim Lernen des Modells entstanden sein können oder durch ein Approximieren bei einer Schlussfolgerung entstanden sein können. In dieser Dissertation untersuchen wir das Problem der *exakten* wiederholten Inferenz in *temporalen* probabilistischen relationalen Modellen um die aufgezeigten Probleme zu beheben. Insbesondere präsentieren wir hierfür den Lifted Dynamic Junction Tree Algorithmus.

Der Lifted Dynamic Junction Tree Algorithmus ist ein *exakter* Algorithmus, der verwendet werden kann, um eine *Vielzahl von Anfragen* für *temporale* probabilistische relationale Modelle effizient zu beantworten. Der Algorithmus nutzt die Stärken des Lifted Junction Tree Algorithmus und des Interface Algorithmus zu seinem Vorteil. Der Lifted Dynamic Junction Tree Algorithmus stellt während des Übergangs von einem Zeitschritt zum nächsten Vorbedingungen des Liftings sicher, um einen gelifteten Vorwärts-Rückwärts-Algorithmus zu formen, der, wenn das Modell es zulässt, nicht groundet. Außerdem erweitern wir die Anfragesprache des Algorithmus um konjunktive Anfragen und Zuweisungsanfragen sowie Anfragen bzgl. eines maximal erwarteten Nützlichkeitwertes zur Unterstützung der Entscheidungsfindung. Des Weiteren steueren wir neue Erkenntisse zu gelifteten Beobachtungen bei. Zum einen führen wir unsichere Beobachtungen ein, d.h. (geliftete) Ereignisse kombiniert mit Wahrscheinlichkeiten. Zum anderen untersuchen wir die Auswirkungen von Beobachtungen, da Beobachtungen in einem tem-

poralen relationalen Modell über die Zeit zu Groundings führen kann. Um dieses Problem zu zähmen, präsentieren wir einen Algorithmus für die Beibehaltung eine gelifteten Repräsentation, indem wir in einem Modell Symmetrien mit einem kleinen begrenzten Fehler approximieren.

Wir werten jeden Beitrag dieser Dissertation empirisch aus und überprüfen, wie die Algorithmen funktionieren, während wir die Anzahl der Instanzen und die maximale Anzahl der Zeitschritte erhöhen. Darüber hinaus erarbeiten wir für jeden Teil der Arbeit eine umfangreiche theoretische Analyse. Wir untersuchen außerdem die Verbindung zum Lifting Junction Tree Algorithmus und dem Interface Algorithmus sowie die Beziehung zwischen Lifting und dem Behandeln zeitlicher Aspekte.

Acknowledgments

First of all, I would like to thank my supervisor, Ralf Möller. His suggestions throughout my PhD time turned out to be invaluable. He always comes around the corner with new paper ideas. Once he explained the idea, his final remarks were always something along the lines of "Now you only have to TEX it", which obviously never was as easy as just writing it down. The ideas and the discussions that followed the ideas were always a great help. Having compiled quite some papers, made writing this thesis an easier task as most of the content was already written down. Also thank you for trusting me when I started to come up with my own ideas and also for always pushing me to take the next step as well as for always supporting me. I also would like to thank Maciej Liśkiewicz for reviewing my thesis and Thomas Eisenbarth for chairing my defense.

The past few years have also greatly been shaped by my colleagues, friends, and family. To Angela and Nils, thank you for always having an eye on us and helping whenever possible. To my colleagues for the inspiring discussions during our coffee breaks and the amazing journeys we did together. Thank you to the Mannemers for getting me to walk 10,000 steps a day, which helps me a lot to clear my head after a long working day, and for always being there for me. Thank you to my parents and my grandparents for always believing in me and for making this amazing adventure possible. Thank you, Tanya, for always being there for me, for the past 11 years, for all the amazing journeys we had, for always pushing me to be my best and to leave my comfort zone, and for always being there for me. Thank you for always sticking with me, even when the times were not so easy. I am looking forward to our journey ahead, knowing that we can achieve anything.

Thank you!

Marcel Gehrke
Lübeck, April 2022

Contents

List of Figures

List of Algorithms

List of Symbols

R	Randvar
X	Logvar
A	PRV
$\mathbf{A}$	Set of PRVs
$\mathcal{A}$	Sequence of PRVs
$\mathbf{X}$	Set of logvars
$\mathcal{X}$	Sequence of logvars
$\#_X[R(\mathbf{X})]$	(P)CRV
h	Histogram
$\mathcal{R}(A)$	Range of a PRV
$\mathcal{D}(X)$	Domain of a logvar
$A = a, R = r$	Event
$\mathbf{E}$	Evidence, set of events
Q	Query term
$C, (\mathcal{X}, C_{\mathcal{X}})$	Constraint restricting logical variables
$\top$	Constraint where no restriction applies
ϕ	Potential function
$g, \phi(\mathcal{A})_{\|C}, \phi(\mathcal{A})$	Parfactor
G	Model
$gr(P)$	Grounding
$rv(P)$	PRVs with constraints
$lv(P)$	Logvars
θ	Substitution, alignment
J	FO jtree
$\mathbf{C}$	Parcluster
$\mathbf{S}^{ij}$	Separator
G^i	Local model
m^{ij}	Message
J'	Subtree
G'	Submodel
w_g	Ground width
$w_\#$	Counting width
t	Current time steps

T	Maximum number of time steps
n_J	Number of nodes in an FO jtree
m	Number of queries
n'_J	Number of nodes in a subtree
n	Largest domain size of all logvars
r	Largest range size of all PRVs
$n_\#$	Largest domain size of all counted logvars
$r_\#$	Largest range size of PRVs in CRVs
$\mathcal{M}^{2lv}$	Class of models with two logvars per parfactor
$\mathcal{M}^{1prv}$	Class of models with one logvar per PRV
$\mathbf{Q}$	Set of query terms, conjunctive query
$\mathbf{Q}_{\|C}$	Set of query terms under constraint C, conjunctive query
$\mathcal{CQ}, \mathcal{CQ}^{lift}$	Class of (liftable) conjunctive queries
$\mathcal{TCQ}, \mathcal{TCQ}^{lift}$	Class of (liftable) temporal conjunctive queries
p	Potential
ϕ^P	Potential in ϕ storing also assignments
ϕ^A	Assignment in ϕ storing also assignments

List of Abbreviations

BN	Bayesian network
DBN	dynamic Bayesian network
2TBN	two-slice temporal bayesian network
PM	parameterised probabilistic model
PDM	parameterised probabilistic dynamic model
2TPM	two-slice temporal parameterised model
PDecM	parameterised probabilistic decision model
PDDecM	parameterised probabilistic dynamic decision model
MLN	Markov logic network
MLDN	Markov logic decision network
DMLN	dynamic Markov logic network
QA	query answering
randvar	random variable
logvar	logical variable
PRV	parameterised randvar
CRV	counting randvar
parfactor	parametric factor
jtree	junction tree
FO jtree	first-order junction tree
parcluster	parameterised cluster
VE	variable elimination

LVE	lifted variable elimination
LJT	lifted junction tree algorithm
LDJT	lifted dynamic junction tree algorithm
MPE	most probable explanation
MAP	maximum a posteriori
MEU	maximum expected utility
KRP	keeping reasoning polynomial
TAMe	temporal approximate merging

Chapter 1

Introduction

Areas such as healthcare, logistics, or publishing and cross-sectional aspects such as IT security involve temporal probabilistic relational data. These areas incorporate many objects in relation to each other with changes over time, and also include uncertainties about object existence, attribute value assignments, or relations between objects. Throughout this dissertation, we use a publishing example. Publishing (the relational part) involves publications (objects) for many authors (objects), streams of papers over time (the temporal part), and uncertainties, for example, due to missing or incomplete information. We need efficient inference algorithms for *temporal* probabilistic relational data (Vlasselaer *et al.*, 2014). Further, in applications we need *exact* inference algorithms, because approximate solutions might not be sufficient (Wemmenhove *et al.*, 2007).

A possibility to answer queries for temporal, probabilistic, and relational data is to use temporal probabilistic databases (TPDBs) (Dignös *et al.*, 2012; Dylla *et al.*, 2013). TPDBs are efficient to answer historical queries. For example, if one is interested in whether David Beckham and Zinedine Zidan ever played together for Real Madrid, one can pose a historical query to a TPDB and the result contains different time intervals with assigned probabilities whether the statement holds. Another form of queries are deductive queries for discrete time steps. For such queries on a database, one needs an expressive query language. For example, one could ask queries with a temporal datalog (Chomicki and Imieliński, 1988) program to a database, possibly a TPDB. In a datalog program, one has to specify influences of attributes or random variables (randvars). However, evaluating such a datalog program can be polynomial in the database size.

To answer deductive queries for discrete time steps, we propose to build more expressive and compact models to answer queries more efficiently. In the asking deductive queries with a datalog program on a database, the database would correspond to the model and the datalog program would correspond to the query. The more expressive model that we propose basically consists of the combination of a database and a datalog program. Thus, influences are directly encoded in the model. By increasing the expressiveness of models to directly represent influences in models, the expressiveness of the query language can be reduced. Further, using a compact model, we propose to answer queries on submodels, and thereby avoid repeated calculations to answer multiple queries efficiently. With a model, query answering reduces to computing marginal distributions

at discrete time steps. As a consequence, answering queries is polynomial w.r.t. domain sizes of so-called logical variables (logvars) instead of the size of a database. In this dissertation, we study the problem of repeated exact inference to answer multiple queries in temporal probabilistic relational models. More precisely, we study the problems of answering *hindsight*, *filtering*, and *prediction* queries, which are different kinds of temporal deductive queries, in the first part of this dissertation. We solve the problem by introducing the lifted dynamic junction tree algorithm (LDJT). Additionally, we extend the query language of LDJT to be able to solve even more problems.

1.1 Related Work

For inference on temporal probabilistic propositional models, a naive approach would be to unroll a model for a number of time steps, i.e., to instantiate a temporal pattern for a number of time steps, and use any algorithm for static, i.e., non-temporal, models. However, Papai *et al.* (2012) show how crucial proper handling of temporal aspects is compared to unrolling the model. To prevent unrolling a model, Murphy (2002) proposes the interface algorithm consisting of a forward and a backward pass, which uses a separation of time steps to apply static inference algorithms. The interface algorithm has the following advantages: The separation allows for only keeping one time step in memory and uses only the necessary state descriptions of randvars to proceed from a time step to the next. Thus, instead of the completely unrolled model only one time step is stored in memory. Additionally, the separation allows for caching descriptions of states from previous time steps as well as performing computations on demand based on queries. However, the interface algorithm is defined only for propositional models, which leads to a runtime complexity exponential in the number of model objects.

First-order probabilistic inference leverages relational aspects of a model. For models with known domain sizes, first-order probabilistic inference uses representatives for groups of indistinguishable, known objects, also known as lifting (Poole, 2003). Poole (2003) introduces parametric factor graphs or parameterised probabilistic models (PMs) to represent probabilistic relational models and proposes lifted variable elimination (LVE) as an exact inference algorithm on relational models. LVE saves computations by reusing intermediate results for isomorphic subproblems (so-called lifted summing out). De Salvo Braz (2007) extends LVE by generalising lifted summing out. Further, Milch *et al.* (2008) introduce counting to lift certain computations where lifted summing out does not apply. Apsel and Brafman (2011) refine counting to lift even more cases and Taghipour *et al.* (2013b) generalise counting further. Taghipour *et al.* (2013c) extend LVE to its current form by decoupling the algorithm from the language used to specify objects.

(L)VE can answer single queries efficiently. To answer multiple queries efficiently, Lauritzen and Spiegelhalter (1988) introduce the junction tree algorithm, which allows for answering multiple queries efficiently in probabilistic propositional models using variable

elimination (VE). To benefit from the ideas of the junction tree algorithm and LVE, Braun and Möller (2016) present the lifted junction tree algorithm (LJT) that performs exact first-order probabilistic inference on relational models given a set of queries. However, static inference algorithms do not account for temporal behaviours, which often makes temporal inference infeasible.

Current approaches for lifted inference in temporal probabilistic relational models are approximate. Additionally to being approximate, these approaches involve unnecessary groundings or are not designed to handle multiple queries efficiently. Ahmadi *et al.* (2013) propose lifted loopy belief propagation. From a propositional factor graph, they build a compressed factor graph, i.e., a relational factor graph, by using a colouring algorithm, and construct a dynamic Markov logic network (DMLN). On the DMLN, Ahmadi *et al.* apply lifted loopy belief propagation with the idea of the factored frontier algorithm (Murphy and Weiss, 2001), which is an approximate counterpart to the interface algorithm, to handle temporal aspects of DMLNs. Geier and Biundo (2011) present an online inference algorithm for DMLNs, similar to the work of Papai *et al.* (2012). Both approaches slice DMLNs, i.e., perform inference on one time step at a time, to run well-studied static Markov logic network (MLN) inference algorithms (Richardson and Domingos, 2006) on each slice. Even though, Geier and Biundo (2011) use MLNs, they perform inference on ground factor graphs with a so-called expanding frontier belief propagation (Nath and Domingos, 2010b), which performs adaptive inference for changing evidence. Thus, they reuse computations in case new evidence does not render the computations incorrect. But they do not perform lifted inference. Further, the approximation error depends on a given inference problem and the introduced error might even be unbounded depending on the approximation scheme. Thus, an approximation error might render obtained results insufficient. Furthermore, Papai *et al.* also present how much computed marginal distributions of approximation techniques can differ, which makes the need for *exact* algorithms even more apparent.

Thon *et al.* (2011) introduce CPT-L, a probabilistic model for sequences of relational state descriptions with a partially lifted inference algorithm. The idea is to solve a part of the overall inference problem directly at the first-order level and solve the rest of the inference problem by compiling it into a binary decision diagram. Further, there are approaches for temporal probabilistic relational models without performing lifted inference. Two ways of performing approximative online inference using particle filtering are described by Manfredotti (2009) and Nitti *et al.* (2013). Vlasselaer *et al.* (2016) introduce an exact approach for temporal probabilistic relational models, but perform inference on a ground knowledge base. Similar to LDJT, the approach presented in this dissertation, Vlasselaer *et al.* (2016) also leverage the interface idea, but they apply the idea to knowledge compilation (Darwiche, 2009) in temporal propositional models.

Current approaches, performing inference in temporal probabilistic relational models are approximate, leaving the problem of *exact* inference open. Niepert and Van den Broeck (2014) show that with a lifted solution, the problem of exact inference is tractable.

Further, approximate approaches only solve the *filtering* problem, leaving the problem to answer *hindsight* and *prediction* queries open as well. The problem to answer *multiple queries* in *temporal* relational models efficiently is also still open. To solve these open problems, we propose the so-called LDJT, including a relational forward backward pass, which leverages the well-studied LVE and LJT algorithms. Next, we have a look at the contributions of this dissertation and thereby, the parts that form LDJT.

Throughout this dissertation, we assume familiarity with dynamic Bayesian networks (DBNs) (see Murphy (2002)) as well as factor graphs and their relation to Markov networks (see Taghipour (2013); Van den Broeck (2013)).

1.2 Contributions

In this dissertation, we make a number of contributions to temporal lifted inference. We can summarise the contributions as follows.

(1) Temporal extension to PMs and temporal FO jtrees To be able to specify a temporal relational model, we define a temporal extension to PMs. Using parameterised probabilistic dynamic models (PDMs), we show how first-order junction tree (FO jtree) structures can be constructed such that we have a temporal copy pattern of FO jtrees with each FO jtree allowing for inference over exactly on time step. This contribution comprises (i) identifying parameterised randvars (PRVs) that make one time step conditionally independent from the next and (ii) constructing two FO jtree structures with these PRVs in a single parameterised cluster (parcluster).

(2) Lifted query answering on temporal FO jtrees To be able to proceed in time and thereby, answer *filtering* and *prediction* queries, we introduce a relational forward pass. To be able to go back in time and thereby, answer *hindsight* queries, we introduce a relational backward pass. Additionally, we also ensure preconditions of lifting by preventing unnecessary groundings, when moving in time. Further, we propose an initial query answering plan such that LDJT answers queries efficient w.r.t. computational efforts as well as memory usage.

(3) Soundness, completeness, and complexity results for LDJT We show that LDJT is sound. Further, we show that the completeness results, i.e., model classes for which a lifted solutions are guaranteed, of LJT are not fully transferable due to the temporal aspects handled by LDJT. We also analyse the complexity of LDJT, demonstrating that LDJT is even in the worst case bounded by the complexity of unrolling a temporal model and using LJT. Lastly, we demonstrate the correspondence of LDJT to the ground interface algorithm to show that lifting is crucial.

(4a) Lifted temporal QA for conjunctive queries on temporal FO jtrees We extend the query answering step of LDJT to handle conjunctive queries, in which a single query may contain a set of query terms. The challenge for LDJT arises when the query terms do not appear in a single time step.

(4b) Soundness, completeness, and complexity results for temporal conjunctive queries For conjunctive queries, not only the model but also the query influences whether groundings occur. Thus, the completeness results now also account for the class of queries next to the model class.

(5a) LDJT version for solving temporal MPE queries We present how LDJT can solve temporal most probable explanation (MPE) queries using a max-out instead of a sum-out, while keeping just one time step at a time in memory.

(5b) LDJT version for solving temporal MAP queries Solving maximum a posteriori (MAP) queries is harder compared to MPE queries as max-out and sum-out operations are not commutative. To solve temporal MAP queries, we combine the LDJT version to answer temporal MPE queries and LDJT.

(5c) Soundness, completeness, and complexity results for LDJT versions answering MPE and MAP queries We show completeness for assignment queries over complete time steps given liftable models. Further, we show the correspondence to the complexity of the LDJT versions for solving MPE and MAP queries to LDJT.

(6a) LJT version for solving MEU queries for decision support To allow for decision support, we extend PMs with action and utility nodes. Further, we present a version of LJT to solve the maximum expected utility (MEU) problem.

(6b) LDJT version for solving temporal MEU queries We present an LDJT version to solve the temporal or sequential MEU problem. The LDJT version can reuse many computations for action assignments while answering MEU queries as LDJT handles the temporal aspects efficiently.

(6c) Soundness, completeness, and complexity results for LJT and LDJT versions answering MEU queries We draw similarities between answering MEU queries and marginal queries and show that the completeness results of LDJT can be transferred. Further, we show in the complexity results that using LDJT is faster compared to unrolling either a model or a temporal FO jtree and using LJT to answer MEU queries.

(7a) LVE version for handling uncertain evidence Uncertain evidence means that events are not only true or false but occur with a certain probability. To handle uncertain evidence, LVE cannot use the absorption operator anymore as it is optimised for evidence that is certain. Hence, we present an LVE version for uncertain evidence.

(7b) LJT version for handling uncertain evidence Similar to LVE, with uncertain evidence LJT also cannot use the absorption operator anymore. Thus, we also present an LJT version for uncertain evidence that can be used for LDJT for temporal models.

(7c) Soundness and completeness results for LVE and LJT versions handling uncertain evidence We show that uncertain evidence does not change completeness results.

(8a) Taming the effect of temporal evidence Evidence can slowly ground a model over time. To tame the effects of evidence, we present temporal approximate merging (TAMe) to use approximate symmetries to restore a lifted representation while proceeding in time.

(8b) Error bounds for taming the effect of temporal evidence We show that the error TAMe introduces while approximating symmetries is indefinitely bounded. Further, we show that TAMe is able to approximate symmetries and thereby, restore a lifted representation to keep reasoning polynomial for temporal models.

1.3 Structure

In the next chapter, we present preliminaries for this dissertation. We recapitulate PMs as the representation formalism based on Poole (2003); Taghipour *et al.* (2013c); Braun (2020) and present LJT to answer multiple queries with PMs based on Braun (2020). For exact temporal propositional inference, we propose the interface algorithm based on Murphy (2002).

Afterwards, we present the contributions of this dissertation. We divided the contributions into three parts.

- Part I presents LDJT as the main algorithm of this dissertation to efficiently answer multiple queries for temporal probabilistic relational models.
  - Chapter 3 presents PDMs and LDJT including FO jtree structures construction (Contributions **1**, **2**).
  - Chapter 4 presents a theoretical analysis of LDJT, looking at soundness, completeness, and complexity (Contribution **3**).
  - Chapter 5 presents an empirical evaluation testing five aspects of LDJT.

  This first part was mainly published in

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the 23rd International Conference on Conceptual Structures*, pages 55–69. Springer, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Relational Forward Backward Algorithm for Multiple Queries. In *Proceedings of the 32nd International Florida Artificial Intelligence Research Society Conference (FLAIRS-32)*, pages 464–469. AAAI Press, 2019

Marcel Gehrke, Tanya Braun, and Ralf Möller. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *8th International Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence*, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Towards Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of KI 2018: Advances in Artificial Intelligence*, pages 38–45. Springer, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the AI 2018: Advances in Artificial Intelligence*, pages 556–562. Springer, 2018

In this basic form, LDJT answers multiple *hindsight*, *filtering*, and *prediction* queries for each time step efficiently.

- Part II contains extensions to the query language of LDJT.
  - Chapter 7 presents LDJT for conjunctive queries, allowing for a set of query terms in a single query (Contribution **4a**).
  - Chapter 8 presents LDJT versions for assignment queries, i.e., MPE or MAP queries (Contributions **5a**, **5b**).
  - Chapter 9 presents LJT and LDJT for MEU queries, including extending PDMs to parameterised probabilistic dynamic decision models (PDDecMs) (Contributions **6a**, **6b**).

  Each chapter contains a theoretical analysis (Contributions **4b**, **5c**, and **6c**) as well as an empirical evaluation. The second part is based on the following conference and workshop papers, which we extend in this dissertation with a full theoretical and empirical analysis.

  Marcel Gehrke, Tanya Braun, and Ralf Möller. Answering Multiple Conjunctive Queries with the Lifted Dynamic Junction Tree Algorithm. In

Proceedings of the AI 2018: Advances in Artificial Intelligence, pages 543–555. Springer, 2018

Marcel Gehrke, Tanya Braun, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhäuser. Towards Lifted Maximum Expected Utility. In *Proceedings of the Joint Workshop on Artificial Intelligence in Health in Conjunction with the 27th IJCAI, the 23rd ECAI, the 17th AAMAS, and the 35th ICML*, pages 93–96. CEUR-WS.org, 2018

Marcel Gehrke, Tanya Braun, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhäuser. Lifted Maximum Expected Utility. In *Proceedings of Artificial Intelligence in Health*, pages 131–141. Springer International Publishing, 2019

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Temporal Maximum Expected Utility. In *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*, pages 380–386. Springer, 2019

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Temporal Most Probable Explanation. In *Proceedings of the 24th International Conference on Conceptual Structures*, pages 72–85. Springer, 2019

- Part III makes contributions w.r.t. evidence in LVE, LJT, and LDJT.
  - Chapter 10 presents handling uncertain evidence with LVE, LJT, and LDJT (Contribution **7a**, **7b**) and completeness results for uncertain evidence (Contribution **7c**).
  - Chapter 11 presents taming the effect of temporal evidence by approximating symmetries in LDJT (Contribution **8a**) as well as error bounds for the approximation (Contribution **8b**).

  The last part is based on the following conference papers

  Marcel Gehrke, Tanya Braun, and Ralf Möller. Uncertain Evidence for Probabilistic Relational Models. In *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*, pages 80–93. Springer, 2019

  Marcel Gehrke, Ralf Möller, and Tanya Braun. Taming Reasoning in Temporal Probabilistic Relational Models. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, pages 2592–2599, 2020

The first part and all chapters in part two and three end with an interim conclusion. In Chapter 12, we provide possible future directions lifted temporal inference, including

changing domain sizes as well as incorporating also spatial data and thereby, spatial uncertainty. Chapter 13 concludes this dissertation.

Chapter 2

Preliminaries

To move towards an inference algorithm for relational temporal models, we begin by recapitulating the lifted junction tree algorithm (LJT) (Braun, 2020) and the interface algorithm (Murphy, 2002). LJT exactly answers multiple queries in relational static models efficiently and the interface algorithm answers queries for propositional temporal models efficiently.

2.1 Exact Inference in Probabilistic Relational Models

We recapitulate parameterised probabilistic models (PMs) as a representation for relational static models, which includes the semantics of the model as well as the corresponding inference problems. Afterwards, we present LJT, which solves the inference problems by clustering a model into submodels and using lifting techniques to answer queries.

2.1.1 Parameterised Probabilistic Models

In the following, we use a publication example similar to the so-called competing workshop example from Milch *et al.* (2008) as a running example. We are interested whether a certain research topic is hot. Therefore, we determine how many people do research and publish papers as well as attend conferences in that topic. To model the example, we use that each random variable (randvar) regarding publications, persons doing research and attendance to conferences behaves in the same way w.r.t. a topic being hot. Hence, we do not want to model and reason for each randvar individually. A PM allows for modelling and reasoning over sets of randvars. A PM combines first-order logic with probabilistic models, representing first-order constructs using logical variables (logvars) as parameters. In a PM, we use logvars to obtain parameterised randvars (PRVs) to represent a set of randvars. This form of PM originates from the work by Poole (2003), but we present the definitions given by Braun (2020). We first define a PRV with all its components.

Definition 2.1.1 (PRV, constraint)**.** Let $\mathbf{R}$ be a set of randvar names, $\mathbf{L}$ a set of logvar names, and $\mathbf{D}$ a set of constants. All sets are finite. A *PRV* A is a syntactical construct of

a randvar $R \in \mathbf{R}$ possibly combined with logvars $L_1, \ldots, L_n \in \mathbf{L}$ into $R(L_1, \ldots, L_n), n \geq 0$. If $n = 0$, the PRV is parameterless and constitutes a propositional randvar. The term $\mathcal{R}(A)$ denotes the possible values (range) of a PRV A. An *event* $A = a$ denotes the occurrence of PRV A with range value $a \in \mathcal{R}(A)$. As is common, we abuse notation and write a instead of $A = a$ if A is clearly identifiable. If the range is boolean, we denote $A = true$ by a and $A = false$ by $\neg a$ with a possibly being parameterised. For a set of PRVs $\mathbf{A} = \{A_1, \ldots, A_n\}$, we define $\mathcal{R}(\mathbf{A}) = \bigcup_{i=1}^{n} \mathcal{R}(A_i)$. For a sequence of PRVs $\mathcal{A} = (A_1, \ldots, A_n)$, we define $\mathcal{R}(\mathcal{A}) = \times_{i=1}^{n} \mathcal{R}(A_i)$. Each logvar L has a domain $\mathcal{D}(L) \subseteq \mathbf{D}$. A *substitution* $\theta = \{X_i \to t_i\}_{i=1}^{n} = \{\mathbf{X} \to \mathbf{t}\}$ replaces each occurrence of logvar X_i with term t_i, $t_i \in \mathbf{L}$ or $t_i \in \mathcal{D}(X_i)$. A *constraint* is a tuple $(\mathcal{X}, C_{\mathbf{X}})$ of a sequence of logvars $\mathcal{X} = (X_1, \ldots, X_n)$ and a set $C_{\mathcal{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X_i)$. A PRV A, or logvar L, under constraint C is given by $A_{|C}$, or $L_{|C}$, respectively. The symbol $\top$ for C marks that no restrictions apply, i.e., $C_{\mathcal{X}} = \times_{i=1}^{n} \mathcal{D}(X_i)$. $|\top$ may be omitted in $A_{|\top}$ or $L_{|\top}$.

The term $lv(P)$ refers to the logvars in P, which may be a set of PRVs or a constraint. The term $gr(P)$ denotes the set of all instances of P w.r.t. given constraints. An instance is an instantiation (grounding) of P, replacing the logvars in P with a set of constants from the given constraints. Constraints act as an abstraction for, e.g., instances stored in a database. Let us now model our illustrative example.

Example 2.1.1 (PRV)**.** *We use the randvar names Att, DoR, Hot, and Pub for attends conference, does research, hot topic, and publishes in, respectively, and the logvar names X for researchers, and J for journals. From the names, we build PRVs Att(C), DoR(X), Hot, and Pub(X, J). The domain of X is {alice, bob, eve} and J has domain {springer, aaai_press}. The ranges of Att(X), DoR(X), Hot, and Pub(X, J) are binary. A constraint C = (X, {alice, bob}) for X allows for restricting X to a subset of its domain, in this case to alice and bob. Using the constraint, the expression gr(DoR(X)|C) evaluates to {DoR(alice), DoR(bob)}. The expression gr(DoR(X)|⊤) also contains DoR(eve).*

With PRVs, we need to define how to determine set relations and operations as well as how to perform element tests, to be able to compare PRVs against each other.

Definition 2.1.2. Let two sets $\mathbf{A}'_{|C'}$ and $\mathbf{A}''_{|C''}$ of constrained PRVs be given. Semantically,

$$\mathbf{A}'_{|C'} \circ \ \mathbf{A}''_{|C''} \text{ iff } gr(\mathbf{A}'_{|C'}) \circ gr(\mathbf{A}'_{|C'}),$$

where $\circ \in \{=, \subset, \subseteq, \supset, \supseteq, \cup, \cap\}$. An element test of a PRV $A_{|C} \in \mathbf{A}'_{|C'}$ is determined by $gr(A_{|C}) \subseteq gr(\mathbf{A}'_{|C'})$. An element test of an instance of a PRV $P(\mathbf{x}) \in \mathbf{A}'_{|C'}$ or $P(\mathbf{x}) \in A_{|C}$ is determined by $P(\mathbf{x}) \in gr(\mathbf{A}'_{|C'})$ or $P(\mathbf{x}) \in gr(A_{|C})$ respectively.

In most cases, one does not need to compare groundings when determining such relations or operations but can rather work on the PRVs and their constraints. E.g., for

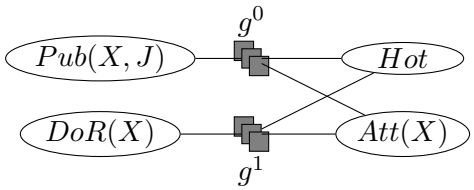


Figure 2.1: Parfactor graph for G^{ex}

an equality of two PRVs, the randvar names and the constraints have to coincide. If one were to assume that different logvar names mean different distinct domains, one might not even need to compare constraints but only the randvar names and the logvars appearing in the PRVs.

We still need a way to set PRVs into relation. Here, parametric factors (parfactors) come into play. A parfactor describes a factor, mapping argument values to real values (potentials), of which at least one is non-zero.

Definition 2.1.3 (Parfactor, PM)**.** Let Φ be a set of factor names, $\mathbf{X} \subseteq \mathbf{L}$ a set of logvars, $\mathcal{A} = (A^1, \ldots, A^k)$ a sequence of PRVs, built from $\mathbf{R}$ and $\mathbf{X}$, and $(\mathcal{X}, C_{\mathcal{X}})$ a constraint on $\mathbf{X}$. Using a function $\phi : \times_{i=1}^{k} \mathcal{R}(A^i) \mapsto \mathbb{R}^+, \phi \in \Phi$, a *parfactor* is given by $\forall \mathbf{x} \in C_{\mathcal{X}} : \phi(\mathcal{A})_{|(\mathcal{X}, C_{\mathcal{X}})}$, substituting $\mathbf{X}$ with $\mathbf{x}$ in $\mathcal{A}$. For short, we write $\phi(\mathcal{A})$ (no substitution), omitting $|(\mathcal{X}, C_{\mathcal{X}})$ if $\top$. A set of parfactors $\{g_i\}_{i=1}^{n}$ forms a *PM*.

The definitions permit models with only propositional randvars. The term $lv(P)$ can also refer to logvars in a parfactor and a PM. The term $gr(P)$ also refers to the set of instances of a parfactor or a PM, leading to a set of grounded parfactors in both cases. The term $rv(P)$ refers to the set of PRVs with their constraints in a parfactor or PM.

Example 2.1.2 (Parameterised model)**.** *Now, we build the PM G^{ex} with the parfactors:*

$$g^0 = \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \phi^0(Pub(x, j), Hot, Att(x))_{|\top} = \phi^0(Pub(X, J), Hot, Att(X))$$
$$g^1 = \forall x \in \mathcal{D}(X) : \phi^1(DoR(x), Hot, Att(x))_{|\top} = \phi^1(DoR(X), Hot, Att(X))$$

We omit the concrete mappings of ϕ^0 and ϕ^1. Parfactors g^0 and g^1 have the constraint $\top$, meaning they hold for all instances. Fig. 2.1 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.

Semantics The semantics of a PM G is given by grounding w.r.t. constraints and building a full joint distribution. In case of a propositional model, grounding does not apply, i.e., $gr(G) = G$. With Z as the normalisation constant, G represents the full joint

probability distribution

$$P_G = \frac{1}{Z} \prod_{f \in gr(G)} f, \qquad Z = \sum_{v \in \mathcal{R}(rv(gr(G)))} \prod_{\phi(\mathcal{A}) \in gr(G)} \phi(\pi_{\mathcal{A}}(v))$$

where $\pi_{\mathcal{A}}(v)$ denotes a projection of the current set of range values v onto $\mathcal{A}$. Three main types of queries in the query answering (QA) problem are:

(i) a probability of a particular event, i.e., $P(Q = q)$,

(ii) a marginal probability distribution of a randvar, i.e., $P(Q)$, or

(iii) a conditional probability distribution of a randvar given a set of events, i.e., $P(Q|\{E^j = e^j\}_{j=1}^m)$.

Answering such queries reduces to computing marginal distributions w.r.t. the full joint distribution of a model. We define a query on a parameterised model as follows.

Definition 2.1.4 (Query, QA problem)**.** A *query* $P(Q|\{E^j = e^j\}_{j=1}^m)$ consists of a *query term* Q, which is a grounded PRV or propositional randvar, and a set of events $\{E^j = e^j\}_{j=1}^m$, where E^j are grounded PRVs or propositional randvars and $e^j \in \mathcal{R}(E^j)$ fixed range values. With a query term and a non-empty set of events, the query is for a conditional distribution. If the set of events is empty, the query is for a marginal distribution. For querying a probability, the query needs to include an event $Q = q$ as query term. The *QA problem* refers to the problem of computing a probability (distribution) for a query.

Example 2.1.3 (Queries)**.** *For G^{ex}, $P(Hot)$ is a query without a set of events asking for the marginal distribution of Hot. The expression $P(Hot|DoR(eve) = true)$ is a query with a single event $DoR(eve) = true$.*

Queries may contain a set of events. Further, the events can also contain symmetries, which can be used for efficient inference. Symmetries in this case mean the same event for multiple groundings. If the underlying model contains PRVs, the set of events may contain symmetries whenever events occur w.r.t. instances of the same PRV. Thus, a set of parfactors can encode the set of events, one parfactor for each subset of events that concern one PRV with the same observation.

Definition 2.1.5 (Evidence)**.** A parfactor $g^e = \phi^e(E(\mathbf{X}))_{|C^e}$ specifies *evidence* for a set of events $\{E(\mathbf{x}^i) = o\}_{i=1}^n$ of a PRV $E(\mathbf{X})$. Factor ϕ^e maps the value o to 1 and the remaining range values of $E(\mathbf{X})$ to 0. Constraint C^e encodes the observed groundings $\mathbf{x}^i$ of $E(\mathbf{X})$, i.e., $C^e = (\mathbf{X}, C_{\mathbf{X}})$ and $C_{\mathbf{X}} = \{\mathbf{x}^i\}_{i=1}^n$.

Example 2.1.4 (Evidence)**.** *Assume we observe $DoR(eve) = true$, which is only one event. The parfactor $\phi^e(DoR(X))_{|C^e}$ represents that eve does research as follows: The factor ϕ^e has the mappings $\phi^e(true) = 1$ and $\phi^e(false) = 0$. C^e restricts the domain of X to eve. As C^e contains one grounding, we can simplify $\phi_e(DoR(X))_{|C^e}$ to $\phi^e(DoR(eve))$ because C^e is a singleton constraint, meaning eve is the only sequence in C^e.*

Assume that there were 100 *people in G_{ex} and we observed the value $DoR(x^i) = true$ for* 90 *of them, i.e., the set of events was $\{DoR(x^i) = true\}_{i=1}^{90}$. Then, C^e would restrict the domain of X to $x^1, \ldots, x^{90}$ in $\phi^e(DoR(X))_{|C^e}$.*

Evidence encoded in parfactors enables an inference algorithm to efficiently handle evidence: The idea is to split parfactors into two, one for the instances with evidence and one for the remaining instances. Then, the parfactor affected by evidence can handle evidence, while the other remains untouched after updating its constraint. The process of splitting up the instances in such a manner is called shattering. Further, the inference algorithm can still employ lifted techniques and does not need to calculate solutions to the QA problem on a ground level (Van den Broeck and Darwiche, 2013). Thus, even symmetries in the evidence help to reduce repeated calculations. Now, we look at LJT, an algorithm to perform inference for multiple queries efficiently.

2.1.2 Inference using the Lifted Junction Tree Algorithm

Having a representation for relational models with uncertainty, we need an efficient inference algorithm, that is to say an inference algorithm performing calculations in a lifted fashion. Let us first give an overview of the steps of LJT (Braun and Möller, 2016) and then have a detailed look at each step. 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 a set of evidence $\mathbf{E}$, by performing the following steps:

(i) Construct a first-order junction tree (FO jtree) J for G, as a helper structure.

(ii) Enter $\mathbf{E}$ in J.

(iii) Pass messages, to use conditional independences to answer multiple queries.

(iv) Compute answer for each query $Q^i \in \mathbf{Q}$.

Now, we go through every step. While explaining the steps, we use operators of lifted variable elimination (LVE) such as *shattering*, *multiplying*, and *lifted summing out* (see (Taghipour *et al.*, 2013c) for a detailed explanation and definition of the operators). We first define an FO jtree, which LJT uses to efficiently answer multiple queries. To be able to define an FO jtree, we begin by defining parameterised clusters (parclusters). Parclusters are the nodes of an FO jtree, which LJT uses to answer a query on a submodel of a PM.

Definition 2.1.6 (Parcluster, FO jtree)**.** Let $\mathbf{X}$ be a set of logvars, $\mathbf{A}$ a set of PRVs with $lv(\mathbf{A}) \subseteq \mathbf{X}$, and $(\mathcal{X}, C_{\mathcal{X}})$ a constraint on $\mathbf{X}$. Then, $\forall \mathbf{x} \in C_{\mathcal{X}} : \mathbf{A}_{|C}$ denotes a *parcluster*, substituting $\mathbf{X}$ in $\mathbf{A}$ with $\mathbf{x}$. We write $\mathbf{A}_{|(\mathcal{X},C_{\mathcal{X}})}$ for short. We omit $|(\mathcal{X}, C_{\mathcal{X}})$ if the constraint is $\top$. Definition 2.1.2 regarding set relations and operations of sets of PRVs also applies to parclusters.

An *FO jtree* for a model G is a cycle-free graph $J = (V, E)$, where $V \subseteq 2^{rv(G)}$ is the set of nodes and $E \subseteq \{\{i, j\} | i, j \in V, i \neq j\}$ the set of edges. Each node in V is a parcluster $\mathbf{C}^i$. J must satisfy three properties: (i) $\forall \mathbf{C}^i \in V : \mathbf{C}^i \subseteq rv(G)$. (ii) $\forall g \in G : \exists \mathbf{C}^i \in V : rv(g) \subseteq \mathbf{C}^i$. (iii) If $\exists A \in rv(G) : A \in \mathbf{C}^i \wedge A \in \mathbf{C}^j$, then $\forall \mathbf{C}^k$ on the path between $\mathbf{C}^i$ and $\mathbf{C}^j : A \in \mathbf{C}^k$ (running intersection property). An FO jtree is *minimal* if by removing a PRV from a parcluster, the FO jtree ceases to be an FO jtree, i.e., it no longer fulfils all properties. The set $\mathbf{S}^{ij}$, called *separator* of edge $\{i, j\} \in E$, is defined by $\mathbf{C}^i \cap \mathbf{C}^j$. The term $nbs(i)$ refers to the neighbours of node i, defined by $\{j | \{i, j\} \in E\}$. Each $\mathbf{C}^i \in V$ has a *local model* G^i and $\forall g \in G_i : rv(g) \subseteq \mathbf{C}^i$. The local models G^i partition G.

LJT constructs a valid and minimal FO jtree (Braun, 2020). A valid FO jtree contains all PRVs of G, but also only these PRVs. Each parfactor is assigned to exactly one parcluster. Thus, the combination of all local models make up G. Further, if a PRV occurs in at least parcluster $\mathbf{C}^i$ and $\mathbf{C}^j$, that PRV also occurs in all parclusters on the path from $\mathbf{C}^i$ to $\mathbf{C}^j$. Additionally, an FO jtree restricts elimination orders for a PM G. For example, to answer a query, all PRVs except the query term need to be eliminated from G. LVE eliminates PRVs using *lifted summing out* and LJT uses LVE for tis calculations. To be able to apply *lifted summing out*, certain preconditions have to hold. Thus, heuristics exist to select an elimination order to answer a query without having to ground. The fact that an FO jtree restricts elimination orders is also the reason why one cannot simply lift the interface algorithm, but an exact lifted algorithm has to ensure preconditions of lifting while proceeding in time, as we will see in Section 3.4.

Example 2.1.5 (FO jtree construction)**.** *Figure 2.2 shows a valid and minimal FO jtree of G^{ex}, with two parclusters,*

$$\mathbf{C}^1 = \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \{Hot, Att(x), Pub(x, j)\}_{|\top} = \{Hot, Att(X), Pub(X, J)\},$$
$$\mathbf{C}^2 = \forall x \in \mathcal{D}(X) : \{Hot, Att(x), DoR(x)\}_{|\top} = \{Hot, Att(X), DoR(X)\},$$

with $\{Hot, Att(X)\}$, as the separator S^{12}. The FO jtree is minimal as removing any PRV leads to violating, e.g., that all randvars of a parfactor are contained in at least one parcluster. The combination of all local models make up G^{ex}. Here, each parfactor from the PM makes up the local model of a parcluster, the ideal case to answer queries. Having the smallest possible submodels in each parcluster of the FO jtree, means that LJT has to eliminate the minimal number of PRVs for query answering. However, such a scenario also means that LJT has to calculate and send the maximum number of messages

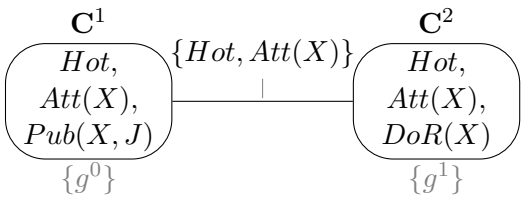


Figure 2.2: Minimised FO jtree of G^{ex} (local models under the parclusters)

during message passing, which is the worst case for message passing. In the following, we detail how LJT performs message passing. Throughout the dissertation, we also work on FO jtrees with more parfactors in a local model of a parcluster, resulting in less messages during message passing but higher query answering efforts.

Having an FO jtree, LJT prepares the FO jtree for query answering by entering evidence in the FO jtree, followed by message passing. Therefore, LJT uses the set of events, builds evidence parfactors from events, and "enters" evidence into the FO jtree. Each evidence parfactor only contains one PRV with corresponding event assignment and constraint. For each evidence parfactor, LJT "enters" the evidence parfactor to each parcluster containing the PRV of the evidence parfactor. Therefore, LJT searches for a parcluster containing the PRV of the evidence parfactor and adds the parfactor to the parcluster. The running intersection property ensures, that if a PRV is contained in multiple parclusters, the PRV is contained in all parclusters on the path between all of the parclusters. Thus, having found one parcluster with the PRV, LJT can use the separators to check whether the PRV is also contained in other parclusters. If the PRV is also contain in another parcluster, LJT also adds the parfactor to that parcluster.

For each evidence parfactor that is added to a corresponding parclusters, LJT performs a so-called *shattering* operation, i.e., split the parfactors of the local model into parts with and without evidence. Basically, LJT multiplies the evidence parfactor e with the parfactor l from the local model. Therefore, LJT splits l into two parts l' and l''. The first part l' is for the instances without evidence. Thus, the parfactor l' is unchanged, just the domain of the logvar is reduced, i.e., the instances for which LJT has evidence are not anymore included in the domain of the logvar. The second part l'' is for the instances with evidence. Hence, the parfactor l'' has a constraint with exactly the instances for which LJT has evidence. Now, LJT could *multiply* e with l''. All rows of the parfactor l'' that disagree with the evidence would be set to 0 and could be removed. LJT has a so-called *absorption* step, to efficiently compute a parfactor that is equivalent to multiplying an evidence parfactor onto another parfactor and dropping the disagreeing lines.

Example 2.1.6 (Evidence entering)**.** *Let us assume that we know that bob does research. Thus, we enter* $DoR(bob) = true$ *as evidence in the FO jtree. First, LJT builds an evidence parfactor* $g^e = \phi^e(DoR(X) = true)_{|C^e}$ *with* C^e *restricted to bob in our example,*

i.e., $C^e = (X, \{bob\})$*. In case we also know that alice does research, then* C^e *would be restricted to bob and alice and so on. Having the evidence parfactor* g^e*, LJT enters* g^e *in the FO jtree. To enter* g^e*, LJT searches for a parcluster containing* $DoR(X)$*, which* $\mathbf{C}^2$ *does. Hence, LJT adds* g^e *to* $\mathbf{C}^2$*. Now, LJT checks whether any separator of* $\mathbf{C}^2$ *contains* $DoR(X)$*, which is not the case for* S^{12}*. Therefore, LJT does not need to add* g^e *to another parcluster.*

Now, LJT first splits the parfactor g^1*, into two parts namely* $g^{1'}$ *and* $g^{1''}$*. The parfactor without the evidence,* $g^{1'}$*, is basically unchanged, with the only difference that the constraint* C *is not top anymore, but* $C = (X, \{alice, eve\})$*, i.e., all instances without bob. The constraint of* $g^{1''}$ *is changed to* $C = (X, \{bob\})$*. Thus, LJT can multiply* g^e *with* $g^{1''}$*, resulting in all lines of* $g^{1''}$ *to be* 0 *where* $DoR(X) \neq true$*. As these lines do not hold any information, LJT can remove them and drop* $DoR(bob)$ *to reduce computation costs. The steps of evidence entering are more efficiently defined in the* absorption *operator.*

The last step to prepare an FO jtree before query answering is message passing. An idea of LJT is to use submodels to answer queries. To be more precise, LJT uses parclusters to answer queries. In general, one possibility to answer queries is to compute a full joint distribution and then eliminate all none query terms, i.e, multiply all parfactors and sum out all non-query terms. However, in the current state of an FO jtree, the parclusters do not necessarily hold all state descriptions of their corresponding PRVs, i.e., the full joint distribution for each parcluster only consists of its assigned parfactors. To be able to answer queries correctly, a parcluster $\mathbf{C}^i$ has to query its neighbours for their state descriptions of the PRVs from $\mathbf{C}^i$, i.e., a parcluster asks for the joint distribution of its PRVs from other parclusters. The neighbours of $\mathbf{C}^i$ might in turn need to query their neighbours to answer the query. Such recursive querying can be efficiently implemented using dynamic programming. To directly distribute all state descriptions, LJT performs a so-called message pass. Each parcluster has local state descriptions, due to the parfactors and evidence assigned to them. Only after local state descriptions of each parcluster is distributed through the FO jtree, LJT can use any parcluster the contains the query term to answer the query. Message passing consists of an *inbound* and an *outbound* pass. After a message pass, each parcluster obtains the complete state descriptions of its PRVs instead of each parcluster querying for the partial state descriptions from other parclusters. During the messages pass, the influences of a PRV to other PRVs are distributed throughout the parclusters. To compute a message, LJT eliminates, i.e., applies lifted summing out to, all non-separator PRVs from the local model and received messages of the parcluster. To calculate a message, incoming messages from the designated receiver of the message to be calculated are ignored. A message is a parfactor as it contains PRVs, a mapping of these PRVs to potentials, and a constraint. After a complete message pass, each parcluster has all the state descriptions required to answer queries about its PRVs.

Example 2.1.7 (Message passing)**.** *During the* inbound *phase of message passing, LJT sends a message* m^{12} *from* $\mathbf{C}^1$ *to* $\mathbf{C}^2$ *and during the* outbound *phase a message* m^{21}

from $\mathbf{C}^2$ *to* $\mathbf{C}^1$*. To calculate* m^{12}*, LJT eliminates the PRV* $Pub(X, J)$ *from* $\mathbf{C}^1$*. Having calculated* m^{12}*, LJT sends the message to* $\mathbf{C}^2$*.*

For m^{21}*, LJT eliminates* $DoR(X)$ *from* $\mathbf{C}^2$*, more precisely from the parfactors* $g^{1'}$ *and* $g^{1''}$*. Here, LJT only eliminates the PRVs from the parfactors in its local model, even though* $\mathbf{C}^2$ *already has one received message. To calculate the message, LJT eliminates* $DoR(X)$ *from* $g^{1'}$ *and* $g^{1''}$ *by lifted summing out. Finally, LJT sends* m^{21} *to* $\mathbf{C}^1$*.*

After message passing, LJT is ready to answer queries by finding a parcluster containing the query term and eliminating all non-query terms in its local model and received messages.

Example 2.1.8 (Query answering)**.** *Having prepared the FO jtree, by evidence entering and message passing, LJT can answer queries on the FO jtree. If we would like to know whether* Hot *holds given bob does research, we query* $P(Hot|DoR(bob) = true)$ *for which LJT can use parcluster* $\mathbf{C}^2$*. LJT eliminates* $DoR(X)$ *and* $Att(X)$ *from* $\mathbf{C}^2$*'s local model* G^2*,* $g^{1'}$ *and* $g^{1''}$*, which are the two shattered parfactors, combined with the received* m^{21}*.*

As mentioned before, to eliminate a PRV, LJT uses lifted summing out. The idea is to compute variable elimination (VE) for one case and exponentiate the result for isomorphic instances. Lifted summing out has some preconditions to be applicable, e.g., the PRV has to have all logvars that are contained in a parfactor (Taghipour *et al.*, 2013c).

Example 2.1.9 (Lifted summing out)**.** *In* G^{ex} *the parfactor* g^0 *has two logvars and one PRVs, namely* $Pub(X, J)$*, contains both logvars. Hence, LJT can apply lifted summing out directly to* $Pub(X, J)$*. Afterwards, LJT can apply lifted summing out to* $Att(X)$*. However, in case LJT needs to compute a message that only contains* Hot *and* $Pub(X, J)$*, then LJT could not apply lifted summing out to* $Att(X)$ *as it does not contain all logvars. LVE cannot eliminate* $Att(X)$ *as* $Pub(X, J)$ *has the logvar* J*, which is not contained in* $Att(X)$*. Hence, without any other techniques to prevent grounding, LVE would have to ground to eliminate* $Att(X)$ *from* g^0*.*

In case LJT would only want to eliminate Hot *from the parfactor* g^1 *it would have a similar problem. Also for this parfactor, LVE cannot apply lifted summing out to* Hot *directly. Hence, without any other techniques to prevent grounding, LVE would also have to ground to eliminate any PRV from* g^1*.*

As there are preconditions for lifted summing out, we have a look at how LJT ensures that it can always apply lifted summing out if the model allows for a lifted solution.

2.1.3 Preventing Groundings

A lifted solution to a query given a model means that LJT computes an answer without grounding a part of the model. Unfortunately, not all models have a lifted solution

because LVE, the basis for LJT, requires certain conditions to hold (Taghipour *et al.*, 2013c) as we have just seen. Therefore, these models involve groundings with any exact lifted inference algorithm. Grounding a logvar is expensive and, during message passing, may propagate through all nodes. LJT has a few approaches to prevent groundings (Braun and Möller, 2017), some originate from LVE and others are specific to LJT and occur due to a non-optimal elimination order.

Lifted Variable Elimination A syntactic construct to prevent groundings is a counting randvar (CRV). The idea behind a CRV is to prevent groundings of a PRV where it does not matter which randvars have a certain range value, but only how many. The range of a CRV is a set of histograms. A particular range value is a histogram that specifies for each range value v of the underlying randvar how many randvars have this value v. A count-conversion helps us to solve our problem form Example 2.1.9, but let us first look at an example.

Example 2.1.10 (CRV as a compact encoding)**.** *Let us have a look at the randvars behind a boolean PRV $R(X)$ to illustrate CRVs. Assuming we have a factor ϕ, mapping the boolean arguments R^1, R^2, and R^3, for the three randvars behind $R(X)$, to potentials, which is the output of the factor, that is defined as follows:*

$$\begin{aligned}&(\neg r^1, \neg r^2, \neg r^3) \mapsto 1,\ (\neg r^1, \neg r^2, r^3) \mapsto 2,\ (\neg r^1, r^2, \neg r^3) \mapsto 2,\ (\neg r^1, r^2, r^3) \mapsto 3,\\ &(r^1, \neg r^2, \neg r^3) \mapsto 2,\ (r^1, \neg r^2, r^3) \mapsto 3,\ (r^1, r^2, \neg r^3) \mapsto 3,\ (r^1, r^2, r^3) \mapsto 2 \end{aligned} \tag{2.1}$$

In all cases, three $false$ values map to 1. Two $false$ values and one $true$ value map to 2. One $false$ value and two $true$ values map to 3. Three $true$ values map to 2. Now, assume a factor ψ with one CRV and a logvar L, denoted as $\psi(\#_L[R(L)])$. Histograms range from $[0, 3]$ to $[3, 0]$, as there are 3 interchangeable arguments, with the first position referring to $true$ and the second to $false$. The factor is defined as follows:

$$[0, 3] \mapsto 1,\ [1, 2] \mapsto 2,\ [2, 1] \mapsto 3,\ [3, 0] \mapsto 2 \tag{2.2}$$

$[2, 1]$ maps to 2 and $[1, 2]$ maps to 3. As the randvars are interchangeable, both factors encode the same information, but the CRV is a more compact representation. Equation (2.1) has $2^3 = 8$ mappings, Eq. (2.2) has $\binom{3+2-1}{2-1} = 4$ mappings (3 randvars, each with 2 range values), which is no longer exponential w.r.t. the number of original inputs.

CRVs are one important construct of LVE to enable lifted computations for, e.g., query answering, and LJT uses LVE for its calculations and lifted dynamic junction tree algorithm (LDJT), one of the main contributions of this dissertation, in turn uses LJT. We formally define a CRV next.

Definition 2.1.7 (Parameterised CRV)**.** Let $R(\mathbf{X})_{|C}$ denote a PRV under constraint C where $lv(R(\mathbf{X})) = \{X\}$, meaning either $\mathbf{X}$ is a singleton set or other inputs to R are

constants. Then, the expression $\#_X[R(\mathbf{X})_{|C}]$ denotes a *CRV*. Its range is the space of possible histograms. A *histogram* h is a set of tuples $\{(v^i, n^i)\}_{i=1}^m$, $v^i \in \mathcal{R}(R(\mathbf{X}))$, $n^i \in \mathbb{N}$, $m = |\mathcal{R}(R(\mathbf{X}))|$, and $\sum_{i=1}^m n^i = |gr(X_{|C})|$. A shorthand notation for the set of tuples is $[n^1, \ldots, n^m]$. As a function, h takes a range value v^i and returns the associated count n^i from the tuple (v^i, n^i). If $\{X\} \subset lv(P(\mathbf{X}))$, the CRV is a *parameterised CRV (PCRV)* representing a set of CRVs. Since counting binds logvar X, $lv(\#_X[R(\mathbf{X})]) = lv(R(\mathbf{X})) \setminus \{X\}$.

Having the definition of a CRV, we can revise Example 2.1.9 to see how a count-conversion can prevent groundings. A count-conversion on a PRV is basically turning the PRV into a PCRV. To be able to count-convert a PRV from a parfactor also some preconditions such as that the logvar, which we want to count, does not occur in another PRV of that parfactor (Taghipour *et al.*, 2013c).

Example 2.1.11 (CRV to prevent groundings)**.** *With a count-conversion, LJT can prevent many groundings. As we saw in Example 2.1.9, LVE cannot directly eliminate $Att(X)$ without grounding. But, LVE can count-convert the J in $Pub(X, J)$. After the count-conversion, $lv(\#_J[Pub(X, J)])$ returns $\{X\}$ and thus, the logvar X is the only remaining logvars and $Att(X)$ contains all logvars in the parcluster. Hence, now LJT can apply lifted summing out to eliminate $Att(X)$.*

However, if one wants to directly eliminate Hot from g^0, LVE can not apply lifting summing out. LVE could count-convert J from $Pub(X, J)$. The X cannot be count-converted as it occurs in both $Pub(X, J)$ and $Att(X)$. Thus, even after the count-conversion, Hot would not contain all logvars of the parfactor, which is a precondition for lifted summing out. Therefore, the elimination order is important to obtain a lifted solution.

As we can see in Example 2.1.11, an elimination order can render a lifted solution impossible. In addition to lifting preconditions, an FO jtree in LJT also poses restrictions on the elimination order. When calculating messages, an FO jtree determines which PRVs from a parcluster have to be eliminated, namely all PRVs that do not occur in the separator. Now, we take a look at how LJT prevents unnecessary groundings, which occur due to a non-ideal elimination order in an FO jtree.

Lifted Junction Tree Algorithm During message passing, LJT eliminates PRVs by lifted summing out. Thus, in case LJT cannot apply lifted summing out, even after count-conversion, it has to ground logvars. The separators in an FO jtree restrict the elimination order. Hence, some PRVs have to be eliminated before others while calculating messages, which can lead to grounding if lifted summing out is not applicable. Based on the elimination order, unnecessary groundings can occur, while another elimination order would not require groundings. In the propositional case the elimination order only influences how many factorisations can be leveraged to avoid building a full joint distribution, i.e., keep the size of intermediate results relatively small. In the lifted case the

elimination order also influences whether LJT can calculate a lifted solution. Finding an optimal elimination order in general is NP-hard (Darwiche, 2009). LJT checks whether unnecessary groundings can occur and adjusts the elimination order accordingly.

LJT applies three tests to check whether groundings occur during message passing. The first test checks if LJT can apply lifted summing out. In case LJT can apply lifted summing out on a PRV to calculate a message, the PRV cannot cause unnecessary groundings during message passing of LJT. In case LJT cannot apply lifted summing out on a PRV, the second test checks to whether groundings can be prevented by count-converting. Unfortunately, even if a count-conversion prevents unnecessary grounding in a parcluster, the count-conversion can lead to groundings in another parcluster. The third test validates that a count-conversion will not result in groundings in another parcluster.

Now, we present the problem and the checks formally, and afterwards we illustrate the checks using G^{ex}. During message passing, a parcluster $\mathbf{C}^i = \mathbf{A}^i_{|C^i}$ sends a message m^{ij} containing the PRVs of the separator $\mathbf{S}^{ij}$ to parcluster $\mathbf{C}^j$. To calculate the message m^{ij}, LJT eliminates the parcluster PRVs not being part of the separator, i.e., $\mathbf{A}^{ij} := \mathcal{A}^i \setminus \mathbf{S}^{ij}$, from the local model and all messages received from other nodes other than j, i.e., $G' := G^i \cap \{m^{il}\}_{l \neq j}$. To eliminate $A \in \mathbf{A}^{ij}$ by lifted summing out from G', we replace all parfactors $g \in G'$ that include A with a parfactor $g^E = \phi(\mathcal{A}^E)_{|C^E}$ that is the lifted product, i.e., the multiplication of the parfactors that include A. Let $\mathbf{S}^{ij^E} := \mathbf{S}^{ij} \cap \mathcal{A}^E$ be the set of randvars in the separator that occur in g^E. For lifted message calculation, it necessarily has to hold $\forall S \in \mathbf{S}^{ij^E}$,

$$lv(S) \subseteq lv(A). \tag{2.3}$$

Otherwise, A does not include all logvars in g^E.

A count conversion may induce Eq. (2.3) for a particular S if

$$(lv(S) \setminus lv(A)) \text{ is count-convertible in } g^E. \tag{2.4}$$

In case Eq. (2.4) holds, LJT count-converts L, yielding a (P)CRV in m^{ij}, otherwise, LJT grounds.

Unfortunately, a (P)CRV can lead to groundings in another parcluster. Hence, count-conversion helps in preventing a grounding if all following messages can handle the resulting (P)CRV. Formally, for each node k receiving S as a (P)CRV with counted logvar L, it has to hold for each neighbour n of k that

$$S \in \mathbf{S}^{kn} \vee \text{ L count-convertible in } g^S. \tag{2.5}$$

LJT adjusts the elimination order in case the checks determine that groundings would occur by message passing between these two parclusters, which is the case

(i) either if Eq. (2.3) and Eq. (2.4) do not hold

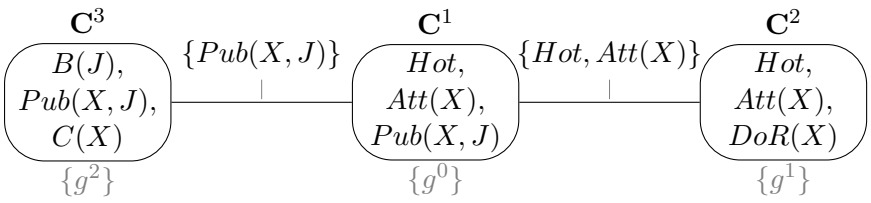


Figure 2.3: FO Jtree of G^{ex} extended to illustrate how LJT prevents groundings

(ii) or if Eq. (2.3) does not hold, Eq. (2.4) holds, and Eq. (2.5) does not hold.

To adjust the elimination order, LJT applies a so-called *fusion* operator to two parclusters, which merges these two parclusters. The idea behind fusing is twofold. On the one hand, fusing the parclusters changes the elimination order by reducing restrictions imposed on the elimination order by an FO jtree and thereby, leads to preventing unnecessary grounding. On the other hand, by fusing the parclusters, LJT does not have to recompute incoming messages to the fused parcluster.

Example 2.1.12 (Preventing groundings in LJT)**.** *Figure 2.3 shows an extended example of G^{ex} to illustrate how LJT prevents unnecessary groundings. Basically, G^{ex} is extended with one new parfactor, $g^2 = \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \phi^2(B(j), Pub(x,j), C(x))_{|\top}$. For the message m^{31} from $\mathbf{C}^3$ to $\mathbf{C}^1$, LJT needs to eliminate $B(J)$ and $C(X)$. To eliminate $B(J)$, LJT cannot apply lifted summing out. LJT also cannot apply a count-convert X in $Pub(X,J)$. Thus, Eq.* (2.3) *and Eq.* (2.4) *do not hold as the logvar X occurs in two PRVs and in one of these PRV together with another logvar. Hence, the checks determine a grounding, and LJT fuses $\mathbf{C}^3$ and $\mathbf{C}^1$.*

Figure 2.4 shows the FO jtree with the nodes fused. For the message m^{12} from $\mathbf{C}^1$ to $\mathbf{C}^2$, LJT needs to eliminate $B(J)$, $C(X)$, and $Pub(X,J)$. To eliminate $Pub(X,J)$, LJT first checks Eq. (2.3)*, which holds. To eliminate $C(X)$, Eq.* (2.3) *holds To eliminate $B(J)$, Eq.* (2.3) *does not hold, but Eq.* (2.4) *and Eq.* (2.5) *do hold. Hence, LJT can eliminate $B(J)$ by count-converting X in $Att(X)$. Thus, LJT can calculate m^{12} without having to ground. For the message m^{21} from $\mathbf{C}^2$ to $\mathbf{C}^1$, LJT needs to eliminate $DoR(X)$. Here, Eq.* (2.3) *holds. Thus, the checks determine no more unnecessary groundings.*

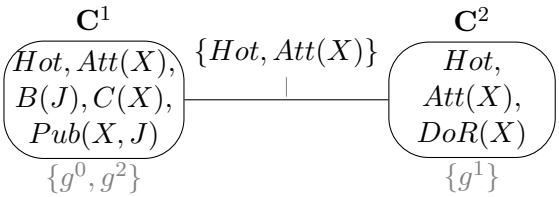


Figure 2.4: FO Jtree of G^{ex} extended and fused to illustrate how LJT prevents groundings

LJT allows for efficiently and exactly answering multiple queries on static relational models. But, our goal is to efficiently answer multiple queries exactly for relational *temporal* models. For propositional models, the interface algorithm (Murphy, 2002) allows for exact answers to multiple queries for propositional temporal models. Next, we have a look at the interface algorithm.

2.2 Exact Inference for Temporal Probabilistic Propositional Models

Murphy (2002) provides in detail an introduction into Bayesian networks (BNs), dynamic Bayesian networks (DBNs), their connections, and how to perform inference on the corresponding models in detail. In the following, we will give a brief overview over inference for DBNs. The interface algorithm is defined on DBNs. A DBN basically is a temporal extension to BNs, which is defined by a prior for the first time step and a temporal copy pattern (Murphy, 2002). One possibility to perform inference on DBNs is to unroll a DBN for a given number of time steps and use any inference algorithm for BNs. Unfortunately, it might not be possible to simply append new time steps while reusing results from previous time steps. In case one cannot append additional time steps, one has to redo all the steps starting from unrolling. Additionally, the unrolled network tends to become very large, as it contains the same information for every time step explicitly, making inference infeasible.

In general, proceeding in time in temporal models boils down to using inference results from time step t in combination with new evidence to answer queries for time step $t+1$. We want to compute $P(\mathbf{A}_{t+1} \mid \mathbf{e}_{1:t+1})$, where $\mathbf{A}_{t+1}$ is the set to state variables from time step $t+1$ and $\mathbf{e}_{1:t+1}$ are the observations from the first time step up to time step $t+1$.

$$P(\mathbf{A}_{t+1} \mid \mathbf{e}_{1:t+1}) \propto P(\mathbf{e}_{t+1} \mid \mathbf{A}_{t+1}) \cdot \sum_{\mathbf{a}_t} P(\mathbf{A}_{t+1} \mid \mathbf{a}_t) \cdot P(\mathbf{a}_t \mid \mathbf{e}_{1:t}) \tag{2.6}$$

Equation (2.6) shows how to proceed in time by using the previous time step for temporal models which follows the Markov assumption (Russell and Norvig, 1995).

To efficiently solve Eq. (2.6), the interface algorithm (Murphy, 2002) comes into play. The idea is to identify randvars, which temporally d-separate time steps. D-separation means that state descriptions about these randvars renders one time step independent from the next. Thus, instead of summing over all $\mathbf{a}_t$, the interface algorithm sums over some of the $\mathbf{a}_t$ that make time steps independent. More specifically, the interface algorithm passes on a description of the state that is needed to answer queries in the next time step. Using d-separation, one can perform inference on one time step and dynamically add new time steps. Further, one is only required to keep the current time step in memory and not all time steps. Additionally, calculations can be carried out on-demand based on queries.

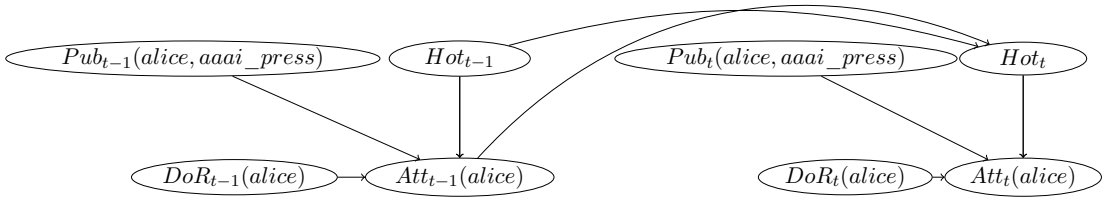


Figure 2.5: $B^{ex}_{\rightarrow}$ a two-slice temporal Bayesian network for model G^{ex}

To recap the interface algorithm, we first recapitulate DBNs as the representation for which the algorithm is defined. Afterwards, we present how the interface algorithm builds temporal d-separated junction tree (jtree) structures (Murphy, 2002) for inference and demonstrate how the structures are reused for multiple queries and time steps.

2.2.1 Dynamic Bayesian Networks

A DBN models temporal behaviour using BNs for discrete time steps (Murphy, 2002). Even though, the name of DBNs include the term "dynamic", stationary processes are modelled, i.e., the structure of the model does not change from one time step to the next and the behaviour of the underlying process remains unchanged.

Definition 2.2.1. A DBN is a pair of BNs $(B_0, B_{\rightarrow})$ where

- B_0 is a BN for the first time step including priors, and
- $B_{\rightarrow}$ is a two-slice temporal bayesian network (2TBN), which models temporal behaviour.

The semantics of a DBN is also given by unrolling the DBN for a given number of time steps and forming a full joint distribution. To unroll a DBN, one starts with B_0 and then expands the BN using $B_{\rightarrow}$ for the given number of time steps.

Example 2.2.1 (Dynamic Bayesian network)**.** *Figure 2.5 shows a 2TBN $B_{\rightarrow}$ for our example and $X = alice$, $J = aaai\_press$. B_0 can be imagined as all the nodes and edges from time step $t-1$ with priors. The 2TBN is basically a temporal copy pattern, defining the temporal behaviour and consisting of two BNs, one for time step $t-1$ and one for t, connected via edges to model the temporal behaviour. Further, we can see that only a subset of the randvars influences the next time step and that a BN is a directed acyclic graph. Using the DBN, we can unroll it for a given number of time steps T, by first instantiating B_0 and then using the temporal copy pattern $B_{\rightarrow}$ to append the BN until the desired time step T is reached, which is the semantics of a DBN.*

Now, we need a way to efficiently answer multiple queries on a DBN.

2.2.2 Inference using the Interface Algorithm

The interface algorithm exploits the fact that the set of nodes with outgoing edges, called interface I_t, to the next time slice from $B_{\rightarrow}$ d-separates the past from the future. With the randvars of I_0, the randvars of I_t for time step 0, the interface algorithm builds a jtree for the BN B_0. While constructing the jtree, the algorithm ensures that the randvars from I_0 end up in one cluster of the jtree to be able to answer a query over I_0 to pass the state descriptions onwards to the next time step. To build a jtree and its clusters from a BN, one way is to first moralise and then triangulate the BN. By adding edges between all nodes from I_0 in the triangulated network, the interface algorithm can ensure that I_0 builds a clique and thus, ends up in a cluster of a jtree. The cluster containing I_0 is labeled *out-cluster*.

Example 2.2.2 (Identifying interface variables and building a jtree for B_0)**.** *In our example, I_t is made up by Hot_t and $Att_t(alice)$. 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. Thus, the interface algorithm first moralises B_0, i.e., the graph is turned into an undirected version by also adding edges between parent nodes. In the moralised graph, the interface algorithm adds edges between the interface randvars. Thus, the algorithm adds edges between Hot_t and $Att_t(alice)$. Lastly, the algorithm constructs a jtree by triangulating the graph and finding clusters. The cluster containing Hot_0 and $Att_0(alice)$ is then labeled* out-cluster*.*

Let us now build a jtree structure for the remaining time steps. As information about the nodes from I_t render one time step independent from the next, we do not need the complete 2TBN, but only the nodes from I_t for time step $t-1$ and the complete network for the time step t. Thus, the algorithm turns $B_{\rightarrow}$ into a 1.5TBN, F_t, by removing all non interface nodes N_{t-1} and their edges from the first slice of $B_{\rightarrow}$, $F_t = B_{\rightarrow} \setminus N_{t-1}$ Murphy (2002). Now, it constructs a jtree J_t for F_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* as a separation from the past and the cluster containing I_t is labeled *out-cluster* as a separation from the future.

Example 2.2.3 (Building the jtree structure for $B_{\rightarrow}$)**.** *First, the interface algorithm constructs F_t out of $B_{\rightarrow}$. Therefore, the algorithm removes all non interface nodes from time step $t-1$. In our example, $DoR_{t-1}(alice)$ and $Pub_{t-1}(alice, aaai\_press)$ are the non-interface randvars. Additionally, all edges in the time step $t-1$ from $B_{\rightarrow}$ are removed. Hence, we are left with the randvars Hot_{t-1} and $Att_{t-1}(alice)$ without any edges for the first time step, but, the edges between time steps and within time step t remain intact. Now, the interface algorithm first moralizes F_t. Afterwards, all interface randvars from I_{t-1} are connected by edges as well as the interface randvars from I_t. Finally, the algorithm triangulates the graph to construct a jtree of the temporal copy pattern. The cluster containing Hot_{t-1} and $Att_{t-1}(alice)$ is labeled* in-cluster *and the cluster containing Hot_t and $Att_t(alice)$ is labeled* out-cluster*.*

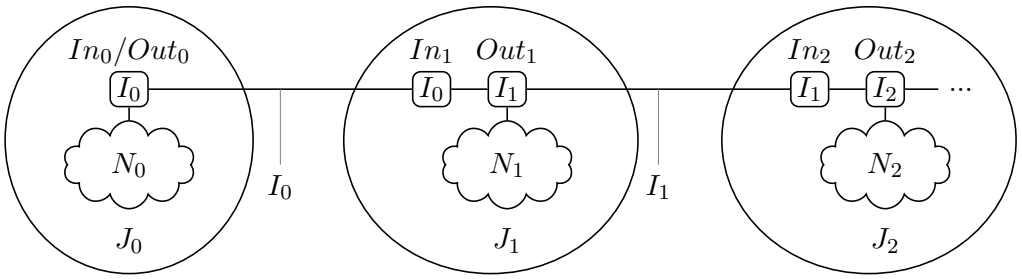


Figure 2.6: Simple abstraction of a possible sequence of jtrees using the interface algorithm (Murphy, 2002)

Now, we investigate how to proceed in time using the structures. To proceed in time, the idea is to compute a message over I_{t-1} from the *out-cluster* of J_{t-1}, i.e., eliminate all non interface randvars from the *out-cluster*. The so-called α_{t-1} message is then passed on to *in-cluster* of J_t combining all past state descriptions in a single message to completely d-separate the jtrees from each other. Thereby, the interface algorithm efficiently solves Eq. (2.6). The α message to proceed in time is also just another query and allows for dynamically adding new time steps. Additionally, one can answer a predefined set of queries for each time step on the jtree, and multiple queries can be solved efficiently on a jtree.

Example 2.2.4 (Proceeding in time using the interface algorithm)**.** *Figure 2.6 illustrates 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 proceed in time. The figure is taken from Murphy (2002) for illustrative purposes. There can be additional clusters between the* in- *and* out-clusters *of a jtree as well as additional clusters connected to the* in-clusters *of a jtree. To reason for $t = 0$, the interface algorithm uses J_0. First, a junction tree algorithm enters evidence in J_0, passes messages, and answers queries. Queries can be either filtering queries, i.e., queries for the current time step, hindsight queries, i.e., queries for a previous time step, or prediction queries, i.e., a time step in the future. For all queries, the queried time step is explicitly referred in the query time, by the name of the randvar. 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, i.e., interface I_0, to J_1. For all $t > 0$, the interface algorithm instantiates J_t on-demand for that time step. 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 from J_0 to J_1's* in-cluster*. After the interface algorithm recovers 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.*

Part I

The Lifted Dynamic Junction Tree Algorithm

Chapter 3

Exact Inference in Temporal Probabilistic Relational Models

In the following, we tackle our goal, namely the problem of exact inference in temporal relational probabilistic models. To that end, we begin by introducing parameterised probabilistic dynamic models (PDMs) as a representation for probabilistic relational temporal models. A PDM is an undirected probabilistic graphical model. For PDMs, we also define different kinds of temporal queries, i.e., *hindsight*, *filtering*, and *prediction* queries, as well as the corresponding query answering problems. Afterwards, we present with LDJT an algorithm to solve the query answering problems. For LDJT, we introduce how to construct timewise-independent FO jtrees from a PDM. The constructed 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, analogously to d-seperation, the counterpart for directional models. To perform inference on the FO jtrees, we introduce a forward pass to solve the *filtering* and *prediction* QA problems and a backward pass to solve the *hindsight* QA problem efficiently. Especially while answering *hindsight* or *prediction* queries, computations can be reused. To save computations and efficiently use resources at hand, we propose a QA plan. Further, we illustrate how LDJT ensures preconditions of lifting by preventing unnecessary groundings that occur based on an non-ideal elimination order. Lastly, we combine all pieces in LDJT, i.e., the first relational forward backward algorithm.

This chapter is based on the following publications:

> Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the 23rd International Conference on Conceptual Structures*, pages 55–69. Springer, 2018
>
> Marcel Gehrke, Tanya Braun, and Ralf Möller. Relational Forward Backward Algorithm for Multiple Queries. In *Proceedings of the 32nd International Florida Artificial Intelligence Research Society Conference (FLAIRS-32)*, pages 464–469. AAAI Press, 2019
>
> Marcel Gehrke, Tanya Braun, and Ralf Möller. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *8th Interna-*

tional Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Towards Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of KI 2018: Advances in Artificial Intelligence*, pages 38–45. Springer, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the AI 2018: Advances in Artificial Intelligence*, pages 556–562. Springer, 2018

3.1 Parameterised Probabilistic Dynamic Models

To define PDMs, we use two PMs and the idea of how BNs give rise to 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 underlying process is stationary, i.e., the model behaviour does not change over time.

Definition 3.1.1 (PDM)**.** A PDM G 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) with $\mathbf{A}_{t-1}$ and $\mathbf{A}_t$ where $\mathbf{A}_\pi$ is a set of PRVs from time slice π.

Example 3.1.1 (PDM)**.** *Figure 3.1 depicts G_0^{ex}, the first time step of our model G^{ex}. Basically, G_0^{ex} is the same parfactor graph as shown in Fig. 2.1. The only difference is that now the PRVs and parfactors are from the first time step. Additionally, each PRV has another parfactor as a form of prior for that PRV.*

Figure 3.2 shows $G_\rightarrow^{ex}$ and thus, defines how the model G^{ex} behaves over time. $G_\rightarrow^{ex}$ consists of G^{ex} for time step $t-1$ and for time step t with inter-slice parfactors for the

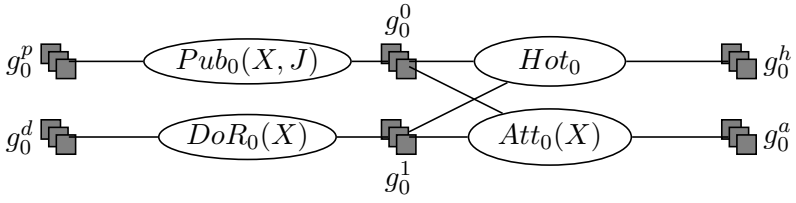


Figure 3.1: G_0^{ex} the first time step for model G^{ex}

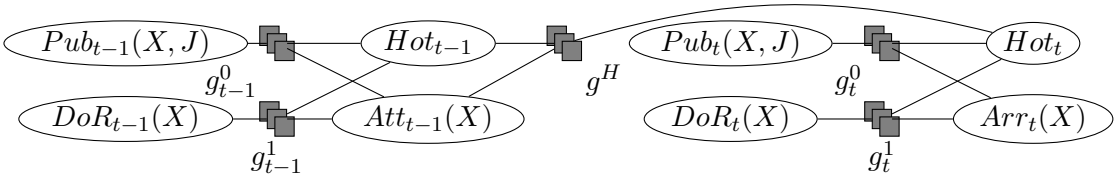


Figure 3.2: $G^{ex}_{\rightarrow}$ the two-slice temporal parfactor graph for model G^{ex}

behaviour over time. Hence, on the left, we can see the parfactor graph as shown in Fig. 2.1 for time step $t-1$ and on the right the same parfactor graph for time step t. g^H is the inter-slice parfactor, modelling temporal behaviour.

Semantics The semantics of a PDM G is given by first unrolling G for T time steps, grounding the unrolled model w.r.t. constraints, and building a full joint distribution. In case of a propositional model, grounding does not apply, i.e., $gr(unroll(G,T)) = unroll(G,T)$, where *unroll* unrolls a PDM G for T time steps into a PM. With Z as the normalisation constant, P_G represents the full joint probability distribution

$$P_G = \frac{1}{Z} \prod_{f \in gr(unroll(G,T))} f, \quad Z = \sum_{v \in \mathcal{R}(rv(gr(unroll(G,T)))) } \prod_{\phi(\mathcal{A}) \in gr(unroll(G,T))} \phi(\pi_{\mathcal{A}}(v))$$

where $\pi_{\mathcal{A}}(v)$ denotes a projection of the current set of range values v onto $\mathcal{A}$. With the difference that now randvars are assigned a time step t, the three main types of queries in the QA problem are again:

(i) a probability of a particular event, i.e., $P(Q_t = q_t)$,

(ii) a marginal probability distribution of a randvar, i.e., $P(Q_t)$, or

(iii) a conditional probability distribution of a randvar given a set of events, i.e., $P(Q_\pi \mid \{E^j_t = e^j_t\}_{j,t})$. Also written $P(Q^i_\pi | \mathbf{E}_{0:t})$.

Answering such queries is equal to computing marginal distributions w.r.t. a model's joint distribution. We define a query on a parameterised model as follows.

Definition 3.1.2 (Temporal queries)**.** A *query* $P(Q_\pi \mid \{E^j_t = e^j_t\}_{j,t})$ consists of a *query term* Q_π, which is a grounded PRV or propositional randvar from time step π, and a set of events $\{E^j_t = e^j_t\}_{j,t}$, where E^j_t are grounded PRVs or propositional randvars from time step t and $e^j_t \in \mathcal{R}(E^j_t)$ are fixed range values. We write $\mathbf{E}_{0:t}$ as a short form for evidence from the first time step up to time step t and $\mathbf{E}_t$ as a short form for evidence from time step t . The problem of answering a query $P(Q_\pi \mid \mathbf{e}_{0:t})$ w.r.t. a model is called *prediction* for $\pi > t$, *filtering* for $\pi = t$, and *hindsight* for $\pi < t$.

Example 3.1.2 (Temporal queries)**.** *For the examples, we represent that no event takes place during a time step with the $\perp$ symbol. For G^{ex}, $P(Hot_0)$ is a query without a set of events asking for the marginal distribution of Hot for the first time step. $P(Hot_2 \mid \perp_0, DoR_1(alice) = true, \perp_2, \perp_3, DoR_4(bob) = true, \perp_5)$ is a* hindsight *query with a single event $DoR(alice) = true$ for time step 1, a single event $DoR(bob) = true$ for time step 4, and no events for all other time steps. $P(Hot_5 \mid \perp_0, DoR_1(alice) = true, \perp_2, \perp_3, DoR_4(bob) = true, \perp_5)$ is a* filtering *query and $P(Hot_8 \mid \perp_0, DoR_1(alice) = true, \perp_2, \perp_3, DoR_4(bob) = true, \perp_5)$ is a* prediction *query.*

Now, we show how we efficiently perform exact lifted temporal inference with PDMs.

3.2 Exact Inference with the Lifted Dynamic Junction Tree Algorithm

To perform exact inference efficiently on PDMs, we introduce LDJT. Before we go into details, we provide a rough overview of LDJT. LDJT efficiently answers queries $P(Q^i_\pi|\mathbf{E}_{0:t})$, with $Q^i_\pi \in \mathbf{Q_t}$, a set of queries for a time step t, and $\mathbf{Q_t} \in \{\mathbf{Q}_t\}_{t=0}^T$, a set of sets of queries for all time steps, possibly the same for all time steps, given a PDM G and evidence $\{\mathbf{E}_t\}_{t=0}^T$, i.e., events for all time steps, by performing the following steps:

(i) Construct two FO jtrees J_0 and J_t with *in-* and *out-clusters* from G.

(ii) For $t = 0$, use J_0 to enter $\mathbf{E}_0$, pass messages on J_0, answer each query term $Q^i_\pi \in \mathbf{Q}_0$, and calculate forward message α_0.

(iii) For $t > 0$, instantiate J_t for the current time step t, add α_{t-1} to the *in-cluster*, enter $\mathbf{E}_t$ in J_t, pass messages on J_t, answer each query term $Q^i_\pi \in \mathbf{Q}_t$, and calculate a forward message α_t.

In the following, we investigate how LDJT can build FO jtrees from a PDMs. Afterwards, we propose a forward and a backward pass for LDJT to proceed in time.

3.2.1 Construction of FO Jtree Structures from a PDM

For LDJT, we adapt the idea of the interface algorithm and lift it to the first-order case to benefit from LJT. Thus, LDJT constructs two FO jtrees, one structure for the initial time step and one for all other time steps. While constructing the FO jtree structures, LDJT ensures that the structures are temporally m-separated. Moreover, LDJT ensures that the interface PRV occur in one cluster, to easily compute the state description for temporal m-separation. In case each FO jtree is time separated from the previous and next FO jtrees, LDJT can perform inference on one time step at a time. LDJT constructs

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 minimised FO jtree for G_0 and remove g_0^I
- g_{t-1}^I := Parfactor for $\mathbf{I}_{t-1}$
- g_t^I := Parfactor for $\mathbf{I}_t$
- F_t := $\{\phi(\mathcal{A})_{|C} \in G_{\rightarrow} \mid \forall A \in \mathcal{A} : A \notin \mathbf{A}_t\}$
- G_t := $(g_{t-1}^I \cup g_t^I \cup F_t)$
- J_t := Construct minimised FO jtree for G_t and remove g_{t-1}^I as well as g_t^I

return $(J_0, J_t, \mathbf{I}_t)$

an FO jtree structure for G_0 with an *out-cluster* and one structure for $G_{\rightarrow}$ with an *in-cluster* and *out-cluster*. Therefore, LDJT first identifies the interface PRVs $\mathbf{I}_t$, i.e., the PRVs which have successors in the next slice, for a time slice t. We define $\mathbf{I}_t$ as follows:

Definition 3.2.1 (Interface PRVs)**.** The forward interface is defined as $\mathbf{I}_t = \{A_t^i \mid \exists \phi(\mathcal{A})_{|C} \in G : A_t^i \in \mathcal{A} \wedge \exists A_{t+1}^j \in \mathcal{A}\}$. The set of non-interface PRVs is $\mathbf{N}_t = \mathbf{A}_t \setminus \mathbf{I}_t$.

Example 3.2.1 (Identifying interface PRVs)**.** *LDJT uses $G_{\rightarrow}$ to identify interface PRVs. In $G_{\rightarrow}$, g^H is the only parfactor that contains PRVs from multiple time steps. Hence, LDJT finds exactly one parfactor having at least one PRV from time step $t-1$ and at least one PRV from time step t. g^H connects the PRVs Hot_{t-1}, $Att_{t-1}(X)$, and Hot_t. Thus, $\mathbf{I}_{t-1}$ contains Hot_{t-1} and $Att_{t-1}(X)$.*

In case the PRVs of $\mathbf{I}$ end up in a single parcluster, LJT can easily compute a message over $\mathbf{I}$. The general idea of ensuring that interface PRVs $\mathbf{I}$ end up in a single parcluster is to add a parfactor g^I over $\mathbf{I}$ to the corresponding PM. To not alter the semantics of the PDM, g^I has uniform potentials in the mappings, i.e., g^I maps all input values to 1. Thereby, the PDM remains semantically the same. In an FO jtree, the PRVs of each parfactor are contained in at least one parcluster. Hence, by adding g^I to a PM, the resulting FO jtree has a parcluster containing $\mathbf{I}$, which LDJT can use as an interface.

The steps of FO jtree constructions are shown in Alg. 1. LDJT constructs two FO jtree structures J_0 and J_t from G_0 and $G_{\rightarrow}$ respectively. To construct J_0, LDJT uses G_0 and adds a parfactor g_0^I over $\mathbf{I}_0$ to G_0. In J_0, g_0^I is assigned to a local model of a parcluster and LDJT labels that parcluster as *out-cluster*. To prevent unnecessary multiplications, LDJT then removes g_0^I from J_0.

Example 3.2.2 (Constructing J_0^{ex})**.** *To construct J_0^{ex}, LDJT adds a parfactor g_0^I to G_0^{ex}. The parfactor g_0^I connects the PRVs from $\mathbf{I}_0$, namely Hot_0 and $Att_0(X)$, and ϕ_0^I*

maps the four assignments to 1. *Figure 3.3 depicts the resulting PM. The only difference to Fig. 3.1 is* g_0^I. *Using the new* G_0^{ex}, *LDJT constructs the* J_0^{ex} *structure. Figure 3.4 shows the corresponding FO jtree structure. The parcluster* $\mathbf{C}_0^1$ *contains the PRVs from* $\mathbf{I}_0$. *Hence, LDJT labels* $\mathbf{C}_0^1$ *as* out-cluster *and removes* g_0^I *from the local model of* $\mathbf{C}_0^1$. *Leaving* g_0^I *would only result in superfluous multiplications with* 1 *and LDJT only needs it to ensure the present of an* out-cluster.

LDJT constructs J_t using $G_\rightarrow$ for all other time steps. To construct J_t, LDJT first prepares $G_\rightarrow$. During inference, LDJT uses J_t to account for the state from the previous time step $t-1$ and answer queries on the current time step t. A description of the state from the interface PRVs suffices to describe the past. Thus, LDJT can prepare the time slice for time step $t-1$ to only include PRVs from $\mathbf{I}_{t-1}$. Hence, LDJT transforms the 2TPM $G_\rightarrow$ into a 1.5-slice TPM F_t, that is to say LDJT eliminates all parfactors from $G_\rightarrow$, which only have PRVs from the first time step. We define the 1.5-slice TPM F_t in the following way:

Definition 3.2.2. $F_t = \{\phi(\mathcal{A})_{|C} \in G_\rightarrow \mid \forall A \in \mathcal{A} : A \notin \mathbf{A}_t^i\}$

During the construction of J_t, LDJT still needs to ensure that J_t obeys the interface idea. Therefore, LDJT ensures that J_t has at least one parcluster containing all PRVs from $\mathbf{I}_{t-1}$ and at least one parcluster containing all PRVs from $\mathbf{I}_t$. Hence, LDJT adds parfactors g_{t-1}^I and g_t^I to F_t. Now, LDJT constructs J_t from F_t. Finally, LDJT labels the parcluster which has g_{t-1}^I assigned as *in-cluster* and the parcluster which has g_t^I assigned as *out-cluster*. Lastly, LDJT removes g_{t-1}^I and g_t^I from the local models.

Example 3.2.3 (Constructing $J_\rightarrow^{ex}$)**.** *LDJT starts by turning the 2TPM* $G_\rightarrow^{ex}$ *into a 1.5-slice TPM* F_t^{ex}. *Therefore, LDJT removes all parfactors and all non-interface PRVs* $\mathbf{N}_{t-1}$ *from time slice* $t-1$ *of* $G_\rightarrow^{ex}$. *Hence, LDJT removes* g_{t-1}^0, g_{t-1}^1, $DoR_{t-1}(X)$, *and* $Pub_{t-1}(X,J)$ *from* $G_\rightarrow^{ex}$ *to obtain* F_t^{ex}. *To ensure that* $J_\rightarrow^{ex}$ *has an* in-cluster *and an* out-cluster, *LDJT adds* g_{t-1}^I *and* g_t^I *to* F_t. *Analogous to the construction of* J_0^{ex}, g_{t-1}^I *connects* Hot_{t-1} *and* $Att_{t-1}(X)$ *and* ϕ_{t-1}^I *maps all four rows to* 1 *and* g_t^I *connects* Hot_t

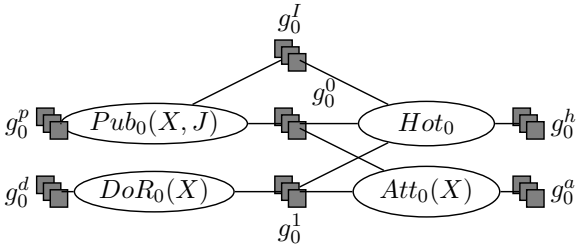


Figure 3.3: G_0^{ex} with interface parfactor

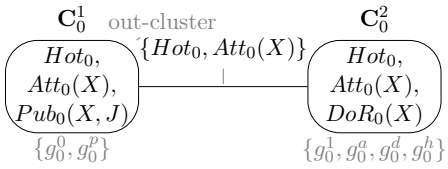


Figure 3.4: FO jtree J_0^{ex} structure

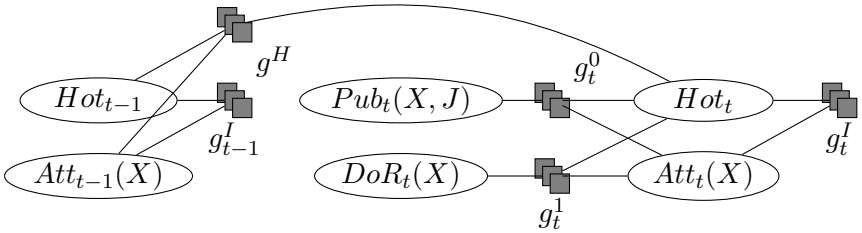


Figure 3.5: 1.5-slice TPM F_t^{ex} with g_{t-1}^I and g_t^I

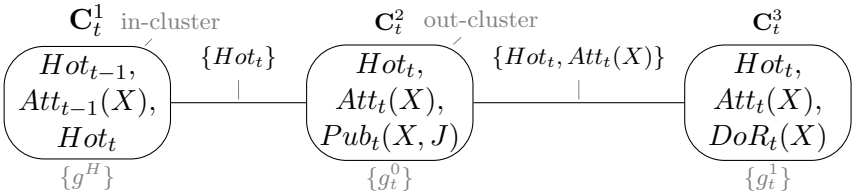


Figure 3.6: FO jtree J_t^{ex} structure

and $Att_t(X)$ and ϕ_t^I maps all rows four to 1*. Figure 3.5 shows the PM corresponding to F_t^{ex} with g_{t-1}^I and g_t^I added.*

Having prepared $G_{\rightarrow}^{ex}$, LDJT constructs the structure $J_{\rightarrow}^{ex}$, which is depicted in Fig. 3.6 with three parclusters. Basically, it is the same structure as shown in Fig. 2.2 with the addition of a parcluster for the incoming interface and the inter-slice parfactor. Additionally, we can see that $\mathbf{C}_t^1$ contains the PRVs from g_{t-1}^I and that that $\mathbf{C}_t^2$ contains the PRVs from g_t^I. Therefore, LDJT labels $\mathbf{C}_t^1$ as in-cluster *and $\mathbf{C}_t^2$ as* out-cluster*.*

Having the FO jtree structures, we now illustrate how LDJT can use the structures to perform inference by combining an *out-cluster* and an *in-cluster* from consecutive time steps. The FO jtree J_0 has an *out-cluster* with the PRVs from $\mathbf{I}_0$, and J_t has an *in-cluster* with the PRVs from $\mathbf{I}_{t-1}$ and an *out-cluster* with the PRVs from $\mathbf{I}_t$. Thus, LDJT can "connect" the *in-* and *out-clusters* of the FO jtrees, i.e. send a message over $\mathbf{I}_t$ from the *out-cluster* of J_t to the *in-cluster* of J_{t+1} . Due to the interface, the clusters contain a minimal set of PRVs to m-separate the FO jtrees. As mentioned, m-separation means that state descriptions about these PRVs renders FO jtrees independent from each other. LDJT uses that given state descriptions about the interface PRVs, the current FO jtree is independent of the FO jtrees from previous time steps.

Next, we present a forward pass for LDJT to be able to answer *filtering* and *prediction* queries. Afterwards, we present a backward pass to also answer *hindsight* queries.

3.2.2 Forward Pass

To proceed in time, LDJT requires a forward pass. The idea of the forward pass is to use the *out-cluster* of J_{t-1}, calculate a message over the interface PRVs $\mathbf{I}_{t-1}$, and then add the message to the *in-cluster* of the FO jtree for the next time step. While the *out-cluster* of J_{t-1} and the *in-cluster* of J_t both can contain more PRVs than just $\mathbf{I}_{t-1}$, they exactly share the PRVs from $\mathbf{I}_{t-1}$. Thus, one could also think of $\mathbf{I}_{t-1}$ being the separator in case one would connect J_{t-1} and J_t with an edge between the *out-* and *in-cluster*. Hence, from that perspective LDJT performs a message pass in one direction to proceed in time.

As J_0 and J_t are valid FO jtrees, LDJT can leverage LJT to perform inference on the FO jtrees. LDJT uses J_0 for the first time step, enters the evidence of the first time step in J_0, and performs a message pass. After the message pass, LDJT answers all queries for the first time step. Having answered all queries, LDJT proceeds in time. LDJT preserves the current state to pass it on to the next time slice. To preserve the state, LDJT computes a forward message called α_0. To compute the message, LDJT sums out all non-interface PRVs from the *out-cluster* of J_0 and stores the result in α_0. The interface PRVs are exactly the PRVs that have an influence on the next time slice and thus, describe all information needed to answer queries for the next time step. Afterwards, LDJT proceeds to the next time step.

For all time steps $t > 0$, LDJT uses the structure of J_t. LDJT instantiates J_t for the current time step t and recovers the state of the previous time step by adding α_{t-1} to the *in-cluster* of J_t. Again, LDJT enters evidence for the current time step and performs a message pass. During message passing, also state descriptions from the α message are distributed. After query answering, LDJT sums out all non-interface PRVs from the *out-cluster* of J_t and saves the result in α_t. Using the interface clusters, the FO jtrees are m-separated from one time step to the next and LDJT can use J_t for all $t > 0$.

Example 3.2.4 (Proceeding in time)**.** *Figure 3.7 depicts how LDJT proceeds from time step* 3 *to* 4*. First, LDJT enters evidence for* $t = 3$ *in* J_3*, distributes local information by message passing, and answers all queries for time step* 3*. To proceed to time step* 4*, LDJT preserves the description of the state of the interface PRVs. To capture the state, LDJT sums out the non-interface PRV* $Pub_3(X, J)$ *from* $\mathbf{C}_3^2$ *and saves the result in message* α_3*. LDJT sums out* $Pub_3(X, J)$ *of the parfactor* g_3^0 *as well as the received messages* m_3^{12} *and* m_3^{32}*. After proceeding in time, LDJT instantiates* J_4 *and adds* α_3 *to the* in-cluster*,* $\mathbf{C}_4^3$*. Now, LDJT enters evidence for* $t = 4$ *in* J_4 *and performs a message pass. During message passing, LDJT accounts for* α_3*. For example, while calculating* m^{12}*, LDJT eliminates* Hot_3 *and* $Att_3(X)$ *from* g^H *as well as* α_3*. Thus,* α_3 *is also accounted for when calculating* α_4*.*

With α messages, LDJT can proceed in time. Now, we illustrate, that the α messages also encode information about previous evidence. To calculate α messages, LDJT accounts for the incoming messages as well as the local model of the *out-cluster*. Dur-

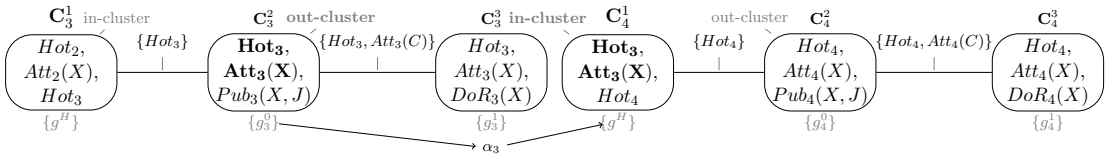


Figure 3.7: Forward pass of LDJT without $\mathbf{C}_3^3$ (local models and labelling in grey)

ing evidence entering, parfactors are shattered into groups with and without evidence. Thus, in case there are shattered logvars in $\mathbf{I}_t$, the shattered groups are present at the *out-cluster* of J_t, either due to its local model or received messages. Therefore, when LDJT calculates an α message, the message also stores information about the different groups. By then adding α to the next time step, the information about the different groups is also present at the next time step, even though LDJT always instantiates a vanilla FO jtree from the corresponding structure. A vanilla FO jtree is a newly instantiated FO jtree from the corresponding structure for the current time step without any entered evidence or calculated message.

Example 3.2.5 (Evidence entering)**.** *Let us assume that alice does research at time step* 3*. Thus, LDJT enters an evidence parfactor encoding $DoR_3(alice) = true$ in J_3. LDJT adds the evidence parfactor to $\mathbf{C}_3^3$, where X is split into a part for alice and into a part for all other instances. During message passes, the message m_3^{32} contains two parts, one for alice and one for all other instances. LDJT then accounts for the two parts, while calculating α_3, which in turn also has two parts. The parfactors of α_3 are in turn accounted for during the message passing in J_4. Thus, LDJT accounts for evidence.*

Using the forward pass, LDJT answers *filtering* and *prediction* queries. To answer *filtering* queries, LDJT uses the FO jtree of the current time step, after LDJT entered evidence and passed messages. To answer *prediction* queries, LDJT uses the evidence up to the current time steps and asks a query for a time step in the future. Thus, LDJT needs to perform forward passes, without new evidence to distribute the state descriptions up until the current time step, while accounting for the temporal model behaviour.

Example 3.2.6 (Answering *filtering* and *prediction* queries)**.** *Assume we have the following queries: $P(Hot_3 \mid .., DoR_3(alice) = true)$, $P(Hot_4 \mid .., DoR_3(alice) = true)$, and $P(Hot_8 \mid .., DoR_3(alice) = true)$. In J_3, LDJT enters $DoR_3(alice) = true$ and passes messages. After the message pass, LDJT answers the* filtering *query and returns the conditional probability distribution of H_3. For the* prediction *query $P(Hot_4 \mid .., DoR_3(alice) = true)$, LDJT computes α_3, instantiates J_4, and adds α_3 to the* in-cluster *of J_4. As it is a prediction query, LDJT has no evidence for time step* 4 *and directly performs a message pass. Afterwards, LDJT uses a parcluster to answer the* prediction *query. For the prediction query $P(Hot_8 \mid .., DoR_3(alice) = true)$, LDJT needs to*

apply additional forward passes until it instantiated J_8. Each forward pass is composed of calculating an α_{t-1} message, adding α_{t-1} to the in-cluster *of J_t, and performing a message pass in J_ts. Using J_8, LDJT answers the second* prediction *query.*

With a forward pass, LDJT can answer *filtering* and *prediction* queries. To answer *hindsight* queries, LDJT needs to propagate state descriptions to previous time steps.

3.2.3 Backward Pass

Using the forward pass, LDJT propagates state descriptions from the the first time step through all time steps in between to the current time step. Thus, the current FO jtree contains state descriptions about all previous time step encoded in the current α message. *Hindsight* queries use the additional evidence from subsequent time steps to answer marginal distribution queries $P(Q^i_\pi|\mathbf{E}_{0:t})$ about an earlier time step, i.e., $\pi < t$. The basic idea here is to use newly observed events to reduce uncertainty about states in previous time steps. For a backward pass to distribute state descriptions also to previous time steps, LDJT also uses the *in-* and *out-clusters*.

To perform a backward pass, LDJT uses the *in-cluster* of J_t, calculates a β_t message over the interface PRVs, and sends β_t to the *out-cluster* of J_{t-1}. To calculate β_t, 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 proceeds with the previous time step. LDJT adds the β_t message to the *out-cluster* of J_{t-1}.

Example 3.2.7 (Backward pass)**.** *Figure 3.8 depicts the backward pass of LDJT. LDJT uses the* in-cluster *of J_4 to calculate β_4. LDJT sums out Hot_4 from the local model of $\mathbf{C}^1_4$, g^H, as well as the received message m^{21}_4. α_3 is not accounted for as the designated receiver of β_4 is the sender of α_3. Having calculated β_4, LDJT proceeds to the previous time step and adds β_4 to the* out-cluster *of J_3. The information of β_4 is then distributed through J_3 during message passing.*

With a backward pass, LDJT can answer *hindsight* queries. To propagate the information, LDJT passes the state descriptions back through every time step between the current time step to the queried time step. In case LDJT only needs to propagate back state descriptions through a time step, but has no queries for that particular time

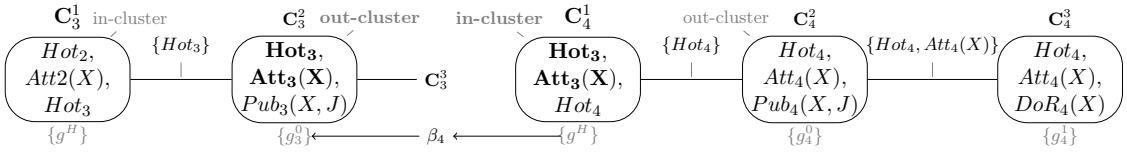


Figure 3.8: Backward pass of LDJT without $\mathbf{C}^3_3$ (local models and labelling in grey)

step, it can reduce computational efforts during a message pass and only ensure that all information are present at the *in-cluster* to calculate the β message.

Let us now illustrate how LDJT answers *hindsight* queries.

Example 3.2.8 (Answering *hindsight* queries)**.** *Assuming that the topic is hot in the current time step* 15*, we would like to know whether alice did research at time step* 5 *and whether alice published at aaai_press at time step* 10*. LDJT answers the conditional distribution queries* $P(Q_{15}^i|\mathbf{E}_{0:15})$*,* $Q_{15}^i \in \mathbf{Q}_{15}$ *with* $\mathbf{E}_{15}$ *consisting of* $\{Hot_{15} = true\}$ *and the set of query terms* $\mathbf{Q}_{15}$ *consisting of at least* $\{Pub_{10}(alice, aaai\_press), DoR_5(alice)\}$*.*

To answer the queries, LDJT instantiates J_{15}*, adds* α_{14} *to the* in-cluster *of* J_{15}*, enters the evidence* $\{Hot_{15} = true\}$*, and passes messages, such that LDJT is able to answer any query for time step* 15*. In this case, LDJT does not have any* filtering *queries, but solely* hindsight *queries. To answer the* hindsight *queries, LDJT has to perform backward passes until it reaches the queried time step. Hence, LDJT calculates* β_{15} *and adds* β_{15} *to the* out-cluster *of* J_{14}*. As LDJT also does not have any queries for time step* 14*, it only needs to ensure that all necessary state descriptions are available to calculate* β_{14}*. Thus, during message passing, LDJT selects the* in-cluster *as root and stops after the* inbound *phase. By doing so, LDJT has all necessary state descriptions present at the* in-cluster *to calculate* β_{14}*, but would not be able to use any other parcluster for query answering, which it also does not need to in this case. LDJT keeps performing backward passes until it reaches time step* 10*. In* J_{10}*, LDJT adds* β_{11} *to the* out-cluster *of* J_{10} *and performs a complete message pass, i.e., pass* inbound *and* outbound *messages. With the complete message pass, all information is distributed and thus, LDJT can use a parcluster for query answering. Hence, LDJT can answer* $P(Pub_{10}(alice, aaai\_press)|Hot_{15} = true)$*.*

To answer the hindsight *query* $P(DoR_5(alice)|Hot_{15} = true)$*, LDJT does not have to start from time step* 15 *again, but can proceed from time step* 10 *to propagate the information to time step* 5*. Thus, LDJT performs five additional backward passes to propagate the information from time step* 15 *to time step* 5*. Having accounted for* β_6 *in* J_5*, LDJT can answer the hindsight query* $P(DoR_5(alice)|Hot_{15} = true)$*.*

If performing solely forward passes, i.e. answering only *filtering* and *prediction* queries, LDJT only needs to keep the current FO jtree in memory. In case LDJT only proceeds in time, LDJT can forget calculations of previous time steps as the forward message encodes all necessary state descriptions to separate the past from the present and thus, previous time steps can be forgotten. However, in case one also wants to answer *hindsight* queries, LDJT has to perform computations on FO jtrees from previous time steps during a backward pass. Hence, LDJT needs an efficient way to perform computations on FO jtrees from previous time steps w.r.t. memory consumption. Additionally, we already have seen that sometimes LDJT does not need to perform a complete message pass, but only an *inbound* message pass, leading to performance considerations as a trade off to being memory efficient. Next, we set up a query answering plan such that LDJT performs only those computation it needs to answer queries.

3.3 Query Answering Plan

During a backward pass, LDJT has different options to instantiate FO jtrees with different implications for performing calculations again as well as memory consumption. For a query answering plan, we start by investigating two approaches how LDJT can obtain FO jtrees from previous time steps. 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 due to the m-separation of the FO jtrees and the forward pass of LDJT. Afterwards, we combine the instantiation approaches into a query answering plan. For the query answering plan, we assume that for each time step LDJT has the same set of queries, i.e., the queries do not change over time and are known in advance.

3.3.1 Preserving FO Jtree Instantiations

Preserving all instantiated FO jtrees, including computed messages, is time-efficient since the approach reuses already performed computations. Thereby, LDJT only needs to account for the newly added β message inside an FO jtree, as this is the only change compared to the previous message pass. By selecting the *out-cluster* as the root node for an *outbound* message pass, this leads to $n - 1$ instead of $2 \cdot (n - 1)$ messages, where n is the number of parclusters. Additionally, if LDJT does not answer any queries on the current FO jtree, LDJT may only pass messages from the *out-cluster* to the *in-cluster*. The *in-cluster* receives the new state descriptions from the *out-cluster* and the remaining messages are still valid as at all others parclusters the information does not change. Further, the required FO jtree is already instantiated and does not need to be instantiated again. The main drawback is the memory consumption, as each FO jtree needs to be stored, which is not always feasible with an increasing number of time steps.

3.3.2 On Demand FO Jtree Instantiation

Instead of keeping all FO jtrees in memory, LDJT can also instantiate an FO jtree on demand. Here, on demand means that LDJT instantiates an FO jtree that is not in memory for computations just in time. To instantiate an FO jtree on demand, LDJT enters evidence, α and β messages, and repeats a message pass. Instantiating FO jtrees on demand is space efficient compared to storing all FO jtrees. Each FO jtree consists of assigned parfactors, messages, including α and β, as well as evidence. Thus, evidence and α message is only a small fraction compared to an FO jtree. With on demand instantiation, LDJT can answer *hindsight* queries even for time steps very far in the past as it is feasible to store required information for more time steps. However, LDJT has to repeat computations. LDJT repeats the steps to instantiate the FO jtree. If LDJT does not answer any queries on the on demand instantiated FO jtree, it needs to compute

$n-1$ messages, by selecting the *in-cluster* as the root for the *inbound* message pass. To prepare an FO jtree for query answering, LDJT has to perform a complete message pass, which results in twice as many messages compared to preserving FO jtree instantiations.

For a query answering plan, let us combine the instantiation approaches to be efficient w.r.t. calculations and memory consumption. Assume that for every time step LDJT answers a predefined set of queries, i.e., for each time step LDJT answers some *hindsight* queries always with the same offset as well as some *filtering* queries, and some *prediction* queries. Then LDJT knows how many backward passes (and forward passes) it has to perform for each time step. With always the same offset, the *hindsight* and *prediction* query answering problem is called a fixed-lag query answering problem. The idea of fixed-lag query answering can be compared to processing a data stream with a sliding window (Özcep *et al.*, 2015), where each window stores a processable amount of data. To preserve FO jtrees instantiated for faster answering of hindsight queries is comparable to sliding windows in stream data processing as one only stores a processable amount of data in a window. Additionally, often in stream data processing, one has predefined queries, which are executed for each window. LDJT preserves a reasonable amount of FO jtrees and while LDJT proceeds in time an FO jtree gets removed from the sliding window and an FO jtree gets added to the sliding window. With a known fixed-lag, a combination of our two approaches is highly advantageous. A combination is also advantageous in case prediction queries and their corresponding offset are predefined.

3.3.3 Combining Instantiation Approaches

As both instantiation approaches have advantages and disadvantages, we leverage the strengths of both by combining them. LDJT can preserve FO jtrees for the fixed-lag and instantiate all other FO jtrees on-demand to allow for on-demand queries, i.e., special queries for a particular time step. Thereby, LDJT can preserve a certain number of FO jtrees instantiated, for fast query answering, as LDJT would need to compute fewer messages. Additionally, with a predefined set of queries, LDJT knows for each FO jtree if there is a query for that time step. Thus, in case there is no query for that particular time step but only for time steps even further in the past, LDJT can perform an *inbound* pass with the *in-cluster* as root to be able to calculate the β message while only performing the necessary computations. In case a *hindsight* query is even further in the past, LDJT instantiates FO jtrees on-demand using evidence and α messages as it is not always feasible to store all FO jtrees and α messages require only a fraction of memory in comparison to FO jtrees. Hence, combining the approaches leads to a query answering plan that is efficient w.r.t. calculations and memory usage.

Example 3.3.1 (Combining instantiation approaches with a fixed-lag)**.** *Assuming the fixed* hindsight *lag is* 10*, LDJT can preserve the last* 10 *FO jtrees and instantiate additional FO jtrees on-demand. Thus, for the* hindsight *queries, which LDJT answers for*

each time step, LDJT only needs to perform one outbound message pass to prepare an FO jtree for query answer and needs to calculate even fewer message if there is no query for a time step. If LDJT only has one predefined hindsight query for $t-10$*, then LDJT may propagate messages only from the* out-cluster *to the* in-cluster *for all FO jtrees but the one for* $t-10$*. For the last FO jtree, LDJT then has to perform an* outbound *message pass from the* out-cluster *to prepare that FO jtree for message passing. Hence, by storing the FO jtrees for the last* 10 *time steps, LDJT can reuse many computations with a manageable overhead w.r.t. memory consumption.*

If an on-demand hindsight *query has a lag of* 20*, LDJT can instantiate the FO jtrees starting with* J_{t-11}*. While answering the default hindsight queries, LDJT already propagates information back to* 10 *time steps earlier. Hence, LDJT can proceed from* J_{t-10}*, calculate* β_{t-10}*, and then instantiate* J_{t-11} *using evidence for* $t-11$*,* α_{t-12}*, and* β_{t-10}*. In case there is no query for time step* $t-11$*, LDJT may perform only an* inbound *message pass with the* in-cluster *as root to calculate* β_{t-11}*. Here a complete* inbound *message pass is needed and not only the messages from* out-cluster *to* in-cluster *as there are no messages stored for that time step. For* J_{t-20}*, LDJT performs a complete message pass to be able to answer queries on that FO jtree and thus, to answer the* hindsight *query.*

With a predefined set of queries that LDJT answers for every time step, LDJT can construct an efficient query answering plan w.r.t. memory consumption and which messages it has to compute. Nonetheless, there still is a potential for investigating heuristics to also efficiently deal with on-demand queries that are often queried for multiple time steps. Another heuristic could deal with which messages LDJT has to compute. Potentially one could also investigate that LDJT does not always need to prepare the FO jtree for query answering but only a subset of parclusters given the predefined queries, leaving room for improvement.

Knowing how LDJT answers *hindsight*, *filtering*, and *prediction* queries, we now show how LDJT prevents unnecessary grounding to be a lifted forward backward algorithm, and then we combine all pieces and present LDJT as a whole.

3.4 Ensuring Preconditions of Lifting

We can distinguish between two different message passes in LDJT, namely an *intra* and an *inter* FO jtree message passing. *Intra* FO jtree message passing takes place inside of an FO jtree. *Inter* FO jtree message passing takes place between two FO jtrees. In both cases unnecessary groundings can occur if the elimination order of an FO jtree does not ensure preconditions of lifting. To prevent groundings during *intra* FO jtree message passing, Braun and Möller propose to fuse parclusters (see Section 2.1.3). LDJT also performs *inter* FO jtree message passing, during forward and backward passes. During an *inter* FO jtree message pass, a message is sent from one time step to another. Hence, LDJT instantiates the FO jtree structure for two consecutive time steps. Unfortunately,

having two FO jtrees from different time steps, LDJT cannot fuse parclusters from different FO jtrees. Otherwise, LDJT would not use temporal independences anymore to reason over only one time step. Hence, LDJT requires a different approach to preventing unnecessary groundings during inter FO jtree message passing. In the following, we present how LDJT prevents grounding and discuss prevention of groundings during *intra* and *inter* FO jtree message passing as well as the implications for a lifted run.

3.4.1 Preventing Groundings while Calculating Temporal Messages

The goal of LDJT is to calculate a lifted solution, if the corresponding model allows for lifted computations. Currently, there might be algorithm induced groundings while calculating temporal messages. To ensure preconditions of lifting while computing temporal messages, we introduce an extension operator on FO jtree structures. LDJT's *extension* performs the following three steps: (i) check whether temporal messages induce groundings, (ii) prevent groundings by extending the set of interface PRVs, and (iii) prevent groundings within one time step with the fusion step of LJT. Algorithm 2 outlines the steps. In the following, we discuss each step in detail.

Checking for Temporal Message Groundings To determine whether calculating temporal messages induces groundings, LDJT uses Eqs. (2.3) to (2.5). For the forward pass, LDJT applies the equations to check whether the α_{t-1} message from J_{t-1} to J_t leads to groundings. More precisely, LDJT checks for possible but unnecessary groundings for the temporal message between J_0 and J_1 as well as between two temporal FO jtree copy patters, namely J_{t-1} to J_t for $t > 1$. Based on the FO jtree construction of LDJT, the structures for J_0 and J_t are different. The parfactors assigned to the out-clusters can be different and also the PRVs that LDJT has to eliminate calculating an α message. Therefore, LDJT checks whether calculations of α_0 and α_t, for $t > 0$, induce groundings.

LDJT checks all PRVs $A \in \mathbf{A}^{ij}$, where i refers to the *out-cluster* from J_{t-1}, j refers to the *in-cluster* from J_t, and $\mathbf{A}^{ij}$ refers to the set of PRVs that LDJT needs to be eliminate calculating α messages, for groundings for $t = 1$ as well as for $t > 1$. In case Eq. (2.3) holds for A, no additional checks for A are necessary as eliminating A does not induce groundings. Otherwise, LDJT checks whether a count-conversion allows for applying lifted summing out on A. In case Eq. (2.4) holds, LDJT tests whether Eq. (2.5) holds in J_t. Here, Eq. (2.5) does not necessarily need to hold for all neighbours. But at least on the path from *in-cluster* to *out-cluster* the count-conversion should not introduce any groundings. For LDJT not having groundings in temporal messages is crucial as these groundings propagate through all time steps. Allowing groundings in parclusters that do not influence temporal messages is not desirable but still better than having groundings in temporal messages. But groundings on the path of temporal messages have to be prevented if possible. Overall, if Eqs. (2.4) and (2.5) hold, eliminating A does not lead to groundings, but if Eq. (2.4) or Eq. (2.5) fail, groundings occur.

Algorithm 2 Preventing Groundings for FO Jtree (J_0, J_t) during a Forward Pass

function PREVENTFORWARDGROUNDINGS(J_0, J_t)
  $\mathbf{C}^i := J_0$(out-cluster)
  $\mathbf{C}^j := J_1$(in-cluster) ▷ J_t instantiated for $t = 1$
  $\mathbf{A}^{\alpha_0} := \mathbf{C}^i \setminus \mathbf{S}^{ij}$
  for $A \in \mathbf{A}^{\alpha_0}$ **do**
    if A induces groundings **then** ▷ Based on Eqs. (2.3) to (2.5)
      Add A to $\mathbf{C}^j$
  $\mathbf{C}^i := J_{t-1}$(out-cluster)
  $\mathbf{C}^j := J_t$(in-cluster)
  $\mathbf{A}^{\alpha_{t-1}} := \mathbf{C}^i \setminus \mathbf{S}^{ij}$
  for $A \in \mathbf{A}^{\alpha_{t-1}}$ **do**
    if A induces groundings **then** ▷ Based on Eqs. (2.3) to (2.5)
      Add A to $\mathbf{C}^j$
  Prevent unnecessary groundings for J_t
  return J_t

Extending Interface Separators If eliminating A leads to groundings, LDJT delays the elimination to a point where the elimination does no longer lead to groundings. Therefore, LDJT adds A to the *in-cluster* of J_t, which results in A also being added to the interface. Later in our theoretical analysis, we also show that delaying eliminations does not change the semantics. Based on the way LDJT constructs the FO jtree structures, the FO jtrees stay valid. Hence, LDJT does not need to eliminate A in the *out-cluster* of J_{t-1} anymore, where the elimination induces groundings.

Checking for Intra FO Jtree Groundings By delaying the elimination, calculating α messages will not induce groundings anymore. Unfortunately, the delayed elimination might induce (unnecessary) groundings inside the J_t structure. To check for and prevent unnecessary groundings in J_t, LDJT uses fusion of LJT as as described in Section 2.1.3. While checking for unnecessary groundings, again it is crucial that no groundings occur on the path from the *in-cluster* to the *out-cluster*. As mentioned before, in case delaying

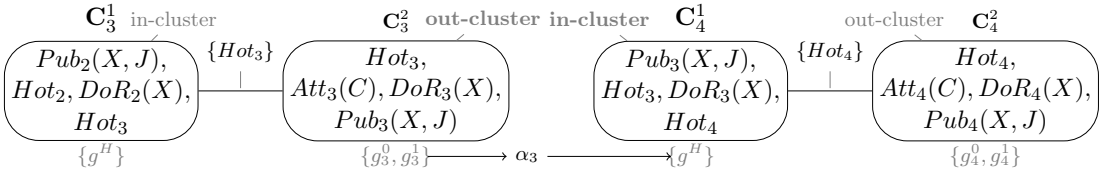


Figure 3.9: J_3 and J_4 with unnecessary groundings

eliminations results in inducing groundings, but not on the path from the *in-cluster* to the *out-cluster*, then delaying eliminations is still worth it as LDJT could calculate α messages using lifting techniques, but will not guarantee a lifted solution is general.

Let us have a look at the central idea of ensuring lifting preconditions while calculating temporal messages.

Example 3.4.1 (Checking for unnecessary groundings during a forward pass)**.** *Figure 3.9 shows J_t instantiated for time step* 3 *and* 4 *with the following example $G^{ex}_{\rightarrow}$:*

$$g^H = \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \phi^H(DoR_{t-1}(x), Hot_{t-1}, Pub_{t-1}(x, j), Hot_t)_{|\top}$$
$$g^0_t = \forall j, x, c \in \mathcal{D}(J) \times \mathcal{D}(X) \times \mathcal{D}(C) : \phi^0_t(Pub_t(x, j), Hot_t, Att_t(c))_{|\top}$$
$$g^1_t = \forall x, c \in \mathcal{D}(X) \times \mathcal{D}(C) : \phi^1_t(DoR_t(x), Hot_t, Att_t(c))_{|\top}$$

J_3 refers to J_{t-1} and J_4 refers to J_t. LDJT checks for groundings while calculating temporal messages for the temporal copy pattern. To compute α_3, LDJT eliminates $Att_3(C)$ from $\mathbf{C}^2_3$'s local model. Hence, LDJT checks whether the elimination leads to groundings. In this example, Eq. (2.3) *does not hold, since $Att_3(C)$ does not contain all logvars, X and J are missing. Additionally, Eq.* (2.4) *is not applicable as X appears in two PRVs and in one PRV with logvar J.*

As eliminating $Att_3(C)$ leads to groundings, LDJT extends the parcluster $\mathbf{C}^1_4$ with $Att_3(C)$. Thereby, LDJT also extends the interface with $Att_3(C)$ and in turn changes the elimination order. Due to the extension, LDJT does not need to eliminate $Att_3(C)$ in $\mathbf{C}^2_3$ anymore and therefore, calculating α_3 does not lead to groundings.

However, LDJT has to check whether adding the PRV $Att_3(C)$ leads to groundings in $\mathbf{C}^1_4$. Figure 3.10 shows the instantiations, after LDJT added $Att_{t-1}(C)$ to the in-cluster *of J_t. For the extended parcluster $\mathbf{C}^1_4$, LDJT needs to eliminate the PRVs Hot_3, $Att_3(C)$, $DoR_3(X)$, and $Pub_3(X, J)$. To eliminate $Pub_3(X, J)$, LDJT first count-converts logvar C in the PRV $Att_3(C)$ and then Eq.* (2.3) *holds for $Pub_3(X, J)$. Afterwards, LDJT can eliminate $DoR_3(X)$ and then the count-converted $Att_3(C)$ as well as the PRV Hot_3 as Eq.* (2.3) *holds for all of them. Thus, by adding the PRV $Att_{t-1}(C)$ to the* in-cluster *of J_t and thereby to the interface, LDJT can prevent unnecessary groundings. Additionally, as LDJT uses this FO jtree structure for all time steps $t > 0$, i.e., the changes to the structure also hold for all $t > 0$.*

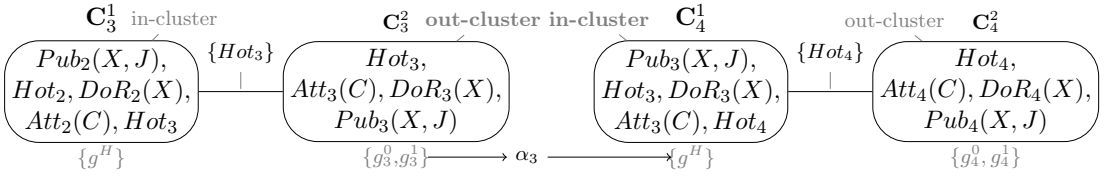


Figure 3.10: J_3 and J_4 after preventing groundings of forward passes

Algorithm 3 Preventing Groundings for FO Jtrees (J_0, J_t) during a Backward Pass

function PREVENTBACKWARDGROUNDINGS(J_0, J_t)
  $\mathbf{C}^i := J_{t-1}$(out-cluster)
  $\mathbf{C}^j := J_t$(in-cluster)
  $\mathbf{A}^{\beta_t} := \mathbf{C}^j \setminus \mathbf{S}^{ji}$
  for $A \in \mathbf{A}^{\beta_t}$ **do**
    if A induces groundings **then**
      Add A to i and J_0(out-cluster)
  Prevent unnecessary groundings for J_0
  Prevent unnecessary groundings for J_t
  return (J_0, J_t)

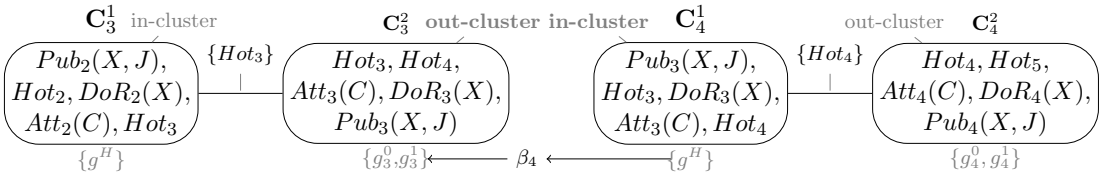


Figure 3.11: J_3 and J_4 after preventing groundings of forward and backward passes

For the forward pass from J_0 to J_1, LDJT identifies the identical unnecessary groundings. Therefore, without extending the in-cluster *of J_t with $Att_{t-1}(C)$, calculating α_0 would induce groundings. In this case, by extending the* in-cluster *of J_t with $Att_{t-1}(C)$, LDJT also prevents unnecessary groundings while calculating α_0.*

So far, we have only focused on how LDJT can prevent unnecessary groundings during a forward pass, as LDJT has to apply a forward pass for each time step. Nonetheless, in the same fashion, LDJT can also prevent unnecessary groundings during a backward pass. The general idea is outlined in Alg. 3. LDJT needs to check whether the calculation of any β message induces groundings. Therefore, LDJT uses the *in-cluster* of the J_t structure and checks whether eliminating any PRV to calculate β_t induces groundings. LDJT then adds all PRVs that induce groundings to the out-clusters of J_0 and J_t. Lastly, LDJT checks J_0 and J_t for unnecessary groundings.

Example 3.4.2 (Preventing groundings during a backward pass)**.** *In the example of Fig. 3.10, LDJT needs to eliminate Hot_4 to calculate β_4. Hot_4 occurs in g^H with $Pub_3(X,J)$, $DoR_3(X)$, $Att_3(C)$, and Hot_3. LDJT cannot apply lifted summing out on Hot_4 and the logvar X cannot be count-converted as it occurs in two PRVs. Thus, eliminating Hot_4 induces groundings. To prevent the groundings, LDJT adds Hot_4 to the* out-cluster *of J_3. Hence, calculating β_4 no longer induces groundings, but LDJT still has to check for groundings in J_3. In J_3 for m^{21}, LDJT may first count-convert C, then*

eliminate $Pub_3(X, J)$. *Afterwards, it eliminates* $DoR_3(X)$. *Finally, LDJT can eliminate the count-converted PRV* $Att_3(C)$ *and* Hot_4. *Thus,* Hot_4 *does not cause any groundings in* J_3 *and LDJT prevents unnecessary groundings during forward and backward passes. Hence, adding* H_1 *to the* out-cluster *of* J_0 *and* Hot_{t+1} *to the* out-cluster *of* J_t *for the forward pass as well as adding* $Att_{t-1}(C)$ *to the* in-cluster *of* J_t *for the backward pass resolves unnecessary groundings.*

3.4.2 Discussion

In the following, we end by discussing workload and performance aspects of preventing grounding of *intra* and *inter* FO jtree message passing. Afterwards, we present model constellations where LDJT cannot prevent groundings.

Performance One needs to ensure preconditions of lifting when lifting an algorithm as shown by the fact that LDJT has to prevent unnecessary groundings. The additional workload for the extension operation of LDJT is moderate. In the best case, LDJT checks Eqs. (2.3) to (2.5) for calculating two messages, namely for the α_{t-1} message and for the message LDJT passes from in *in-cluster* of J_t in the direction of the *out-cluster* of J_t. In the worst case, LDJT needs to check $1 + (m - 1)$ messages, where m is the number of parclusters on the path from the *in-cluster* to the *out-cluster* in J_t. These messages need to be checked twice, once for α_0 and once for α_t.

From a performance point of view, increasing the size of the α messages and of a parcluster is not ideal, but always better than the impact of groundings, which would result in ground calculations for each time step. By applying the *intra* FO jtree message passing check, LDJT may fuse the *in-cluster* and *out-cluster*, which most likely results in a parcluster with many PRVs of the model. Increasing the number of PRVs in a single parcluster increases LDJT's workload for query answering. But even with the increased workload, a lifted run is faster than grounding as we will empirically show in our evaluation. However, in case the checks determine that a lifted solution is not obtainable, using the initial model with local clustering is the best solution due to the smaller parclusters.

LDJT uses the FO jtree construction of LJT, which includes the *fusion* step. Applying *fusion* before *extension* is also more efficient as fusing the *out-cluster* with another parclusters could increase the number of its PRVs and thus, LDJT would have to rerun the *extension* check. Hence, LDJT first applies *fusion* and then *extension*.

Groundings LDJT Cannot Prevent In the following, we have a look at a case where *extension* cannot prevent groundings. The case can be described with a PRV that depends on it predecessors.

Fusing the *in-cluster* and *out-cluster* during *extension* is a case for which LDJT cannot prevent groundings. For such a case to happen, LDJT cannot eliminate a PRV A in the

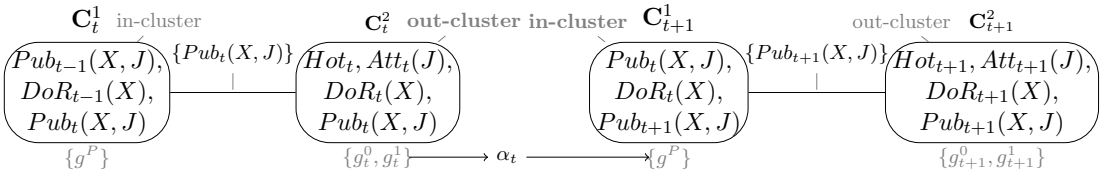


Figure 3.12: Groundings LDJT cannot prevent

out-cluster of J_{t-1} without grounding. Thus, LDJT adds A to the *in-cluster* of J_t. The checks for testing whether LDJT can eliminate A on the path from the *in-cluster* to the *out-cluster* of J_t fail. Thereby, LDJT fuses all parclusters on the path between the two parclusters, but LDJT still cannot eliminate A. LDJT has not been able to eliminate A in the *out-cluster* of J_{t-1} without groundings as well as on the path from the *in-cluster* to the *out-cluster* in J_t. Hence, LDJT also cannot eliminate A in the fused parcluster on the subtree from *in-cluster* to the *out-cluster* without grounding. Even worse, LDJT cannot eliminate A from time step $t-1$ and t in the *out-cluster* to calculate α_t without grounding. For an unrolled model, a lifted solution might be possible, however, with many PRVs in a single parcluster since, in addition to other PRVs, a single parcluster contains A for all time steps. Depending on domain sizes and the maximum number of time steps, either using LDJT with groundings or using the unrolled model with LJT is advantageous as we show in the empirical evaluation.

Example 3.4.3 (Groundings LDJT cannot prevent)**.** *Assume the following* $G_{\rightarrow}^{ex}$*:*

$$\begin{aligned} g_t^0 =& \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \phi^0(Pub_t(x,j), Hot_t, Att_t(j))_{|\top} \\ g_t^1 =& \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \phi^1(DoR_t(x), Hot_t, Att_t(j))_{|\top} \\ g^P =& \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \phi^P(Pub_t(x,j), DoR_t(x), Pub_{t+1}(x,j))_{|\top} \end{aligned}$$

Figure 3.12 depicts FO jtrees for two time steps of $G_{\rightarrow}^{ex}$*. To calculate* α_t*, LDJT could count-convert* X *in* $Pub_t(X,J)$ *from* g_t^0 *and count-convert* X *in* $DoR_t(X)$ *from* ϕ^1*. Afterwards, LDJT can multiply the parfactors and eliminate* $Att_t(J)$ *using lifted summing out. Now, LDJT can count-convert* J *in* $Pub_t(X,J)$*, leading to both variables being count-converted, and finally, summing out* Hot_t*. Unfortunately, the count-conversions lead to groundings in* $\mathbf{C}_{t+1}^1$ *as LDJT cannot count-convert* X *and* J *in* g^P*. To eliminate* $Pub_t(X,J)$ *for the message from* $\mathbf{C}_{t+1}^1$ *to* $\mathbf{C}_{t+1}^2$*, LDJT first needs to multiply* α_t *and* g^P*. However, in* α_t $Pub_t(X,J)$ *is completely count-converted and* g^P *contains two PRVs both with the logvars* X *and* J*. Therefore, LDJT cannot count-convert* X *and* J *in* g^P*. Hence, preparing* g^P *for the multiplication with* α_t *leads to grounding* g^P*.*

The extension *step of LDJT now would try to delay the eliminations of* Hot_t *and* $Att_t(J)$ *to prevent the groundings. However, trying to eliminate* Hot_t *and* $Att_t(J)$ *at*

$\mathbf{C}^1_{t+1}$ *leads to a similar problem. LDJT can eliminate* $Pub_t(X, J)$ *by multiplying* g^I *and* g^0_t*. Afterwards, LDJT has a parfactor that includes* Hot_t*,* $Att_t(J)$*,* $DoR_t(X)$*, and* $Pub_{t+1}(X, J)$*. From that parfactor, LDJT needs to eliminate* Hot_t*,* $Att_t(J)$*, and* $DoR_t(X)$*, which leads to groundings because* X *as well as* J *cannot be count-converted without groundings.*

LJT fuses $\mathbf{C}^1_{t+1}$ *and* $\mathbf{C}^2_{t+1}$*. Unfortunately, LDJT also does not eliminate* $Pub_{t+1}(X, J)$ *in the fused parcluster to calculate* α_{t+1}*. After multiplying* g^I *and* g^0_t *and eliminating* $Pub_t(X, J)$*, LDJT has the same problem as before* fusing*. Therefore, LDJT also cannot eliminate the PRVs here without inducing groundings, even after applying all known techniques to prevent algorithm-induced groundings.*

Let us now combine all pieces, i.e., the forward pass, the backward pass, and ensuring preconditions of lifting, to obtain a relational forward backward algorithm for efficiently solving the *hindsight*, *filtering*, and *prediction* problems by obtaining a lifted solution if the corresponding model allows for a lifted temporal solution. Afterwards, we also look at the theoretical bounds of LDJT, including the model classes for which it is complete.

3.5 Complete Specification of the Lifted Dynamic Junction Tree Algorithm

Algorithm 4 outlines LDJT and its inputs. LDJT constructs FO jtree structures J_0 and J_t and the set of interface PRVs using the function *DFO-JTREE* as described in Section 3.2.1. The construction of the structures also includes the *fusion* step of LJT to prevent unnecessary groundings within an FO jtree as described in Section 2.1.3. Then, LDJT checks the structures for unnecessary groundings while calculating temporal messages and prevents them (cf. Section 3.4).

Afterwards, LDJT answers queries by entering evidence, message passing, query answering for the current time step, and proceeding in time. With the *AnswerQuery* procedure, LDJT answers *hindsight*, *filtering*, and *prediction* queries. LDJT answers *filtering* and *prediction* queries as described in Section 3.2.2 with the *ForwardPass* function and *hindsight* queries as described in Section 3.2.3 with the *BackwardPass* function. For query answering, LDJT starts by identifying the query type of the current query, namely *hindsight*, *filtering*, and *prediction*. To perform *filtering*, LDJT passes the query and the current FO jtree to LJT to answer the query. For *prediction* queries, LDJT applies the forward pass until it reaches the time step of the query and then answers the query. To answer *hindsight* queries, LDJT applies the backward pass until the time step of the query is reached and answers the query.

The *AnswerQuery* procedure is the point to implement a query answering plan (cf. Section 3.3) Answering multiple queries on an FO jtree is efficient as LJT reuses computations to answer queries. With a good query answering plan, LDJT reuses as many computations as possible to answer queries while also being memory efficient.

Algorithm 4 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$(G_0, G_\rightarrow)$
  $(J_0, J_t) :=$ PreventForwardGroundings(J_0, J_t)
  $(J_0, J_t) :=$ PreventBackwardGroundings(J_0, J_t)
  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)

For example, a robot with a stream of location data always queries where he was 2 and 4 time steps ago. LDJT can first answer *hindsight* query with a lag of 2. For the *hindsight* query with a lag of 4, LDJT can reuse 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} and does not need to recompute the already performed two backward passes for lag 2. For a query answering plan, there are two options to reuse the computations.

To reuse computations, the first option for a query answering plan 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 also reuses computations for *prediction* queries. Additionally, under the presence of *prediction* queries, LDJT uses the computed α_t to proceed to the next time step as otherwise LDJT would compute the same α_t message twice. However, given new evidence for a new time step, all other α and β messages that LDJT has calculated for the previous time step are invalid.

Now, we illustrate how LDJT works.

Example 3.5.1 (LDJT)**.** *Assume that the query terms are:*

- $\{Hot_0, Pub_0(alice, aaai\_press), Hot_2, Pub_2(alice, aaai\_press)\}_0$,
- $\{Hot_1, Pub_1(alice, aaai\_press), Hot_3, Pub_3(alice, aaai\_press)\}_1$,
- $\{Hot_0, Pub_0(alice, aaai\_press), Hot_2, Pub_2(alice, aaai\_press), Hot_4, Pub_4(alice, aaai\_press)\}_2$,
- $\{\bot_3\}_3$, *and*
- $\{Hot_0, Pub_0(alice, aaai\_press), Hot_2, Pub_2(alice, aaai\_press), Hot_4, Pub_4(alice, aaai\_press)\}_4$

and the evidence is:

- $\{DoR_0(alice) = true, DoR_0(eve) = true\}_0$,
- $\{DoR_1(alice) = true, DoR_1(eve) = true\}_1$,
- $\{\bot_2\}_2$,
- $\{DoR_3(bob) = true\}_3$, *and*
- $\{DoR_4(alice) = true, DoR_4(eve) = true\}_4$.

We have queries for time steps 0, 1, 2, *and* 4 *and evidence for time steps* 0, 1, 3, *and* 4. *Providing LDJT with queries, evidence, and PDM* G^{ex}*, we illustrate how LDJT works.*

LDJT first sets the current time step to be 0 *and then constructs the FO jtree structures* J_0 *and* J_t*. Therefore, LDJT applies* DFO-Jtree *construction to* G_0^{ex}*, which is depicted in Fig. 3.1, and* $G_{\rightarrow}^{ex}$*, which is depicted in Fig. 3.2, and obtains* J_0^{ex}*, which is shown in*

Fig. 3.4, and J_t^{ex}, which is shown in Fig. 3.6. The FO jtree construction of LJT, which LDJT uses after it changed the PMs G_0^{ex} and $G_{\rightarrow}^{ex}$ already includes fusion *to prevent groundings. Nonetheless, LDJT still has to check for unnecessary groundings while the calculation of temporal messages. Thus, LDJT first checks and prevents unnecessary groundings while calculating α messages and applies first the* PreventForwardGroundings *function from Alg. 2 and then the* PreventBackwardGroundings *function from Alg. 3 to J_0^{ex} and J_t^{ex}. The checks find no unnecessary groundings and return J_0^{ex} and J_t^{ex}.*

With the structures J_0^{ex} and J_t^{ex}, LDJT can proceed to answer queries. The current time step 0 is not equal to $4+1$. Hence, LDJT enters the main while loop. LDJT begins by entering the evidence for the first time step in J_0^{ex}. LDJT builds an evidence parfactor encoding $DoR_0(alice) = true$ and $DoR_0(eve) = true$ and enters the evidence parfactor to the corresponding parcluster of J_0^{ex}. Afterwards, LDJT performs a message pass on J_0^{ex} to distribute the information of the local models and the evidence. Having J_0^{ex} prepared, LDJT can start to answer queries. For time step 0, LDJT answers the query terms Hot_0, $Pub_0(alice, aaai\_press)$, Hot_2, and $Pub_2(alice, aaai\_press)$, that is, two filtering *and two* prediction *queries. First, LDJT answers the query for query term Hot_0. In the* AnswerQuery *procedure, LDJT does not enter the while loop as the query term is for the current time step. Thus, LDJT can directly use J_0^{ex} to answer the query. The same holds for the next query term $Pub_0(alice, aaai\_press)$. For the query term Hot_2, LDJT enters the while loop, as $0 \neq 2$. As $0 < 2$ holds, LDJT applies a forward pass. During the forward pass, LDJT calculates α_0, increases t by one, instantiates J_1, and adds α_0 to J_1. To prepare J_1, LDJT then performs a message pass. An efficient query answering plan determines that only an* inbound *phase with the* out-cluster *as root suffices as currently LDJT has no queries to be answered for time step 1. As 1 is still smaller than 2, LDJT performs another forward pass and prepares J_2. After the message pass on J_2, LDJT can answer the query term Hot_2 using J_2. The last query term is $Pub_2(alice, aaai\_press)$. In the* AnswerQuery *procedure, LDJT again enters the while loop since $0 < 2$ holds. However, LDJT can reuse computations and directly use J_2 to answer the query. Now, LDJT has answered all queries for time step 0.*

With all queries answered for the first time step, LDJT proceeds to the next time step. Therefore, LDJT performs a forward pass. However, LDJT has already calculated α_0 during a prediction query. Thus, LDJT does not to compute α_0 again. As mentioned earlier, the α messages store state descriptions about groups that are split due to evidence. Hence, α_0 contains parfactors constrained to alice and eve as well as parfactors for all other persons. During this forward pass, LDJT only increases t by 1.

LDJT remains in the main while loop. Hence, LDJT enters the evidence for $t = 1$ in J_1. LDJT builds an evidence parfactor encoding $DoR_1(alice) = true$ and $DoR_1(eve) = true$ and enters the evidence parfactor into the corresponding parclusters of J_1^{ex}. During messages passing, LDJT distributes state descriptions of local models. Now, the local models also contain α_0 and thereby, state descriptions from time step 0. Answering the query terms Hot_1, $Pub_1(alice, aaai\_press)$, Hot_3, and $Pub_3(alice, aaai\_press)$, which

again are two filtering *and two* prediction *queries, leads to the same steps, which we omit, as in time step* 0*. After answering all queries for time step* 1*, LDJT again performs a forward pass to proceed in time.*

For time step 2*, LDJT does not have any evidence. Thus, LDJT can reuse* J_2^{ex}*, for which LDJT already performed an* inbound *message pass while answering* prediction *queries from time step* 1*. To complete message passing, LDJT performs an* outbound *message pass. For the time step* 2*, LDJT has two* hindsight*, two* filtering*, and two* prediction *queries. LDJT again uses the current FO jtree* J_2^{ex} *to answer the two* filtering *queries,* Hot_2 *and* $Pub_2(alice, aaai\_press)$*, and performs two forward passes for the* prediction *queries,* Hot_4 *and* $Pub_4(alice, aaai\_press)$*. For the query terms* Hot_0 *and* $Pub_0(alice, aaai\_press)$*,* $2 > 0$ *holds in the* AnswerQuery *procedure. Hence, LDJT performs a backward pass. LDJT calculates* β_2 *using the* in-cluster *of* J_2^{ex} *and adds* β_2 *to the* out-cluster *of* J_1^{ex}*. Since LDJT does not have any* hindsight *query for time step* 1*, the query answering plan of LDJT determines that LDJT only needs to perform an* inbound *message pass with the* in-cluster *of* J_1^{ex} *as root to be able to calculate* β_1*. As* 1 *is still larger than* 0*, LDJT performs another backward pass. LDJT propagates state descriptions from time step* 2 *back to time step* 0*, to be able to answer the* hindsight *queries. Using* J_0^{ex}*, LDJT answers the query terms* Hot_0 *and* $Pub_0(alice, aaai\_press)$*.*

Since LDJT does not have to answer queries for time step 3*, it is only important that information about the evidence, local models, and* α_2 *is present at the* out-cluster *of* J_3*, to calculate* α_3*. The message passing step on* J_3^{ex} *ensures that all necessary state descriptions to calculate* α_3 *are present at the* out-cluster*. Thus, LDJT proceeds in time by performing a forward pass.*

For time step 4*, LDJT first enters evidence and performs a message pass. The queries LDJT answers for time step* 4 *are similar to the queries for time step* 2*. The only difference is that LDJT has hindsight queries for different time steps. Here, we omit answering the* filtering *and* prediction *queries since the procedure from time step* 0 *and* 2 *remains the same. To answer the two* hindsight *queries* Hot_2 *and* $Pub_2(alice, aaai\_press)$*, LDJT again performs two backward passes. To answer the remaining two* hindsight *queries* Hot_0 *and* $Pub_0(alice, aaai\_press)$*, LDJT only needs to perform two additional backward passes, as it can start from* J_2^{ex}*, which LDJT instantiated to answer the hindsight queries* Hot_2 *and* $Pub_2(alice, aaai\_press)$*. Having answered all queries for time step* 4*, LDJT again performs a forward pass. Now, the while loop check fails, as* $5 = 5$ *and LDJT leaves the while loop. In case one would add new evidence and queries for additional time steps, LDJT would proceed to answer these queries.*

In Example 3.5.1, we have illustrated how LDJT works as a whole and how it efficiently can reuse computations for the temporal aspects. Before we also empirically evaluate LDJT, we first take a look from a theoretical perspective at how LDJT performs in principle.

Chapter 4

Theoretical Analysis

This chapter presents soundness, completeness, and complexity results of LDJT. In this context, soundness means that the algorithm produces answers equivalent to answers of any sound inference algorithm. Completeness investigates for what kind of classes of models the lifted algorithm provides an answer without grounding. A class of models is characterised in terms of logvars, e.g., one class consists of all possible models built with at most 2-logvars in a parfactor (2-logvar models). First, we take a look at soundness and then at completeness of LDJT. Finally, we investigate the complexity of LDJT and its instantiation approaches.

4.1 Soundness

To show that LDJT is sound, i.e., all numbers are computed correctly, we show that all parts of LDJT are sound. We begin with the soundness of FO jtree constructing and end with the soundness of query answering.

Braun (2020) show that the FO jtree construction of LJT is sound. For LDJT, we show that the constructed FO jtree structures obey the interface idea. Additionally, we show that after *extension*, FO jtrees structures are still valid.

Theorem 4.1.1. *LDJT constructs FO jtree structures with interface PRVs in a single parcluster.*

Proof. To ensure that interface PRVs $\mathbf{I}$ end up in a single parcluster, LDJT uses that LJT constructs a valid FO jtree. In a valid FO jtree, it has to hold that for each parfactor, of the underlying PM, its PRVs must be contained in at least one parcluster of the resulting FO jtree. Thus, by having a parfactor with $\mathbf{I}$ as arguments in a PM, LJT constructs an FO jtree with at least one parcluster that contains $\mathbf{I}$. LDJT adds g_0^I to G_0 and constructs an FO jtree J_0. As the structure J_0 is a valid FO jtree, at least one parcluster contains at least all PRVs of g_0^I. Further, LDJT adds g_{t-1}^I and g_t^I to $G_{\rightarrow}$ and constructs an FO jtree J_t. Hence, one parcluster contains all PRVs of g_{t-1}^I and one parcluster contains all PRVs of g_t^I. Therefore, LDJT constructs FO jtree structures that obey the interface idea. □

For the FO jtree construction of LDJT, we additionally need to show that the FO jtree structures are still valid after LDJT prevents unnecessary groundings.

Theorem 4.1.2. *The* extension *step of LDJT is sound, i.e., yields a valid FO jtree.*

Proof. In the FO jtree structures before *extension*, the separator between FO jtree J_{t-1} and J_t consists of exactly $\mathbf{I}_{t-1}$. Thus, by taking the intersection of the PRVs in J_{t-1} and J_t, we get the set of PRVs from $\mathbf{I}_{t-1}$. While LDJT calculates α_{t-1}, it only needs to eliminate PRVs $\mathbf{A}$ not contained in the separator $\mathbf{I}_{t-1}$. Therefore, none of $A \in \mathbf{A}$ is contained in any parcluster of J_t. Hence, by adding A to the *in-cluster* of J_t, LDJT does not violate any FO jtree properties. After *extension*, the resulting FO jtree structures still fulfil all FO jtree properties. The running intersection property still holds, as *extension* only adds PRVs, which previously were not included in the corresponding FO jtrees structure. Additionally, semantically a PDM corresponds to the unrolled ground model. Therefore, all added PRVs are included in the model and we do not change the assigned parfactors. Thus, all FO jtree properties still hold. Further, as we do not change the assigned parfactors, the full joint distribution of the model remains the same. The proof also holds in the other direction, to prevent groundings during a backward pass □

Using the FO jtree structures, we now show that the forward pass of LDJT is sound, leading to sound *filtering* and *prediction* query answering.

Theorem 4.1.3. *LDJT is sound regarding* filtering *and* prediction *queries, i.e., LDJT produces the same results as LJT does for an unrolled PDM.*

Proof. The interface PRVs m-separate time steps for a given PDM. Further, LDJT ensures that the FO jtrees for the initial time step and the copy pattern have an *in-cluster* and an *out-cluster* consisting of interface PRVs. 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 and model behaviour including 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. As each α_t message is equivalent to calculating a query over the interface PRVs and LDJT performs sound operations of LJT within a time step, the forward pass of LDJT is sound. □

Finally, we show that also the backward pass of LDJT is sound.

Theorem 4.1.4. *LDJT is sound regarding* hindsight *queries, i.e., LDJT produces the same results as LJT does for an unrolled PDM.*

Proof. LDJT enters evidence for time step t into FO jtree J_t. Additionally, the α_{t-1} message describes the influences of all evidence and the model behaviour up until $t-1$.

Thus, each FO jtree contains evidence up to the time step the FO jtree is instantiated for. During a backward pass, LDJT distributes state descriptions from J_t backwards. Therefore, LDJT performs an *inter* FO jtrees backward message pass over the interface separator. The β_t message is correct, since all necessary state descriptions are present and accounted for while calculating the β_t message. The β_t message, which LDJT adds to J_{t-1} as well as the α_{t-1} message, are then accounted for during the message pass inside J_{t-1}. 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 $t = 0$.

Further, LDJT constructs valid and sound FO jtree structures. The FO jtree structures can be unrolled, instead of unrolling the underlying PDM. Providing LJT with the sound unrolled FO jtree, LJT would compute the very same α and β messages, as well as all other messages. Hence, LJT produces the very same results as LDJT. Thus, as LJT is sound, the calculations of LDJT are also sound. □

Having shown that LDJT is sound, we now investigate for which models LDJT is complete that is it answers queries without grounding.

4.2 Completeness

In the following, we show completeness results for LDJT. Taghipour *et al.* (2013d) show that LVE with generalised counting is complete for 2-logvar models and Braun (2020) show that LJT is complete for 2-logvar models. The same completeness results also hold for other exact static lifted inference algorithms (Van den Broeck, 2011). A 2-logvar model has at most two logvars in each parfactor. In general, with completeness, one can classify the models for which a probabilistic inference algorithm runs in polynomial time w.r.t. the domain size.

Definition 4.2.1 (2-logvar models)**.** Let $\mathcal{M}^{2lv}$ be the model class of 2-logvar models.

Theorem 4.2.1. *LVE and LJT are complete for PDMs that are in* $\mathcal{M}^{2lv}$.

Proof. LVE and LJT are complete for $\mathcal{M}^{2lv}$ and if a PDM G is a from $\mathcal{M}^{2lv}$, then the unrolled version of G is also from $\mathcal{M}^{2lv}$ and thus, LVE and LJT are complete for G. Therefore, LVE and LJT answer queries in polynomial time w.r.t. the domain size by computing a lifted solution. □

Theorem 4.2.2. *LDJT is not complete for all models in* $\mathcal{M}^{2lv}$.

Proof. Example 3.4.3 shows a model from $\mathcal{M}^{2lv}$ and LDJT does not guarantee a lifted solution for that model. Hence, LDJT is not complete for all models in $\mathcal{M}^{2lv}$.

With the counterexample from Example 3.4.3, we have shown that LDJT is not complete for all 2-logvar models. Similar counter examples can also be build with other inter-slice parfactors. For example with the inter-slice parfactors

$$g^P = \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \phi^P(DoR_t(x), Pub_{t+1}(x, j))_{|\top}$$
$$g^D = \forall j, x \in \mathcal{D}(J) \times \mathcal{D}(X) : \phi^D(Pub_t(x, j), DoR_{t+1}(x))_{|\top},$$

we can construct a similar counterexample. The main problem here is that in the inter-slice parfactors there is at least one PRV with two logvars for time slice t and at least one PRV with two logvars for time slice $t + 1$. Such a pattern leads to the fact that we cannot eliminate PRVs with fewer logvars without eliminating PRVs with two logvars first. □

By unrolling the corresponding model and using LJT, it builds a parcluster containing the PRV $Pub_t(X, J)$ for all time steps. Most likely, the FO jtree consists of very few parclusters, which basically results in performing LVE on the unrolled model. Further, by clustering a PRV for all time steps in one parcluster, the model is not time-separated anymore. We also could adjust the *extension* of LDJT to allow for such parclusters and therefore, be also complete for 2-logvar models. However, with LDJT, we aim at handling temporal aspects efficiently, which is not given anymore by performing LVE on the unrolled model. Therefore, we trade in completeness to handle temporal aspects efficiently. Later on, we also empirically analyse the trade off.

Theorem 4.2.3. *LDJT is complete for models from $\mathcal{M}^{2lv}$ with inter-slice parfactors that do not have PRVs with two logvars for time slice t and $t + 1$.*

Proof. For the proof, we consider the three cases that remain, namely:

i) Only PRVs with at most one logvar in inter-slice parfactors,

ii) only PRVs with two logvars for time slice t in inter-slice parfactors, and

iii) only PRVs with two logvars for time slice $t + 1$ in inter-slice parfactors,

Case i) means that an *out-cluster* can have PRVs with two logvars, but all of them can be eliminated at the *out-cluster*. Therefore, there cannot be any algorithm-induced groundings while calculating an α message as Eq. (2.3) holds. Additionally, the same argumentation also holds for β messages during backward passes with an *in-cluster*. Inside a time step, LJT ensures that it is complete for 2-logvar models. Thus, LDJT is complete for case i).

Case ii) means that at least one PRV with two logvars is in the interface. Therefore, *out-clusters* and *in-clusters* have at least one PRV p with two logvars. As trying to eliminate the non-interface PRVs could lead to count-converting the two logvars of p,

these count-conversions then could lead to groundings in an *in-cluster*. Therefore, all descriptions about PRVs from an *out-cluster* need to be sent to an *in-cluster*. However, between an *in-cluster* and an *out-cluster*, LDJT can eliminate p and afterwards the remaining PRVs from time-slice t: In the inter-slice parfactors, p only occurs for time slice t, but not for time-slice $t+1$. On the path from the *in-cluster* to the *out-cluster*, α_t is multiplied with the inter-slice parfactors. Multiplying α_t with the inter-slice parfactors ensures that LDJT can eliminate p. Further, p cannot occur for time slice $t+1$ in the inter-slice parfactors. Thus, LDJT can eliminate the remaining PRVs from time-slice t with generalised counting. Hence, all PRVs that need to be eliminated for α_{t+1} can be eliminated using lifted operations. The argumentation is valid for a forward pass as well as for a backward pass. Hence, LDJT is complete for case ii).

Case iii) is similar to case i). To calculate temporal messages there are no algorithm-induced groundings. The difference is that on the path from an *in-cluster* to an *out-cluster*, LJT might need to prevent algorithm-induced groundings by fusion, which in the worst case leads to merging *in-cluster* and *out-cluster*. However, when calculating a temporal message, LDJT eliminates all two logvar PRVs, and then generalised counting ensures that LDJT does not have to ground. Therefore, LDJT is complete for all three cases, which, in turn, means that LDJT is complete for 2-logvar models with inter-slice parfactors that do not have PRVs with two logvars for time slice t and $t+1$. □

Taghipour (2013, Section 6.7) conjectures that one could easily generalise the counting operation to also allow counting of multiple logvars. Thus, from a theoretical point of view, by generalising the counting operation even further, one could solve our grounding problem, making LDJT also complete for all 2-logvar models. However, Taghipour (2013, Section 6.7) also mentions that additional research on this counting problem is needed.

Definition 4.2.2 (1-logvar PRV models)**.** Let $\mathcal{M}^{1prv}$ be the model class where each PRV has at most 1 logvar.

Corollary 4.2.1. *LDJT is complete for* $\mathcal{M}^{1prv}$*.*

Proof. The proof directly follows from Thm. 4.2.3 and Taghipour (2013, Thm. 7.2). □

In general, completeness results for relational inference algorithms assume liftable evidence. In case evidence breaks symmetries, query answering might not run in polynomial time but in exponential time w.r.t. domain sizes. Also, even if an algorithm is not complete for a certain class, the algorithm might still be able compute a lifted solution for some models of that class. For example, LDJT calculates a lifted solution for the model in Example 3.4.1, even though LDJT is not complete for 3-logvar models.

Next, we have a look at the complexity of LDJT.

4.3 Complexity

For the runtime complexity of LDJT, we first present the complexity of LJT based on Braun (2020). Additionally, LDJT also has different space complexities, depending on the instantiation approach, which in turn also has an influence on the runtime complexity.

4.3.1 LJT

The complexity of LJT depends on an FO jtree constructed from a corresponding PM G. In the propositional case (Darwiche, 2009), the runtime complexity of a jtree corresponds to the number of randvars in the cluster with the most randvars, also called *ground width*. We define the *lifted width* of an FO jtree (Braun, 2020) and recapitulate the complexity of each step of LJT as well as the overall complexity of LJT.

Definition 4.3.1. The *lifted width* w_J is a pair $(w_g, w_\#)$, where w_g is the largest number of PRVs in any parcluster of J and $w_\#$ is the largest number of CRVs in any parcluster of J.

Further, n is the largest domain size among $lv(G)$, $n_\#$ is the largest domain size of the counted logvars, r is the largest range size in a G, $r_\#$ is the largest range size among the PRVs in the CRVs, and n_J being the number of nodes in J. The largest possible factor in J is given by $r^{w_g} \cdot n_\#^{w_\# \cdot r_\#}$.

Let us now go through the steps of LJT. The steps that make up the overall complexity of LJT are FO jtree construction, evidence entering, message passing, and query answering. The effort of FO jtree construction is negligible compared to the other runtime complexities, as it only depends on an intermediate representation to construct the FO jtree as well as w_g and $w_\#$, but not the ranges.

Evidence entering consists of absorbing evidence at each applicable node.

Lemma 4.3.1. *The complexity of absorbing an evidence parfactor is*

$$O(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_\#^{w_\# \cdot r_\#}). \tag{4.1}$$

Passing messages consists of calculating messages with LVE.

Lemma 4.3.2. *The complexity of passing messages is*

$$O(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_\#^{w_\# \cdot r_\#}). \tag{4.2}$$

The last step is *query answering*, which consists of finding a parcluster and answering a query on an assembled submodel.

Lemma 4.3.3. *The complexity of answering a set of ground queries* $\{Q_k\}_{k=1}^m$ *is*

$$O(m \cdot \log_2(n) \cdot r^{w_g} \cdot n_\#^{w_\# \cdot r_\#}). \tag{4.3}$$

We now combine the stepwise complexities to present the complexity of LJT by adding up the complexities in Eqs. (4.1) to (4.3).

Theorem 4.3.1. *The complexity of LJT is*

$$O((n_J + m) \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.4}$$

Knowing the complexity of LJT, we now investigate the complexity of LDJT.

4.3.2 LDJT

For LDJT, the complexity of the FO jtree construction is also negligible. Compared to LJT, the complexity of the FO jtree construction of LDJT only differs in constant factors. The construction of the FO jtree structures J_0 and J_t is actually the very same as for LJT. The difference is that two FO jtrees are constructed. For *extension*, LDJT additionally performs checks on two messages and twice the *fusion* step of LJT. The additional checks are constant factors and therefore, do not change the complexity of the FO jtree construction.

For the complexity analysis of LDJT, we assume that the FO jtree structures of LDJT are minimal and do not induce groundings. Further, we slightly change the definition from Definition 4.3.1, as we now consider a PDM G and two FO jtrees, J_0 and J_t.

Definition 4.3.2. Let $w_{J_0} = (w_g^0, w_{\#}^0)$ be the *lifted width* of J_0 and let $w_{J_t} = (w_g^t, w_{\#}^t)$ be the *lifted width* of J_t. The *lifted width* w_J of a pair (J_0, J_t) is a pair $(w_g, w_{\#})$, where $w_g = \max(w_g^0, w_g^t)$ and $w_{\#} = \max(w_{\#}^0, w_{\#}^t)$.

Further, T is the maximum number of time steps, n is the largest domain size among $lv(G)$, $n_{\#}$ is the largest domain size of the counted logvars, r is the largest range size in G, $r_{\#}$ is the largest range size among the PRVs in the CRVs, and n_J being the $max(n_{j_0}, n_{j_t})$. The largest possible factor is given by $r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}$. Hence, we always look at the highest number that occurs either in J_0 or J_t.

Evidence entering consists of absorbing evidence at each applicable node.

Lemma 4.3.4. *The complexity of absorbing an evidence parfactor is*

$$O(T \cdot n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.5}$$

The difference to LJT and Eq. (4.1) is that LDJT does not only enter evidence in each parcluster of a FO jtree, but enters evidence in each instantiated FO jtree. Overall, there is evidence for up to T time steps. Therefore, LDJT enters evidence in T FO jtrees.

Passing messages consists of calculating messages with LJT for every time step. Here, we consider the worst case, i.e., for each time step querying the first and last time step, the average case, i.e., *hindsight* and *prediction* queries with a constant offset, and the best case, i.e., only *filtering* queries.

Lemma 4.3.5. *The worst case complexity of passing messages is*

$$O(T^2 \cdot n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.6}$$

The average case complexity of passing messages is

$$O(T \cdot n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.7}$$

The best case complexity of passing messages is

$$O(T \cdot n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.8}$$

Equation (4.2) shows the complexity of one complete message pass in an FO jtree. The message pass consists of calculating $2 \cdot (n_J - 1)$ messages and each message has a complexity of $O(\log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$. One difference in LDJT compared to LJT is that LDJT needs to calculate $2 \cdot (n_J - 1) + 2$ messages for the current FO jtree, because LDJT calculates an α and a β message in addition to the normal message pass. For theFO jtree used to answer *prediction* or *hindsight* queries, LDJT calculates $2 \cdot (n_J - 1) + 1$ messages, as LDJT calculates either an α or β message respectively. Additionally, LDJT computes at least one message pass for each time step and at most a message pass for all time steps for each time step. Therefore, we investigate the worst, average, and best case complexity of message passing in LDJT.

The worst case for LDJT is that for each time step there is a query for the first and the last time step. Therefore, for each of the T time steps, LDJT would need to perform a message pass in all T FO jtrees, leading to $T \cdot T$ message passes. Hence, LDJT would perform a message pass for the current time step t, a backward pass from t to the first time step, which includes a message pass on each FO jtree on the path, and a forward pass from t to the last time step, which include a message pass on each FO jtree on the path. These message passes are then executed for each time step. Thus, LDJT performs overall $T \cdot T$ message passes. The complexity of Eq. (4.6) is also the complexity of LJT given an unrolled FO jtree constructed by LDJT and evidence for each time step. However, the complexity of message passing for LDJT is normally much lower, as one is hardly ever interested in always querying the first and the last time step.

The best case for LDJT is that it only needs to answer *filtering* queries. That it to say it needs to calculate $2 \cdot (n_J - 1) + 1$ messages for each FO jtree, as LDJT calculates an α for each FO jtree. Further, LDJT needs to perform exactly one message pass on each instantiated FO jtree. Therefore, LDJT needs to pass messages on T FO jtrees.

The average case for LDJT is that for each time step LDJT answers a constant number of *hindsight* and *prediction* queries. Assume for each time step, LDJT has a query for $t - 10$ and $t + 15$. Then, LDJT needs to perform 25 message passes to answer all queries for one time step. Therefore, LDJT passes messages $25 \cdot T$ times. In general, *prediction* and *hindsight* queries are often close to the current time step and T can be huge.

Under the presence of *prediction* and *hindsight* queries, LDJT does not always need to calculate $2 \cdot (n_J - 1)$ messages for each FO jtree. In case LDJT has no query for time step t, but only needs J_t to calculate an α or β message, then calculating $(n_J - 1)$ messages suffice for J_t. By selecting the *out-cluster* for a *prediction* queries and respectively the *in-cluster* for a *hindsight* queries as root, then all required messages are present at the *out-cluster* or the *in-cluster* to calculate an α or β message. Hence, an efficient query answering plan can reduce the complexity of message passing with constant factors.

The last step is *query answering*, which consists of finding a parcluster and answering a query on an assembled submodel. For query answering, we combine all queries in one set instead of having a set of queries for each time step.

Lemma 4.3.6. *The complexity of answering a set of queries* $\{Q_k\}_{k=1}^{m}$ *is*

$$O(m \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.9}$$

The complexity for query answering in LDJT does not differ from the complexity of LJT. Nonetheless, the number of queries m for LDJT is often higher than the number of queries m for LJT.

We now combine the stepwise complexities to arrive at the complexity of LDJT by adding up the complexities in Eqs. (4.5) to (4.9).

Theorem 4.3.2. *The worst case complexity of LDJT is*

$$O(((T^2 + T) \cdot n_J + m) \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.10}$$

The average case complexity of LDJT is

$$O((T \cdot n_J + m) \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.11}$$

The best case complexity of LDJT is

$$O((T \cdot n_J + m) \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{4.12}$$

4.3.3 Comparison to the Ground Interface Algorithm

In this section, we show that lifting is crucial when it comes to temporal models, where multiple instances influence the next time step. Therefore, we show that the complexity of the ground interface algorithm is exponential in the number of instances influencing the next time step, while the runtime complexity of LDJT is independent of the domain sizes (except for the $log_2(n)$ part). The implication of this result is that even for small domain sizes lifting is necessary, otherwise, the algorithm becomes infeasible.

For LJT compared to a ground junction tree algorithm, the speed up is twofold. The first speed up is that LJT has fewer nodes in an FO jtree than the corresponding ground jtree has, i.e., $n_{gr(J)} >> n_J$. The other speed up originates from the counted part,

$n_{\#}^{w_{\#} \cdot r_{\#}}$. Under the presence of counting, the lifted width is smaller than the ground width. Further, Taghipour *et al.* (2013a) shows that in models that do not require count-conversions, the lifted width is equal to the ground width. Therefore, the factor r^{w_g} of the complexity of LJT is the same as the ground width of a junction tree algorithm without count-conversions.

From a complexity perspective, LDJT has another advantage over the interface algorithm. The w_g part of lifted width w_J from LDJT is much lower compared to the ground tree width of the interface algorithm. The interface algorithm ensures that all randvars that have successors in the next time step are grouped in one cluster of an jtree. Therefore, the corresponding jtree has at least $|gr(\mathbf{I}_t)|$ randvars in a cluster. Thus, the ground width w_g of the interface algorithm depends on the domain sizes of the interface PRVs. The lifted width w_J with its part w_g of LDJT is independent of the domain sizes of the interface PRVs. Hence, the lifted width, even without count-conversions, is much smaller compared to the ground width. Therefore, LDJT can answer queries for large domain sizes, which can be infeasible for the interface algorithm.

Overall, we can show that LDJT handles temporal models also with large domain sizes. Thus, lifting greatly matters and already for small domain sizes LDJT with all its parts, including ensuring preconditions of lifting, is necessary to compute solutions for models. For large domain sizes, the ground interface algorithm becomes infeasible, while LDJT can obtain a solution with a low runtime complexity.

4.3.4 Space and Time Requirements of Different Query Answering Plan

Now, we look at the space and time requirements of different query answering plans.

Space Requirements The local models of parclusters have the biggest impact on the memory consumption of an FO jtree. They encode all necessary state descriptions to answer queries with a parcluster. Now, we investigate the number of rows stored as our space requirement. The space requirement of LDJT is determined by the maximum number of rows for all parfactors and the maximum number of rows for all messages, including α and β messages. In the best case, the number of rows stored in an FO jtree is $R = r^l \cdot m + r^{k-1} \cdot (2 \cdot n_J)$, where r is the largest possible range, l is the maximum number of PRVs in a parfactor, m is the number of parfactors, k is the maximum number of PRVs in a parcluster, n_J is the number of parclusters. Roughly speaking, $r^l \cdot m$ denotes the size of a model and $dot(2 \cdot n_J)$ the size of messages. We look at the best case as we do not include evidence and assume that LDJT calculates the messages solely using lifted summing out, which means that no count-conversions or groundings occur.

Each parfactor has at most r^l rows and LDJT assigns exactly m parfactors in the local models. Thus, $r^l \cdot m$ provides the maximum number of rows for the local model. As the PRVs in different parclusters cannot be completely overlapping, a message contains at most $k - 1$ PRVs. During a complete *intra* FO jtree message pass, LDJT calculates

$2 \cdot (n_J - 1)$ messages. Additionally, LDJT calculates 2 messages (α and β) during an *inter* FO jtree message pass. Thus, each message has at most r^{k-1} rows, and $2 \cdot n_J$ messages are calculated.

If LDJT preserves all FO jtrees instantiated, the space requirement is $T \cdot R$, where T is the maximum number of time steps. In case LDJT instantiates FO jtrees on-demand, the space requirement is $R + (T-1) \cdot a$, where a is the size of an α message, which is ${r_i}^i$, where i is the number of PRVs in the interface and r_i is the maximum number of possible values of a PRV in $\mathbf{I}$. Thus, the space requirements for instantiating FO jtrees on-demand is composed of the rows for the current FO jtree and all previous α messages.

Now, we are interested in the difference in terms of space requirements of the two approaches. $R - a$ provides the space requirement difference for one time step, but LDJT needs to store $R - a$ many rows for $T - 1$ time steps, resulting in a space requirements difference of $(T-1) \cdot (R-a)$ rows. $R-a$ is $r^l \cdot m + r^{k-1} \cdot (2 \cdot n_J - 1)$ as we subtract exactly one message. Thus, the space requirements for instantiating FO jtrees on-demand is much lower than preserving all instantiated FO jtrees, as we only need to store one message, instead of all messages and parfactors, for each time step. Thus, due to the much lower space consumption of instantiating FO jtrees, LDJT can in principle answer *hindsight* queries with huge lags.

Time Requirements To answer queries, LDJT needs to enter evidence again and perform a complete message pass in case LDJT instantiates an FO jtree. Therefore, LDJT would not only perform T times evidence entering, but if we always have hindsight queries with a lag of 10, then LDJT would need to perform $10 \cdot T$ times the evidence entering. If LDJT answers queries on a preserved FO jtree, LDJT does not enter evidence again, resulting in T times the evidence entering, but LDJT only needs to account for the new β message. Therefore, an *out-bound* pass with the *out-cluster* as root suffices, resulting in $n_J - 1$ messages, to be able to answer queries. Additionally, LDJT can also only calculate a new β message if no queries are asked for the given time step. To calculate a β message using a preserved FO jtree, LDJT only needs to propagate messages from the *out-cluster* to the *in-cluster*. To calculate a β message with instantiating an FO jtree on demand, LDJT performs an *inbound* pass with the *in-cluster* as root.

The space gain comes at a cost of redoing calculations. Further, to answer *hindsight* queries, LDJT has to ensure that the necessary state descriptions are available at the corresponding parcluster for each time step inside the lag.

Preserving a limited number of FO jtrees is advantageous for fast query answering. However, as preserving FO jtrees requires significantly more memory, LDJT cannot preserve all FO jtrees. To allow *hindsight* queries with a huge lag, LDJT has to instantiate FO jtrees on demand and thereby redo computations.

Chapter 5

Evaluation

This chapter empirically investigates LDJT to support claims of the theoretical analysis. In the evaluation, we compare LDJT to DJT, LJT with an unrolled FO jtree, LJT with an unrolled model, and UUMLN (Geier and Biundo, 2011). DJT is the interface algorithm, as explained in Section 2.2, and UUMLN is an approach for dynamic Markov logic network (DMLN) and therefore temporal probabilistic relational models (available here `https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.090/Software/slice-dmlns-V1.0.zip`). Further, Papai *et al.* (2012) propose another approach for DMLN, for which we could not find an implementation. But, Papai *et al.* claim that their runtimes are equal to UUMLN. However, to the best of our knowledge, UUMLN always answers queries for all ground instances and does not employ lifting techniques for their calculations. For the evaluation, we use variations of G^{ex} to support claims of the theoretical analysis. There are no default data sets for temporal probabilistic relational models. The existing data sets for probabilistic relational models include less PRVs than our example and only result in one parcluster. Thus, to efficiently evaluate LDJT, we use variations of G^{ex}. We run all experiments on a virtual machine with 16 GB of RAM.

In the theoretical analysis, we make at least the following claims:

- LDJT runs in at most polynomial time w.r.t. domain sizes given a lifted computation is achieved.
- In the best and average case, LDJT runs in linear time w.r.t. the maximum number of time steps.
- In the worst case, LDJT runs in quadratic time w.r.t. the maximum number of time steps, which is the same as unrolling the FO jtrees and using LJT.
- It is not always feasible to keep all FO jtrees in memory.

Based on these claims, in the empirical evaluation we run the following experiments:

(i) Filtering Queries:
   - Does LDJT run in linear time w.r.t. the maximum number of time steps?

- How does the domain size influence runtimes?
- How does the number of queries influence runtimes?
- How does LDJT compare to approximate approaches such as UUMLN (Geier and Biundo, 2011) from a run time perspective?

(ii) Prediction and Hindsight Queries:

- Does LDJT run in linear time w.r.t. the maximum number of time steps?
- How do *prediction* and *hindsight* queries influence runtimes?
- Are the runtimes of LDJT, with always a prediction query to the last time step and a hindsight query to the first time step, bounded by unrolling an FO jtree and using LJT?
- Is it feasible to keep all FO jtrees in memory?
- Can LDJT answer hindsight queries with huge lags?

(iii) Count Conversions:

- Does LDJT run in linear time w.r.t. the maximum number of time steps?
- How does the domain size influence runtimes?

(iv) Preventing Groundings:

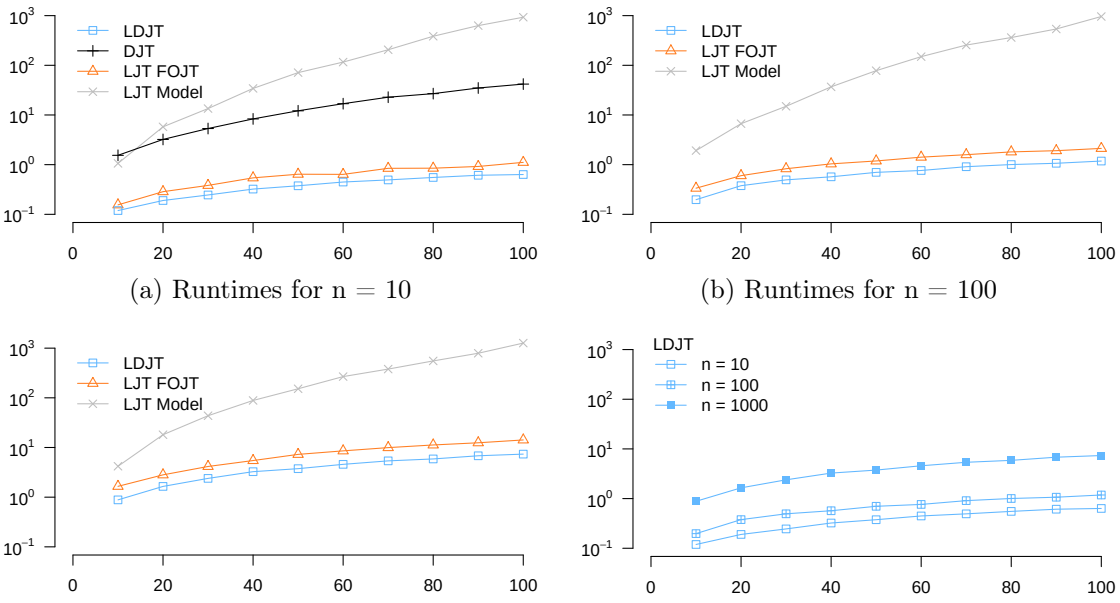
- Does the effort to prevent groundings pay off?
- Is it sometimes better to unroll the model and obtain a lifted solution compared to handling temporal aspects efficiently and suffer groundings?

(v) Evidence:

- How does the domain size influence runtimes?
- How does the number of symmetry groups influence runtimes?

5.1 Filtering Queries

For *filtering* queries, we use a variation of G^{ex}, with the main difference that now also the PRV Hot is parameterised with X to have the biggest impact on the model while increasing the domain size of X. By only having *filtering* queries, we have the best case for LDJT from a complexity perspective w.r.t. message passes. To investigate whether LDJT runs in linear time w.r.t. the maximum number of time steps and how domain sizes influence runtimes, we evaluate LDJT, DJT, LJT with an unrolled FO jtree, and LJT with an unrolled model by asking *filtering* queries on representatives of each PRV. Further, to compare these approaches to UUMLN, we also ask queries for each instance.



(a) Runtimes for n = 10

(b) Runtimes for n = 100

(c) Runtimes for n = 1000

(d) Runtimes for LDJT

Figure 5.1: Filtering queries for one representative and all PRVs, y-axis: runtimes [seconds, log], x-axis: time steps

Querying Representatives To investigate whether LDJT runs in linear time w.r.t. the maximum number of time steps, we compare runtimes of LDJT for different maximum number of time steps T. To investigate the influence of domain sizes, we evaluate LDJT by fixing the domain sizes for all but one logvar and increase the domain size for this logvar. For this part of the evaluation, we vary T between 10 and 100, always increasing it by 10, i.e., we ask filtering queries for each PRV and time step for $T = 10$, afterwards for $T = 20$, and so on. Further, we vary the domain size for the logvar X, i.e., $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$, while leaving $|\mathcal{D}(P)| = 3$ unchanged. For the runtimes of LJT with an unrolled FO jtree and LJT with an unrolled model, we only perform one message pass instead of T message passes, which would correspond to having the model and getting queries and evidence for one time step after the other. One message pass for LJT represents the best case possible for LJT. But With one message pass, LJT also answers slightly different queries, i.e., only *filtering* queries for a single time step and the remaining time steps *hindsight* or *prediction* queries. Figures 5.1a to 5.1d show the results of the evaluation.

Figure 5.1a shows the runtimes of LDJT, DJT, LJT with an unrolled FO jtree, and LJT with an unrolled model for $|\mathcal{D}(X)| = 10$. By comparing LDJT and DJT, we can see that being able to calculate a lifted solution is much faster. The speed up of more than an order of magnitude can be explained by the fact that DJT needs to perform

each operation for each ground instance and, as shown in Section 4.3.3, the interface is also dependent on the domain size. The runtimes of LJT with an unrolled model are not linear, but at least polynomial in T. Here, LJT builds an FO jtree based on the unrolled model and the constructed FO jtree is not optimised for temporal models. In this case, by increasing T, also the lifted tree width w_J increases as well as the number of parclusters n_J as the number of PRVs increases with T. Hence, the runtimes of LJT with an unrolled model are not linear w.r.t. T. Therefore, by not handling temporal aspects, LJT with an unrolled model is much slower compared to LDJT. LJT with an unrolled FO jtree is only slightly slower compared to LDJT. Here, LJT unrolls the FO jtree structures of LDJT. Thus, LJT uses an FO jtree optimised for the temporal case. However, LJT with an unrolled FO jtree is still slower than LDJT. LJT with an unrolled FO jtree performs a complete message pass, i.e., LJT sends messages from the first time step to the last time step and back again. Even though LDJT also performs a complete message pass inside one FO jtree, LDJT only sends messages from the first time step to the last time step and therefore computes fewer messages. The difference is in calculating β messages. The speedup mostly originates from calculating fewer messages, as LDJT and LJT with an unrolled FO jtree can use the very same parcluster to answer the queries. Another speed up of LDJT is that LJT has a bigger search space to find the corresponding parcluster. Nonetheless, with only one message pass LJT only answers *filtering* queries for a single time step and the queries for the remaining time steps are either *hindsight* or *prediction* queries. To also answer all queries as *filtering* query, LJT would need to perform T message passes, which would significantly increase runtimes. Nonetheless, LDJT already outperforms the best case for LJT making the need for temporal approaches apparent.

Figure 5.1b depicts the runtimes of LDJT, LJT with an unrolled FO jtree, LJT with an unrolled model for $|\mathcal{D}(X)| = 100$ and the runtime of the approaches for $|\mathcal{D}(X)| = 1000$ are shown in Figure 5.1c. Here, we can see the same behaviour as for $|\mathcal{D}(X)| = 10$. The main differences are that our implementation DJT could not handle the higher domain sizes and that the runtimes of the other approaches are slightly elevated.

So far, the questions whether LDJT runs in linear time w.r.t. the maximum number of time steps and how the domain size influences runtimes are still open. Figure 5.1d shows

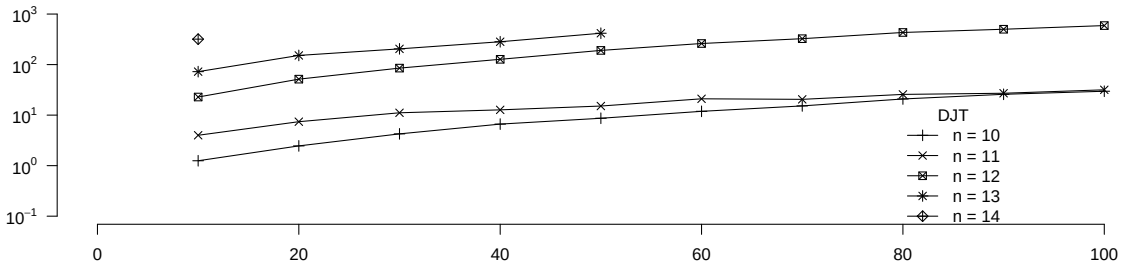


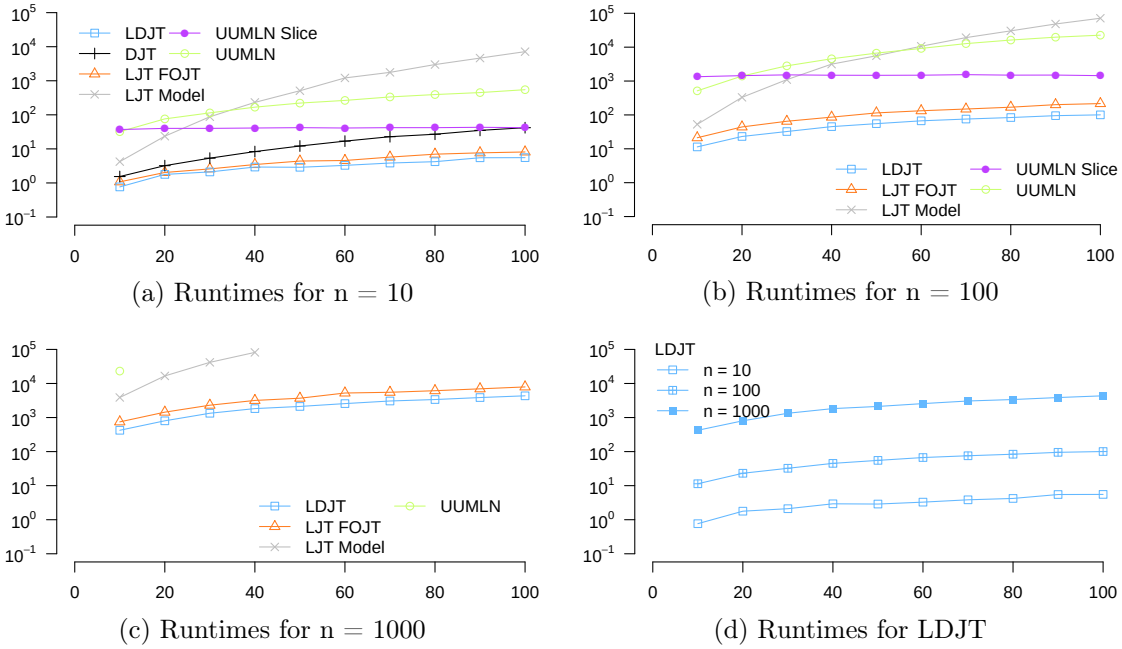
Figure 5.2: Runtimes for DJT, y-axis: runtimes [seconds, log], x-axis: time steps

the runtimes of LDJT, for $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$. We can see that LDJT runs in linear time w.r.t. T for filtering queries independent of the domain sizes. Further, we can see that the domain sizes have a small impact on the runtimes. For example from $|\mathcal{D}(X)| = 10$ to $|\mathcal{D}(X)| = 100$, LDJT needs about twice as long. Hence, by having 10 times as many instances for X the runtimes only multiply by 2. Comparing $|\mathcal{D}(X)| = 10$ to $|\mathcal{D}(X)| = 1000$, the runtimes take about an order of magnitude longer. This can be explained with the $\log_2 n$ in the complexity of LDJT as with an increasing domain size the exponent for indistinguishable instances also increases.

Figure 5.2 shows the runtimes of DJT for increasing domain sizes. Here, the runtimes drastically increase for slight increments, making the need for a lifted solution apparent.

Querying Each Instance Let us now compare LDJT to UUMLN (Geier and Biundo, 2011). We use the very same setting as for querying representatives. The difference is that we now query $gr(rv(G^{ex}))$ for each time step as UUMLN does not support asking specific queries, but always answers queries for each grounding. With a lifted algorithm, such as LJT and LDJT, querying a representative suffices, because all other instances of that group behave exactly the same and thus, have identical marginal distribution. However, by increasing the number of instances in the domain of the logvar X, we now also increase the number of queries. Hence, we can also show the influence of the number of queries on the runtimes. Overall, UUMLN is an approach for DMLN, which to the best of our knowledge does not employ any lifting techniques. For UUMLN, Geier and Biundo use the expanding frontier belief propagation (Nath and Domingos, 2010b), which performs adaptive inference for changing evidence. Nath and Domingos (2010a) also propose a similar solution using lifting, but that is to the best of our knowledge not incorporated in UUMLN. Papai *et al.* (2012) compare their approximate approach against UUMLN showing that they reach the same runtimes but have a different level of accuracy on their test set. As LDJT is an exact algorithm, we are interested how fast LDJT is compared to UUMLN. Thus, is a lifted exact algorithm faster than an approximate ground algorithm? The results of the evaluation are shown in Figs. 5.3a to 5.3d, with a limit of 1 day for the runtimes.

In addition to LDJT, DJT, LJT with an unrolled FO jtree, and LJT with an unrolled model, we also use UUMLN with Gibbs sampling and UUMLN with time sliced Gibbs sampling for this evaluation. For $|\mathcal{D}(X)| = 10$, the runtimes are depicted in Fig. 5.3a. For LDJT, DJT, LJT with an unrolled FO jtree, and LJT with an unrolled model, we can see a similar behaviour as shown in Fig. 5.1a. The difference is that they all take slightly longer as they need to answer 60 instead of 4 queries for each time step. Additionally, we can see that LDJT is up to two magnitudes faster than UUMLN. Thus, LDJT lifts indistinguishable instances makes it feasible to calculate exact solutions instead of having to approximate with an unknown and possbile unbounded error. UUMLN with the time slice option seems to be independent of the maximum number of time steps. To the



(a) Runtimes for n = 10

(b) Runtimes for n = 100

(c) Runtimes for n = 1000

(d) Runtimes for LDJT

Figure 5.3: Filtering queries for all instances and all PRVs, y-axis: runtimes [seconds, log], x-axis: time steps

best of our understanding, they adapt to new evidence. Hence, the approach is time independent as we have not included evidence in this evaluation.

Figure 5.3b shows the runtimes for $|\mathcal{D}(X)| = 100$ and the runtimes for $|\mathcal{D}(X)| = 1000$ are depicted in Fig. 5.3c. Again, DJT could not perform inference on these large domains. We can see similar trends between Fig. 5.3a and Fig. 5.1a, i.e., the runtimes are only elevated as they need to answer more queries. For $|\mathcal{D}(X)| = 1000$, UUMLN could only produce a result for 10 time steps and UUMLN with the slice option could not produce any results at all within a day of calculations. Further, LJT with an unrolled model, also could only produce results for up to 40 time steps within a day of calculations.

The runtimes of LDJT for different domain sizes are depicted in Fig. 5.3d. We can see that increasing the domain sizes leads to an increase of about an order of magnitude for the runtimes. The runtimes increase as much because we did not only increase the domain sizes by a factor of 10, but also have 10 times as many queries for each time step. There is a linear dependency of the number of queries and the runtimes as the number of queries becomes the dominate factor in the runtime complexity.

Overall, to answer the questions for *filtering* queries, we can say that:

- LDJT runs in linear time w.r.t. the maximum number of time steps for *filtering* queries.

- With a lifted solution and without any count-conversions, increasing domain sizes only increases runtimes by a logarithmic factor.
- For each query, LDJT needs to eliminate the non-query terms from a corresponding parcluster. Therefore, increasing the number of queries also increases the runtimes by a linear factor.
- LDJT can find exact solutions to queries orders of magnitudes faster compared to UUMLN calculating only approximate solutions.

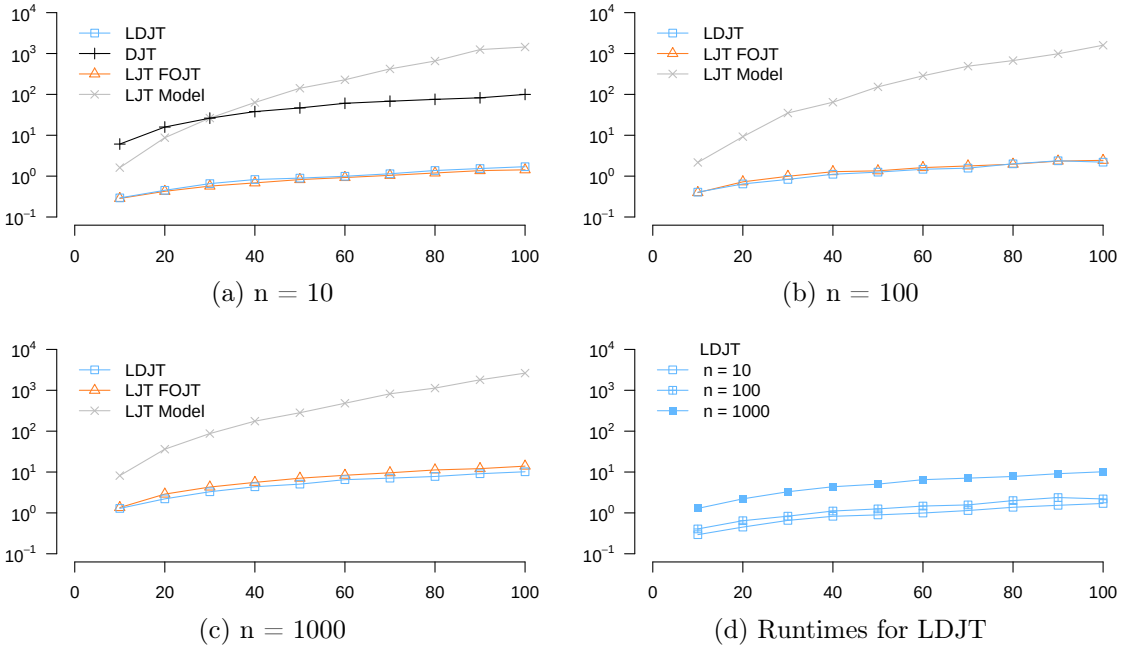
5.2 Prediction and Hindsight Queries

Next, we evaluate *prediction* and *hindsight* queries instead of only *filtering* queries. For this part of the evaluation, we use the same model as for evaluating *filtering* queries. We begin by including *prediction* queries. Afterwards, we include *hindsight* queries and evaluate whether there is a difference between performing 10 times *prediction* or *hindsight* given a lifted solution without count-conversions. Further, we empirically investigate for how many time steps LDJT can keep FO jtrees in memory. Finally, we combine *prediction* and *hindsight* queries.

Prediction Queries Similar to *filtering* queries for one representative, we now have one *filtering* and one *prediction* query, 10 time steps into the future, for one representative of each PRV and time step. The results are shown in Figs. 5.4a to 5.4d

Figures 5.4a to 5.4c show the runtimes of the approaches, except for UUMLN, for $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$. The runtimes of LDJT alone for the three domain sizes are depicted in Fig. 5.4d. In these figures, we can see highly similar curves as we have seen for *filtering* queries for one representative. By comparing Figs. 5.4a to 5.4c to Figs. 5.1a to 5.1c and Fig. 5.4d to Fig. 5.1d, we can see that the curves are slightly elevated. The elevation can be explained as follows: For each time step, LJT needs to answer 4 additional *prediction* queries. LJT still only performs a single message pass. Again, we look at the best case for LJT, but answer slightly different queries, just the number of queries for each time step is identical. LDJT and DJT perform 10 additional message passes on an FO jtree for each time step to be able to answer the *prediction* queries. Hence, LDJT and DJT perform 10 additional message passes and 4 additional queries for each time step compared to only answering *filtering* queries. Overall, LDJT also answers the queries in linear time w.r.t. the maximum number of time steps under the presence of *prediction* queries with a constant offset. Thus, the runtimes of LDJT for each time step only increase by a constant factor.

Hindsight Queries Next, we investigate wether there is a difference between *hindsight* and *prediction* queries given a lifted solution without count-conversions. Similar to the



(a) n = 10 (b) n = 100

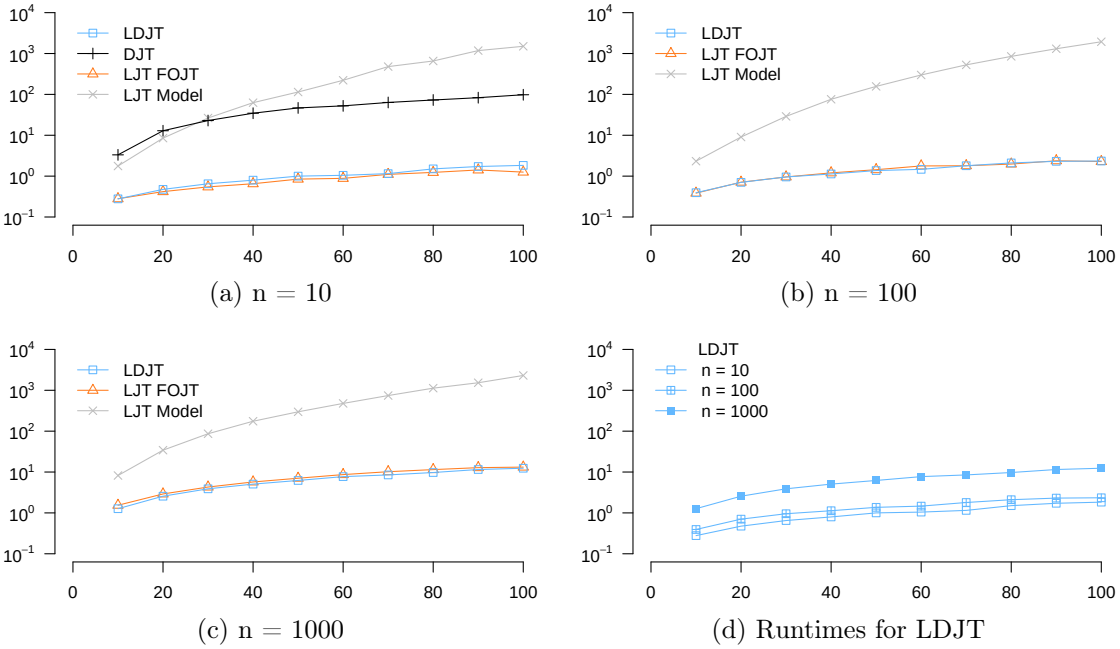
(c) n = 1000 (d) Runtimes for LDJT

Figure 5.4: Prediction queries for different domain sizes, y-axis: runtimes [seconds, log], x-axis: time steps

prediction queries, we ask one *filtering* and one *hindsight* query with 10 time steps into the past for each PRV and time step as well as on-demand instantiation for *hindsight* queries. The results are shown in Figs. 5.5a to 5.5d.

Figures 5.5a to 5.5c show the runtimes for $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$. The runtimes of LDJT alone for the three domain sizes are depicted in Fig. 5.5d. Comparing Figs. 5.5a to 5.5c to Figs. 5.4a to 5.4c and Fig. 5.5d to Fig. 5.4d, we can see that the curves are highly similar. For each domain size, answering *hindsight* or *prediction* queries with the same off set yields similar runtimes. Thus, we can say that answering *hindsight* or *prediction* queries given a lifted solution without count-conversions takes about the same time in our evaluation. In both cases, LDJT needs to perform 10 additional message passed and answer 4 additional queries for each time step. The only difference is whether an α or β message needs to be computed, i.e, only the parcluster which is used to calculate a message over the same PRVs changes.

Now, we have a look at how many FO jtrees LDJT can keep in memory as keeping FO jtrees in memory is mostly crucial for *hindsight* queries. Therefore, we increase T up to 10000. Figures 5.6a to 5.6c show the runtimes for $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$. The runtimes of LDJT alone for the three domain sizes are depicted in Fig. 5.6d. In Figs. 5.6a to 5.6c, we can see that LJT could only unroll the FO jtrees



(a) n = 10 (b) n = 100

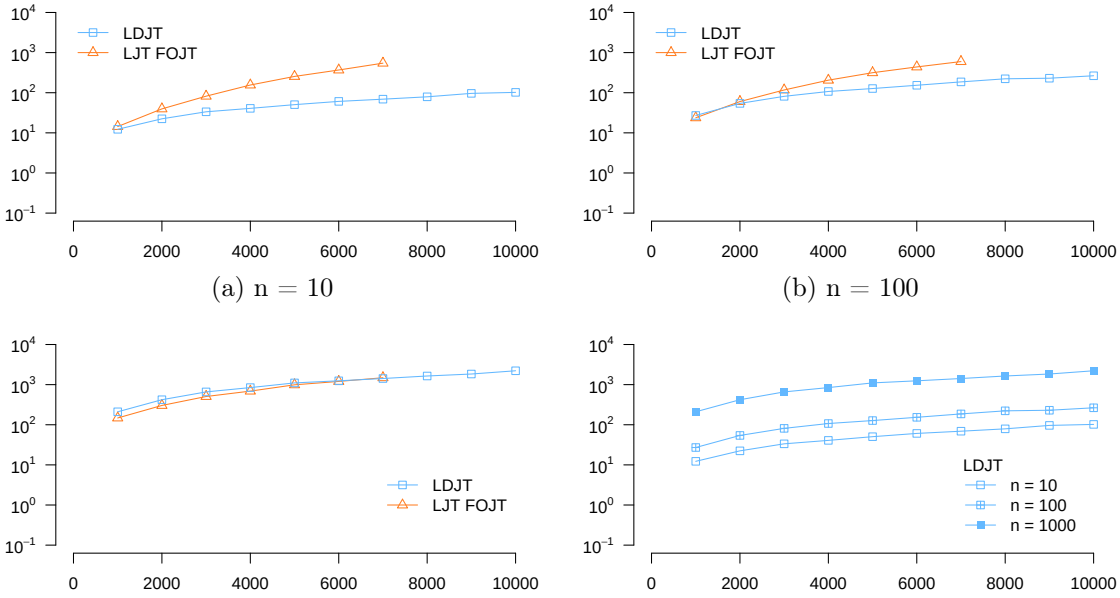
(c) n = 1000 (d) Runtimes for LDJT

Figure 5.5: Hindsight queries for different domain sizes, y-axis: runtimes [seconds, log], x-axis: time steps

for about 7000 time steps. Thus, with bounded memory, LDJT cannot always preserve all FO jtrees in memory. Hence, being able to instantiate FO jtrees from α message and evidence again is crucial, because α message and evidence require far less memory compared to an FO jtree. Therefore, LDJT can answer queries with huge lags only because LDJT is able to instantiate FO jtrees on demand while answering hindsight queries. Additionally, in Fig. 5.6d, we can see that the behaviour of LDJT remains the same also with large T's.

Combination of Prediction and Hindsight Queries Knowing how *prediction* and *hindsight* queries influence the runtime of LDJT, we now combine *hindsight*, *filtering*, *prediction* queries for each time step. Here, we empirical evaluate two aspects, namely a combination of *hindsight*, *filtering*, *prediction* queries with a fixed offset as well as querying always the first and the last time step for each time step. The results for a fixed offset, here 10 time steps into the past and 10 time steps into the future, are shown in Figs. 5.7a to 5.7d. The results for always querying the first and last time step are shown in Figs. 5.8a to 5.8f.

Figures 5.7a to 5.7c show the runtimes for $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$. The runtimes of LDJT alone for the three domain sizes are depicted in Fig. 5.7d.



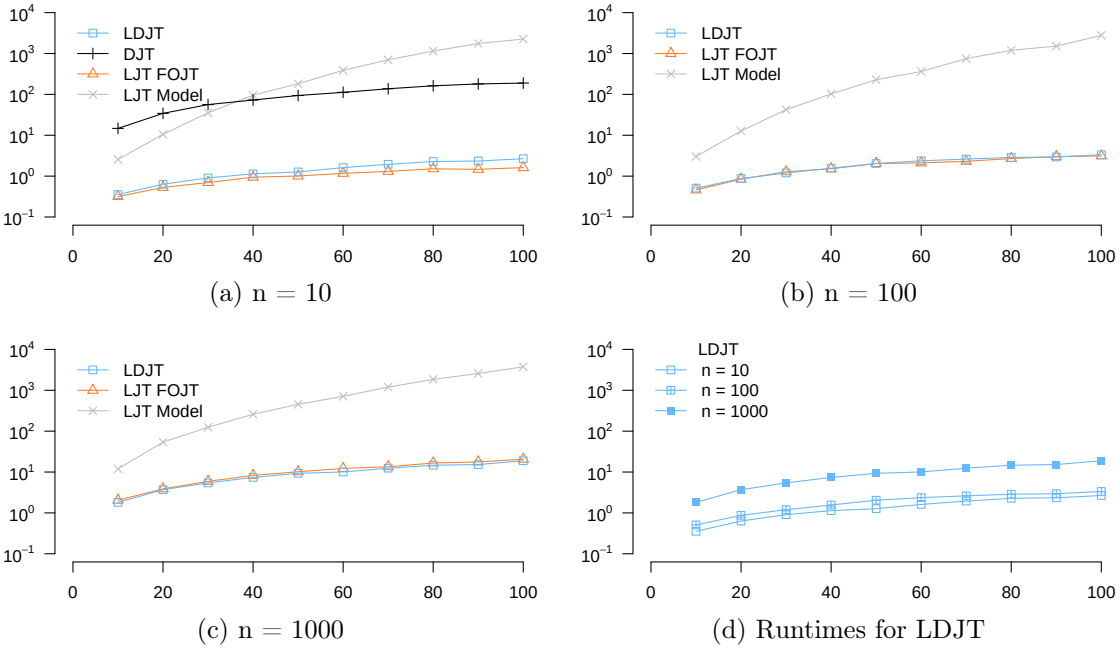
(a) n = 10 (b) n = 100

(c) n = 1000 (d) Runtimes for LDJT

Figure 5.6: Hindsight queries for different domain sizes and a large T, y-axis: runtimes [seconds, log], x-axis: time steps

In these figures, we can see highly similar curves as we have seen for *hindsight*, *filtering*, and *prediction* queries. By comparing Figs. 5.7a to 5.7c to either Figs. 5.4a to 5.4c or Figs. 5.5a to 5.5c, and comparing Fig. 5.7d to either Fig. 5.4d or Fig. 5.5d, we can see that the curves for *hindsight* and *prediction* queries are slightly elevated. Further, we can see that the elevation is about the same as we have seen before from comparing Figs. 5.4a to 5.4c or Figs. 5.5a to 5.5c to Figs. 5.1a to 5.1c. Thus, we see what is to be expected, namely that also a combination of the query types only adds the additional workload of the *prediction* and *hindsight* queries to the overall runtime.

Analogous to the elevated runtimes of *prediction* queries, we can explain the elevation in this setting. LJT needs to answer 8 additional query, the *hindsight* and *prediction* query, for each time step, but still only performs one message pass. LDJT and DJT perform 20 additional message passes on an FO jtree for each time step to be able to answer the *hindsight* and *prediction* queries. Thus, LDJT and DJT perform 20 additional message passes and 8 additional queries for each time step compared to only answering *filtering* queries. Here, we can also see that LDJT sometimes is slower compared to LJT with an unrolled model, because LDJT performs more message passes to answer the actual queries. Nonetheless, it is still impressive that LDJT requires roughly the same time, while calculating roughly 20 times as many messages. Overall, LDJT answers the queries in linear time w.r.t. the maximum number of time steps under the presence of



(a) n = 10 (b) n = 100

(c) n = 1000 (d) Runtimes for LDJT

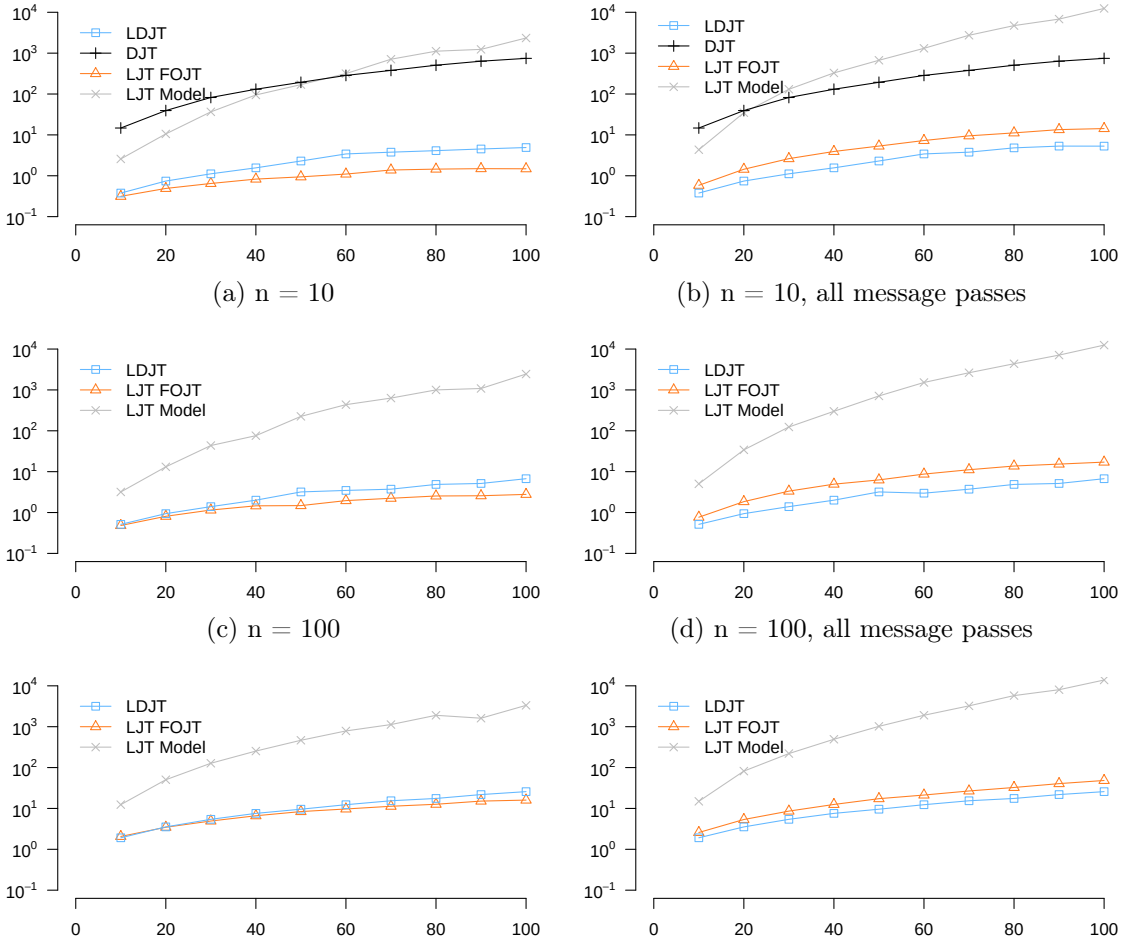
Figure 5.7: Prediction and hindsight queries for different domain sizes, y-axis: runtimes [seconds, log], x-axis: time steps

hindsight and *prediction* queries with a constant offset. Thus, the runtimes of LDJT for each time step only increase by a constant factor for each time step due to answering a combination of *hindsight* and *prediction* queries. Further, the runtime elevation is a combination of the runtime elevation for *hindsight* and *prediction* queries.

Having empirically evaluated the best and average complexity case of LDJT, we still need to evaluate the worst case for LDJT, namely always querying the first and last time step from each time step. Figures 5.8a to 5.8f show the runtimes for the approaches for $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$. In Figs. 5.8a, 5.8c and 5.8e LJT performs one message pass and inFigs. 5.8b, 5.8d and 5.8f LJT performs a message pass for each time step. As expected, Figs. 5.8a, 5.8c and 5.8e are even further elevated compared to Figs. 5.7a to 5.7c. Further, now LJT with an unrolled FO jtree is always faster compared to LDJT as LDJT computes roughly 100 times more messages. However, to answer the very same queries, i.e., the same *hindsight*, *filtering*, and *prediction* queries, LJT would have to perform a message pass for each time step. The corresponding runtimes are depicted in Figs. 5.8b, 5.8d and 5.8f and as we can see then LDJT is always the fastest approach again.

With LJT performing one message pass for each time step and therefore answering the very same queries as LDJT and DJT, we can see that now LDJT is again always

faster compared to LJT with an unrolled FO jtree. Even though from a complexity perspective LDJT and LJT with an unrolled FO jtree are the same, LDJT actually needs to compute fewer messages. For $T = 100$, LJT with an unrolled FO jtree roughly calculates 600 messages and LDJT computes only about 500 messages for each time step for our example. The difference originates from calculating only the necessary α and β messages. Further, LDJT answers queries on an FO jtree with 3 parclusters, while LJT has an FO jtree with 299 parclusters. Thus, identifying the parcluster to answer a query is easier for LDJT.



(a) n = 10

(b) n = 10, all message passes

(c) n = 100

(d) n = 100, all message passes

(e) n = 1000

(f) n = 1000, all message passes

Figure 5.8: Always querying all time steps from each time step for different domain sizes, y-axis: runtimes [seconds, log], x-axis: time steps

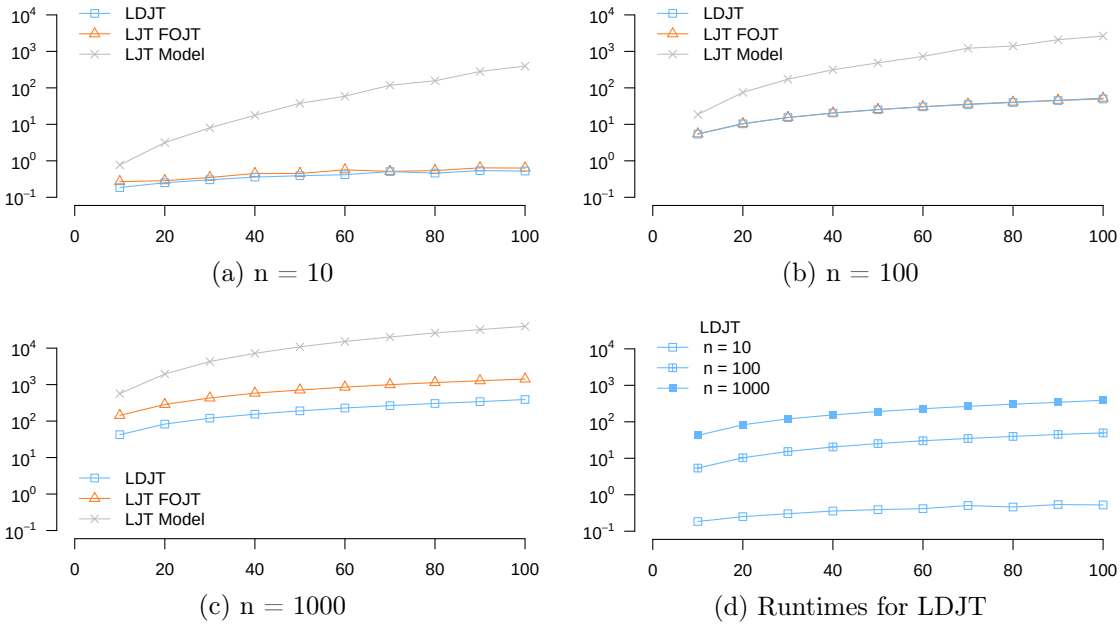
Overall, to answer the questions for *hindsight* and *prediction* queries, we can say that:

- LDJT runs in linear time w.r.t. the maximum number of time steps for *hindsight* queries, *prediction* queries, and combination of both with a fixed offset.
- With a lifted solution and without any count-conversions, *prediction* and *hindsight* queries with the same offset influence the runtimes roughly the with same overhead.
- LDJT runs in quadratic time w.r.t. the maximum number of time steps for always querying the first and the last time step.
- With always querying the first and the last time step, the runtimes of LDJT are bounded by unrolling the FO jtree and using LJT.
- LDJT cannot always keep all FO jtrees in memory.
- LDJT answer hindsight queries with huge lags, because it can instantiate FO jtrees from α messages and evidence during a backward pass.

5.3 Count Conversions while Calculating Temporal Messages

Knowing the influences of *hindsight*, *filtering*, and *prediction* queries for a completely lifted run without any count-conversions, we now investigate the influence of count-conversions. Thus, we use a variation of G^{ex} with count-conversions in α and β messages to evaluate the influences of count-conversions. Therefore, we vary the domain size for the logvar C, i.e., $|\mathcal{D}(C)| = 10$, $|\mathcal{D}(C)| = 100$, and $|\mathcal{D}(C)| = 1000$, with C being the logvar that LDJT count-converts. For the runtimes of LJT with an unrolled FO jtree and LJT with an unrolled model, LJT only performs one message pass instead of T message passes. As we have already evaluated the influence of *prediction* and *hindsight* queries, we now focus on *filtering* queries for one ground PRV. Figures 5.9a to 5.9d show the results of the evaluation.

In Figs. 5.9a to 5.9c we can see the runtimes of LDJT, LJT with an unrolled FO jtree, and LJT with an unrolled model for $|\mathcal{D}(C)| = 10$, $|\mathcal{D}(C)| = 100$, and $|\mathcal{D}(C)| = 1000$. We have not included DJT, because even for the smallest domain size DJT has not completed a single run. Further, as we have seen in Section 4.3.3, DJT anyhow clusters all ground interface PRVs together. Thus, unlike for LJT compared to a ground JT algorithm, LDJT always works on a tree where w_g of the lifted width is lower compared to the ground width of DJT. LJT only has a smaller w_g of the lifted width compared to the ground width of an JT algorithm in the presence of count-conversions. In all three figures, we can see similar trends in the curves. LJT with an unrolled model always takes the longest and LJT with an unrolled FO jtree bounds the runtimes of LDJT. Further, we can see that increasing the domain size also increases the runtimes. Generally, the results do not differ much from the runtimes for *filtering* queries, shown in Figs. 5.1a



(a) n = 10 (b) n = 100

(c) n = 1000 (d) Runtimes for LDJT

Figure 5.9: Count conversions in α messages for different domain sizes, y-axis: runtimes [seconds, log], x-axis: time steps

to 5.1c. The main difference is that now increasing the domain sizes has a bigger impact compared to runs without count-conversion.

Let us now have a look at the runtimes of LDJT, which are depicted in Fig. 5.9d. We can see that the runtimes of LDJT are linear w.r.t. the maximum number of time steps. Further, we can see that the runtimes increase by increasing the domain sizes. For example increasing the domain size from $|\mathcal{D}(C)| = 100$ to $|\mathcal{D}(C)| = 1000$, LDJT roughly takes an order of magnitude longer to answer the corresponding queries. In the runtime complexity of LDJT, the largest domain size of a count-conversions is a term that is exponentiated with the maximum number of count-conversions in a parcluster times the maximum number of range values of a CRV. Thus, the runtimes are polynomial w.r.t. to the domain size of logvars that are count-converted.

Overall, to answer the questions for count-conversions we can say that:

- LDJT runs in linear time w.r.t. the maximum number of time steps.
- The runtimes of LDJT are polynomial w.r.t. to the largest domain size of logvars that are count-converted.

5.4 Preventing Groundings while Calculating Temporal Messages

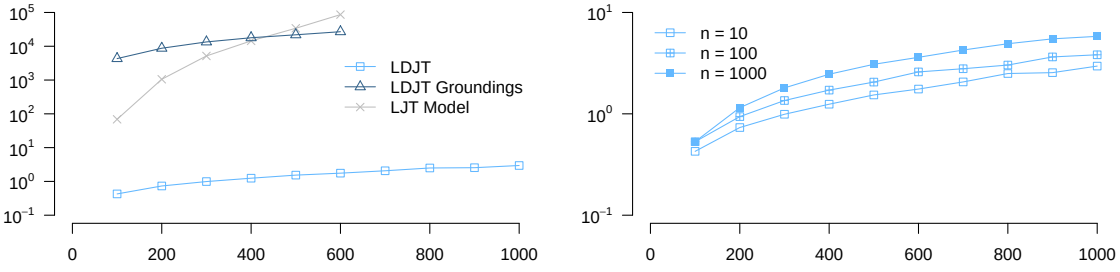
Let us now evaluate why LDJT needs the preventing grounding step as described in Section 3.4. For the evaluation, we use a variation of G^{ex} that allows to prevent groundings. Further, we compare the runtimes of LDJT with and without preventing groundings and the runtimes of unrolling the model with LJT with only one message pass.

In Fig. 5.10a, we can see the runtimes of LDJT with and without preventing groundings and the runtimes of unrolling the model with LJT for $|\mathcal{D}(X)| = 10$. Overall, the model has 110 groundings for each time step. What we can see is that at first, using LJT with an unrolled model is faster compared to using LDJT without preventing groundings. However, after about 400 time steps LDJT without preventing groundings becomes faster, due to handling temporal aspects efficiently. For this evaluation, LJT with an unrolled model constructs an FO jtree with only 2 parcluster as we would expect with PRV becoming correlated over time. Thus, the size of a parcluster grows fast with an increasing number of time steps. Additionally, we can see that LDJT with preventing groundings is several orders of magnitude faster compared to LDJT without preventing groundings.

For LDJT with preventing groundings, we also increase the domain size of X to 100 and 1000. The corresponding runtimes are depicted in Fig. 5.10b. The runtimes are as expected for a lifted run in LDJT. Hence, we can say that the small overhead of preventing grounding is always well spent compared to grounding.

Overall, to answer the questions for preventing groundings, we can say that:

- The effort to prevent groundings pays off.
- It is sometimes better to unroll the model and obtain a lifted solution compared to handling temporal aspects and suffering groundings.



(a) n = 10 (b) Runtimes for LDJT

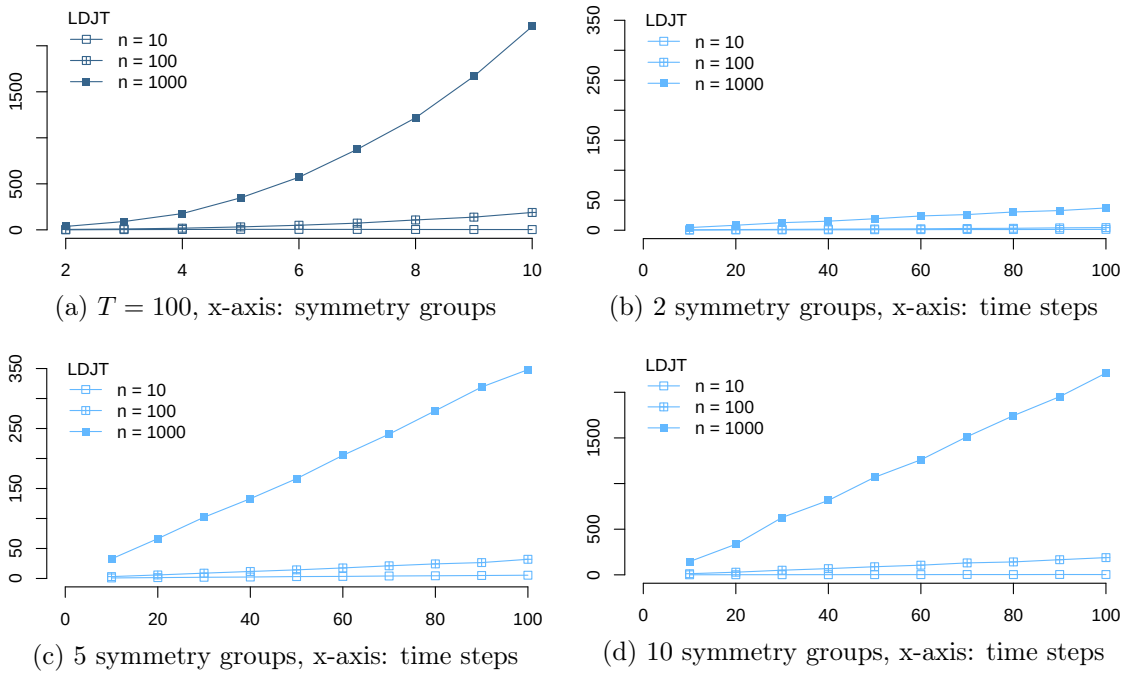
Figure 5.10: Preventing groundings, y-axis: runtimes [seconds, log], x-axis: time steps

5.5 Evidence

Finally, let us evaluate the influence of evidence on LDJT. For this part of the evaluation we use the same model we have used for the *hindsight*, *filtering*, and *prediction* queries evaluation. We do not evaluate how increasing the number of instances for which we have evidence influences runtimes as this has already been throughly evaluated for LJT. However, we evaluate how we increase the number of distinct groups for which we observe evidence influences a temporal system. Thereby, LDJT needs to always reason over a given number of groups for each time step. To evaluate the influence of evidence, we provide symmetric evidence, i.e., we have groups of instances that behave the same. By behaving the same, we mean that they receive the same evidence for each time step, but the evidence can change from one time step to the next. For example, we have a group of instances, which receives $DoR_4(X') = true$ and $DoR_6(X') = false$. Hence, we observe that these instances do research at time step 4 and do not do research at time step 6. For the evaluation, we again vary the domain size for the logvar X, i.e., $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$, and we provide evidence for X. While creating symmetry groups, we always leave 10% of the instances without evidence and distribute the rest evenly on the number of symmetry groups for doing research. For example, with $|\mathcal{D}(X)| = 100$ and 2 symmetry groups, we have a group of 10 instances without evidence and two groups of each 45 instances with symmetric evidence. Thus, overall LDJT reasons over 3 groups.

In Fig. 5.11a we can see the runtimes for LDJT with T fixed to 100 for 2 up to 10 symmetry groups with evidence. Thus, LDJT needs to reason over 3 to 11 groups as we always have one group without evidence and that for every time step. Lifted inference in general is defined to run in polynomial time w.r.t. the domain size. We can observe in Fig. 5.11a that LDJT runs in polynomial time w.r.t. number of symmetry groups. By introducing more symmetry groups, LDJT needs to perform the same eliminations more often as it needs to eliminate the same PRVs for each symmetry group. However, from a complexity perspective, we would expect a rather linear increase in runtimes. For $n = 1000$, the complexity between 2 and 10 symmetry groups is that 2 symmetry groups implies a factor of $2 \cdot \log_2(450) + \log_2(100)$ and using 10 symmetry groups implies a factor of $10 \cdot \log_2(90) + \log_2(100)$. Hence, from a complexity perspective the runtimes should not increase in a polynomial way w.r.t. the number of symmetry groups. However, by increasing the number of symmetry groups, the messages that LDJT calculates and the local models contain more parfactors. With more parfactors, a heuristic for determining the next best action also has a much larger search space. Thus, the actual runtimes behaviour differ from the to be expected runtime behaviour based on the runtime complexity.

Figures 5.11b to 5.11d depict the runtimes of LDJT for 2, 5, and 10 symmetry groups. In these figures, we can see that the runtimes for a fixed domain size again are linear w.r.t. the maximum number of time steps. Without evidence, we have observed a logarithmic



(a) $T = 100$, x-axis: symmetry groups

(b) 2 symmetry groups, x-axis: time steps

(c) 5 symmetry groups, x-axis: time steps

(d) 10 symmetry groups, x-axis: time steps

Figure 5.11: Evidence for different number of symmetry groups, y-axis: runtimes [seconds]

increase in the runtime when increasing the domain sizes. With evidence, the increase does not seem to be logarithmic anymore. From a complexity perspective, increasing domain sizes increases the runtime by a logarithmic factor that is multiplied with a linear but constant factor. Roughly, in the LVE implementation by Taghipour (`https://dtai.cs.kuleuven.be/software/lve`), the constraints contain the excluded individuals, and we use that implementation as the basis for our implementation. Thus, with more symmetry grounds and increasing domain sizes, the list of excluded individuals also grows. For the actual runtimes the complexity of constraints for each parfactor also influence the runtime, leading to the observed behaviour in Figs. 5.11b to 5.11d.

Overall, to answer the questions how evidence influences the runtimes, we can say that:

- With evidence, the runtimes are linear w.r.t. the domain size.
- LDJT runs in polynomial time w.r.t. number of symmetry groups.

Before we extend the query language of LDJT, we provide an interim conclusion for the basic for of LDJT with *hindsight*, *filtering*, and *prediction* queries.

Chapter 6

Interim Conclusion

We present LDJT to exactly and efficiently answer multiple *hindsight*, *filtering*, and *prediction* queries for temporal probabilistic relational models. To the best of our knowledge, LDJT is the first relational forward backward algorithm. LDJT answers multiple queries by reusing a compact FO jtree structure for multiple queries and time steps. 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. To obtain a relational forward backward algorithm, we explain how to construct FO jtree structures and computes temporal messages on these structures with LDJT (Contribution **1**). To efficiently handle different types of queries, *hindsight*, *filtering*, and *prediction* queries, we present a query answering plan that is efficient w.r.t. computations and memory consumption (Contribution **1**). We also show that one cannot always simply lift a propositional algorithm, but also has to ensure lifting preconditions, especially when additional constraints on an elimination exist. To ensure lifting preconditions in LDJT, we introduce a step to prevent unnecessary groundings (Contribution **2**). Further, we show that for temporal probabilistic relational models one has to trade off completeness with handling temporal aspects efficiently (Contribution **3**). Additionally, we show that query answering for temporal probabilistic relational models with the propositional interface algorithm becomes infeasible for large domain sizes, as it is exponential in the randvars in the interface (Contribution **3**). With a lifted solution, LDJT does not encounter the problem of the propositional interface algorithm, making query answering feasible even for large domain sizes (Contribution **3**). Empirical results show that the runtime of LDJT is linear in the number of time steps and significantly outperforms LJT when answering identical queries. Overall, answering *hindsight*, *filtering*, and *prediction* queries becomes manageable in combination with lifting.

Interesting future work includes a tailored automatic learning algorithm for PDMs, parallelisation of LDJT as well as using local symmetries. For learning a temporal model, one may need a forward backward algorithm, such as the Baum–Welch algorithm (Baum *et al.*, 1970). Thus, we now take the first step to be able to learn temporal probabilistic relational models. The presented backward pass could also be helpful with incrementally changing models, i.e., parfactors, PRVs, or individuals changing from one time step to

the next. Depending on the semantics, changes could also influence previous time steps and therefore, a backward pass is needed. Another interesting direction is extending LDJT to handle continuous time.

In the next part, we extend the querying language with conjunctive queries, assignment queries, and maximum expected utility (MEU) queries.

Part II

Extending the Query Language

Chapter 7

Conjunctive Queries

The first extension to the query language that we investigate is conjunction of queries. LDJT in its basic form answers queries with single ground query terms. However, one might also be interested in the marginal distribution of a conjunction of ground query terms. Conjunctive queries can be, for example, used to perform probabilistic complex event processing (Wang *et al.*, 2013). In our publishing example, this could correspond to asking how likely it is that *bob* does research in time step t, publishes in $aaai\_press$ in $t + 2$, and attends a conference in $t + 6$. In other domains, such as healthcare, one might be interested whether an Alzheimer patient put his jacket on before leaving his home. Such temporal patterns can be modelled with temporal conjunctive queries.

In this chapter, we introduce LDJT^{con} to answer multiple temporal conjunctive queries efficiently. LJT answers conjunctive queries by merging a subtree of an FO jtree, which contains all query terms (Braun and Möller, 2018a). The idea here is to reuse computations already performed during a message pass. However, the underlying assumption to reuse computations is that query terms of a conjunctive query can be found closely together in an FO jtree. If the query terms only appear in opposite leaf nodes of an FO jtree, the subtree and the FO jtree fall together in the worst case. In temporal models, we cannot assume that query terms of a conjunctive query only span a small subtree, but they will most likely span several time steps. Therefore, we show how to avoid eliminations of query terms to answer multiple conjunctive queries efficiently. Further, by avoiding eliminations, LDJT^{con} only slightly increases the size of parclusters, compared to merging subtrees. Thereby, LDJT^{con} allows for reusing more computations to answer multiple conjunctive query for different representatives.

In the following, we begin by recapitulating how LJT answers static conjunctive queries based on Braun and Möller (2018a). Afterwards, we introduce LDJT^{con}. Lastly, we analyse LDJT^{con} theoretically as well as evaluate it empirically and conclude by looking at possible extensions.

This chapter is based on the following publication:

> Marcel Gehrke, Tanya Braun, and Ralf Möller. Answering Multiple Conjunctive Queries with the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the AI 2018: Advances in Artificial Intelligence*, pages 543–555. Springer, 2018

In addition to what the paper contains, we present how to use LDJT^{con} without unrolling an FO jtree for conjunctive queries, as well as theoretical analysis and empirical results.

7.1 Conjunctive Queries in LJT

Braun and Möller (2018a) present LJT^{con} for conjunctive queries. To allow for conjunctive queries, we begin by extending Definition 2.1.4 to allow for multiple query terms in a query for a PM G.

Definition 7.1.1. Given grounded PRVs $\mathcal{Q}$ and grounded PRVs with fixed range values $\mathbf{E} = \{E^i{=}e^i\}_i$, the expression $P(\mathcal{Q}|\mathbf{E})$ denotes a conjunctive query w.r.t. $P(G)$.

Having the possibility to ask conjunctive queries, we still need means to answer them with LJT^{con}. To answer single term queries, LJT finds a parcluster that contains the query term and uses that parcluster to answer the query. In general, after message passing, a parcluster can answer queries about all of its PRVs. Even more, LJT^{con} can also directly answer conjunctive queries about ground PRVs that occur together in a parcluster. In case, the corresponding PRVs of conjunctive query terms do not occur together in a parcluster, then LJT^{con} cannot use any parcluster alone for query answering. Thus, LJT^{con} builds a parcluster containing all query terms to leverage its default query answering behaviour. To do so, LJT^{con} identifies a subtree containing all query terms. LJT^{con} merges the subtree into one parcluster to answer the query. Further, LJT^{con} can still use the messages calculated during the initial message pass, which enter the subtree from the outside. Thus, after merging the subtree, LJT^{con} can directly use LVE on the local model of the merged subtree with the messages to answer a conjunctive query.

Example 7.1.1 (Answering conjunctive queries)**.** *For our initial example G^{ex}, depicted in Fig. 2.1, one possible FO jtree is shown in Fig. 2.2, which has two parclusters. One parcluster contains the PRVs Hot, $Att(X)$, and $Pub(X, J)$ and the other parcluster contains the PRVs Hot, $Att(X)$, and $DoR(X)$. Assume that we have the queries $P(Hot, Att(bob))$ and $P(Pub(bob, springer), DoR(bob))$. For each query, LJT^{con} finds a subtree containing the query terms. The query terms of the first query, $P(Hot, Att(bob))$, are contained in both parclusters. Hence, LJT^{con} can use either of the parclusters to answer the query, without having to merge any parclusters. For the other query, no parcluster contains both query terms. One parcluster contains the PRV $Pub(X, J)$ and the other parcluster the PRV $DoR(X)$. Hence, the subtree for the query terms spans the two parclusters shown in Fig. 2.2. To answer $P(Pub(bob, springer), DoR(bob))$, LJT^{con} then merges the two parclusters, resulting in one parcluster containing all query terms. Now, LJT^{con} can answer the query $P(Pub(bob, springer), DoR(bob))$.*

LJT^{con} is highly efficient if the query terms only span a small subtree in comparison to the corresponding FO jtree. With a small subtree, LJT^{con} can still use most of the

messages calculated for the FO jtree. Additionally, the size of a parcluster only increases slightly with a small subtree. However, in temporal models, the subtree can consist of several time steps, resulting in a large subtree, which can be more or less equivalent to an unrolled FO jtree for these time steps. Therefore, hardly any messages can be reused. Further, with LDJT it can be the case that the messages for all other excerpt the current time step need to be computed for the conjunctive query. Hence, merging a subtree does not have the advantage of reusing many messages. Merging a large subtree also leads to a huge parcluster. Answering multiple queries on that huge parcluster also unnecessarily increases runtimes.

7.2 Conjunctive Queries in LDJT

Next, we introduce LDJT^{con} to answer multiple temporal conjunctive queries. In case there only are conjunctive *filtering* queries, meaning that all query terms are from the same time step, then we use LJT^{con}. However, in case the query terms of a conjunctive query are from time step t up to time step $t + \delta$, LJT^{con} would need to instantiate FO jtrees for δ time steps and identify a subtree for the combination of δ FO jtrees. The subtree contains at least $(\delta-2)\times m+2$ parclusters, where m is the number of parclusters on the path between *in-* and *out-cluster*. Thus, merging the parclusters of the subtree leads to a parcluster with many PRVs. Further, we have seen that the runtime of LJT depends on the maximum number of PRVs in a parcluster. Hence, we propose LDJT^{con}, an approach to answer multiple temporal conjunctive queries, which merges fewer PRVs in a parcluster.

Before we introduce LDJT^{con}, we extend Definition 3.1.2 to allow for multiple query terms in a temporal query.

Definition 7.2.1. Given a PDM G, grounded PRVs $\mathcal{Q}_t$ and grounded PRVs with fixed range values $\mathbf{E}_{0:t} = \{E_t^i = e_t^i\}_{i,t}$, $P(\mathcal{Q}_t|\mathbf{E}_{0:t})$ denotes a query w.r.t. $P(G)$.

Each query that LDJT^{con} answers can be a conjunctive query. To answer a conjunctive query, LDJT^{con} needs a parcluster containing all query terms. LDJT^{con} constructs this parcluster without over-approximating the number of PRVs as much as if merging a subtree. Basically, LDJT^{con} avoids eliminations of query terms to obtain one parcluster with all query terms. To send a message from parcluster $\mathbf{C}^1$ to $\mathbf{C}^2$, LDJT eliminates all PRVs from $\mathbf{C}^1$ that are not included in the separator $\mathbf{S}^{12}$. Hence, LDJT^{con} extends separators with PRVs corresponding to the query terms. To be able to answer multiple conjunctive queries that only differ in the representative, LDJT^{con} does not only add the query terms to the separator but the corresponding PRVs. A PRV is in a separator iff the PRV is contained in associated parclusters. To avoid the elimination of a PRV, LDJT^{con} adds the PRV to all parclusters on the path from the parcluster, where the PRV would be eliminated, to a designated parcluster. By extending parclusters with query

Algorithm 5 Answer Conjunctive Query for Unrolled FO Jtree $\mathcal{J}$ for Time Steps t to $t + \delta$ and Conjunctive Query $\mathcal{Q}$

```
procedure AnswerConjunctiveQuery(𝒥, 𝒬)
    root := Parcluster with the most query terms from time step t + δ
    for all Leaf parcluster p ∈ 𝒥 do
        current := p
        while current ≠ root do
            qt := 𝒬 ∩ current
            next := next parcluster on the path to root
            next := next + qt
            current := next
    𝒥 := LJT.PassMessages(𝒥)
    LVE.AnswerQuery(root, 𝒬)
```

PRVs, LDJT^{con} avoids the elimination of the query terms to answer conjunctive queries by leveraging the behaviour of LDJT for answering a query.

A naïve approach to extend parclusters is to add the query PRVs to all parclusters of the relevant time steps. Unfortunately, by over-approximating the extension of parclusters, the number of PRVs in each parcluster increases, and unfortunately the complexity of LVE depends on the PRVs in parclusters.

For ease of explanation, we start with a semi-naive approach to add the query PRVs on demand, which is outlined in Alg. 5. To answer a conjunctive query, the semi-naive approach instantiates FO jtree $\mathcal{J}$ for the time steps t to $t + \delta$ of $\mathcal{Q}$. From $\mathcal{J}$, the semi-naive approach selects a *root* parcluster, which contains most of the query terms from $\mathcal{Q}$ and is from the last time step of $\mathcal{J}$, as designated receiver of all query PRVs. Now, the semi-naive approach needs to avoid the elimination of the query terms of $\mathcal{Q}$ to the *root* parcluster. Therefore, starting from each leaf parcluster, the semi-naive approach traverses the path to the *root* parcluster. As FO jtrees are cycle-free graphs, there is exactly one path from each leaf parcluster to the *root* parcluster. While traversing the paths, the semi-naive approach checks whether a parcluster contains query PRVs and adds the query PRVs to all parclusters on the path to the *root* parcluster. Thereby, the semi-naive approach avoids the elimination of query terms to the *root* parcluster. Another way of interpreting the extension of the *root* parcluster is to add all the query terms of $\mathcal{Q}$ to the *root* parcluster and then ensure the running intersection property of an FO jtree. After *root* is extended, the semi-naive approach has to repeat a message pass, as the PRVs in parclusters changed. Lastly, the semi-naive approach can use LVE to answer the conjunctive query with the local model of the *root* parcluster.

Unfortunately, by avoiding eliminations of query terms, the semi-naive approach needs to perform an extra message pass and needs to unroll the FO jtree for δ time steps, as

outlined in Alg. 5. Nonetheless, the approach is still advantageous over identifying a subtree and merging the subtree into one parcluster for conjunctive queries over multiple time steps. Even though the work to answer one conjunctive query is the same, our approach is parallelisable and the search space for the elimination order is smaller. Further, for a second conjunctive query with the same query PRVs but different groundings, the work of the message pass can be reused.

Example 7.2.1 (Answering conjunctive queries with the semi-naive approach)**.** *Assume that we are interested in* $Pub_t(x_1, j_1)$*,* Hot_{t+2}*, and* $DoR_{t+2}(x_1)$*. Figure 3.7 shows our example model unrolled for time step* 3 *and* 4*, without parcluster* $\mathbf{C}_3^3$*. For the conjunctive query* $P(Pub_2(eve, springer), Hot_4, DoR_4(eve))$*, LDJT*^{con} *can apply the steps of Alg. 5 to answer the query. First, LDJT*^{con} *selects* $\mathbf{C}_4^3$ *as root parcluster because* $\mathbf{C}_4^3$ *is from the latest time step and is the parcluster containing most of the query terms in* $t = 4$*. Second, LDJT*^{con} *extends the parclusters on the path from the leaf parclusters* $\mathbf{C}_3^1$ *and* $\mathbf{C}_3^3$ *to the root.* $\mathbf{C}_3^1$ *includes the query term* $Pub_2(eve, springer)$*. Hence, LDJT adds* $Pub_2(X, J)$ *to all parclusters on the path to the root parcluster, namely* $\mathbf{C}_4^1$*,* $\mathbf{C}_4^2$*, and root* $\mathbf{C}_4^3$*. No additional parcluster on the path from* $\mathbf{C}_3^1$ *to root contains any query term that is not contained in* $\mathbf{C}_4^3$*. The same holds for the path from* $\mathbf{C}_3^3$ *to root. Third, LDJT*^{con} *performs a message pass on the extended FO jtree. Last, LDJT*^{con} *uses root to answer the conjunctive query. LDJT*^{con} *increases the maximum number of PRVs in a parcluster from* 3 *to* 4*, allowing it to efficiently answer multiple conjunctive query, e.g., also for alice and bob. By performing merging, all parclusters would be merged in a parcluster with* 9 *PRVs. Further, instead of increasing the size of a parcluster with merging by* 6 *PRVs, LDJT*^{con} *increases the size of a parcluster only by* 2 *PRVs. Thus, by merging, fewer computations could be reused to answer conjunctive queries for alice and bob.*

Even though Alg. 5 is well-suited to illustrate the idea of LDJT^{con}, Alg. 5 still has room for improvement, e.g., currently, in case there are paths to the *root* parclusters join, they are traversed multiple times. Further, LDJT^{con} could directly perform message passing, while extending the parclusters and in case one only wants to use the unrolled FO jtree to answer conjunctive query with different grounding of the query PRVs, an *inbound* pass to the *root* parcluster would suffice to answer the conjunctive query. Furthermore, instead of unrolling FO jtrees, LDJT^{con} can use temporal messages to avoid unrolling. Therefore, let us now introduce LDJT^{con}.

Algorithm 6 outlines LDJT^{con}. The basic idea still is to avoid elimination of query terms. Unlike Alg. 5, Alg. 6 works on one time step at a time, traverses each parcluster only once, and requires only a single *inbound* message pass over all time steps. First, LDJT^{con} determines the earliest i and latest j time step that is queried in a conjunctive query $\mathcal{Q}$. To only require one *inbound* message pass, LDJT^{con} performs a forward message pass from time step i to j. Therefore, LDJT^{con} instantiates an FO jtree for time step i and performs a message pass with the *out-cluster* as root. During each message pass, LDJT^{con} adds the corresponding PRVs of $\mathcal{Q}$ to each separator. Hence, query terms are

Algorithm 6 LDJT^{con} for Conjunctive Query $\mathcal{Q}$

procedure LDJT^{con}($J_0, J_t, \mathcal{Q}$)
  $i :=$ earliest in $\mathcal{Q}$
  $j :=$ latest in $\mathcal{Q}$
  while $i \neq j$ **do**
    $J_i :=$ FO jtree instantiated for time step i
    $root := J_i(out - cluster)$
    Inbound message pass with *root* as root and $\mathcal{Q}$ in each separator
    Calculate new α_i with $\mathcal{Q}$ in the separator
    $i := i + 1$
  $J_j :=$ FO jtree instantiated for time step j
  root := Parcluster with the most query terms from J_j
  Inbound message pass with *root* as root and $\mathcal{Q}$ in each separator
  LVE.AnswerQuery($root, \mathcal{Q}$)

not eliminated during a message pass, but passed through to time step j. An *inbound* message pass to the *out-cluster* as root suffices because LDJT^{con} only needs to be able to calculate α_i. For α_i, LDJT^{con} also adds the corresponding PRVs of $\mathcal{Q}$ to the separator. By going through each time step from i to j in this fashion, LDJT^{con} propagates all query terms to time step j. LDJT^{con} performs an *inbound* message pass with the parcluster having the most query terms in J_j as root. Then, LDJT^{con} uses this parcluster to answer $\mathcal{Q}$. Therefore, with one inbound forward message pass, LDJT^{con} can answer $\mathcal{Q}$ by avoiding to eliminate PRVs of $\mathcal{Q}$ until $\mathcal{Q}$ is collected in one parcluster.

Example 7.2.2 (Answering conjunctive queries with LDJT^{con})**.** *We again assume the conjunctive query* $P(Pub_2(eve, springer), Hot_4, DoR_4(eve))$. *As described in Alg. 6, LDJT^{con} first determines the earliest i and latest j time step of the conjunctive query, i.e., $i = 2$ and $j = 4$. In the while loop, LDJT^{con} then instantiates J_2 and performs an* inbound *message pass with the* out-cluster*, $\mathbf{C}_2^2$, as root. During the message pass, LDJT^{con} extends the separators with $Pub_2(X, J), Hot_4, DoR_4(X)$. Now, LDJT^{con} can use $\mathbf{C}_2^2$ to calculate α_2, for which LDJT^{con} also extends the separator with $Pub_2(X, J)$, Hot_4, and $DoR_4(X)$. Then, LDJT^{con} does the same for time step* 3.

For time step 4*, LDJT^{con} leaves the while loop, instantiates J_4, and selects $\mathbf{C}_4^3$ as root, because it holds the most query terms in J_4. After an* inbound *message pass, LDJT^{con} can answer the conjunctive query. Using $\mathbf{C}_4^3$, LDJT^{con} can also answer conjunctive queries for other representatives while reusing the computations from the message pass. Further, LDJT^{con} only traverses each parcluster once with only one* inbound *message pass.*

Knowing how LDJT^{con} works, we have a look at a theoretical analysis of LDJT^{con}.

7.3 Theoretical Analysis

This section investigates soundness, completeness, and complexity of LDJT^{con}.

7.3.1 Soundness

To show that LDJT^{con} is sound, we show the message pass of LDJT^{con} to be sound. Thereby, LDJT^{con} can use the selected parcluster to answer a corresponding conjunctive query. For LDJT^{con} and LJT^{con}, a parcluster needs to contain all query terms for the algorithms to be able to answer the corresponding conjunctive query. LDJT^{con} avoids the elimination of the corresponding query term PRVs. Therefore, we show that the message pass with avoiding eliminations is sound.

Theorem 7.3.1. *LDJT^{con} is sound regarding a conjunctive query $\mathcal{Q}$, i.e., LDJT^{con} produces the same results as LJT^{con} does for an unrolled FO jtree of a PDM.*

Proof. During a message pass of LDJT^{con}, it extends all separators with the corresponding PRVs of $\mathcal{Q}$. By extending separators, LDJT^{con} ensures that the PRVs of $\mathcal{Q}$ are not eliminated during a message pass. Hence, $\mathcal{Q}$ is present at the last parcluster of the message pass. Further, the FO jtrees instantiated during the message pass of LDJT^{con} are still valid FO jtrees. For them to be valid FO jtrees three properties have to hold. The first property is that all PRVs in a parcluster must originate from the underlying model, which they do as all PRVs come from the corresponding unrolled PDMs. The second property ensures that all parfactors are assigned to parclusters. The message pass does not change parfactors assignments. Thus, also the second property still holds during the message pass of LDJT^{con}. The last property is the running intersection property. By extending separators, LDJT^{con} does not eliminate PRVs of $\mathcal{Q}$ during a message pass anymore. Thus, LDJT^{con} basically adds the corresponding PRVs from their first occurrence to one parcluster of the latest time step. Hence, the PRVs are added to all parclusters from the initial parcluster to the last root parcluster. Therefore, LDJT^{con} also ensures the running intersection property.

Further, LDJT^{con} and LJT^{con} perform the same calculations only possibly in a different order. LJT^{con} finds a subtree containing all PRVs of $\mathcal{Q}$ and then eliminates everything but $\mathcal{Q}$ from that parcluster. LDJT^{con} still eliminates all PRVs that are not in a separator to calculate a message. Hence, LDJT^{con} has in the end a parcluster containing the PRVs of the root parcluster from the last time step as well as the PRVs of $\mathcal{Q}$. Further, the *inbound* message pass suffices to propagate all state descriptions to that parcluster so that LDJT^{con} can use exactly that one parcluster to answer the conjunctive query. All other parclusters, do not necessarily receive all messages and therefore, cannot be used for query answering. However, after the inbound message pass and answering $\mathcal{Q}$, LDJT^{con} performed the very same calculations as LJT^{con}. The only difference is that LJT^{con} first collects all parfactors and then starts to eliminate, while LDJT^{con} already starts to eliminate while collecting parfactors. □

7.3.2 Completeness

In general, we cannot make any completeness statements for LDJT^{con}. Even if LDJT^{con} works on a model, for which LDJT is complete (c.f. Thm. 4.2.3), the conjunctive query can result in a case where LDJT^{con} cannot eliminate a PRV with two logvars. For such a case, we have already shown in Thm. 4.2.2 that LDJT is not complete. However, in case we do not only restrict the model to be of a certain class, but also the conjunctive query, then we can make statements about the completeness of LDJT^{con}.

For LJT^{con}, the query class $\mathcal{CQ}^{lift}$ is restricted to query terms with at most one constant of each logvar (Braun, 2020). Otherwise, there could be a conjunctive query in which each instance of each logvar occurs in a different ground PRV, leading to grounding logvars. Unfortunately, for LDJT^{con}, $\mathcal{CQ}^{lift}$ is not restrictive enough.

Theorem 7.3.2. *LDJT^{con} is not complete for $\mathcal{CQ}^{lift}$.*

Proof. $\mathcal{CQ}^{lift}$ allows for query terms with two constants. Then, LDJT^{con} would extend separators with a PRV with two logvars. Thus, there can be the case that LDJT^{con} would not eliminate a PRV with two logvars for several time steps, which corresponds to Thm. 4.2.2. Therefore, LDJT^{con} is not complete for $\mathcal{CQ}^{lift}$. □

In case LDJT^{con} would only extend separators with the query terms of a conjunctive query instead of the corresponding PRVs, LDJT^{con} would be complete for $\mathcal{CQ}^{lift}$. Extending separators with query terms adds only PRVs with zero logvars, which is a subset of 1-logvar models. The problem with $\mathcal{CQ}^{lift}$ and LDJT^{con} is that $\mathcal{CQ}^{lift}$ allows PRVs with two logvars and LDJT^{con} may need to eliminate a PRV with zero logvars, which cannot be guaranteed to be performed without groundings.

Definition 7.3.1 (Liftable temporal conjunctive queries)**.** Let $\mathcal{TCQ}^{lift}$ be the class of temporal conjunctive queries with query terms having at most one constant of each logvar and at most one logvar for each PRV.

Theorem 7.3.3. *LDJT^{con} is complete for $\mathcal{TCQ}^{lift}$ given a liftable model and evidence.*

Proof. Adding PRVs with at most one logvar to separators does not result in groundings due to generalised counting. Further, the model stays in the same class for which LDJT and thereby LDJT^{con} is complete. By bounding the number of constants for each logvar, there is only one split for each logvar. The number of splits is dependent on the number of logvars and not the domain size. Therefore, LDJT^{con} is complete for $\mathcal{TCQ}^{lift}$ □

7.3.3 Complexity

The complexity of LDJT depends on the lifted width. In LDJT, the number of PRVs in a separator is always a subset of the parclusters that the separator connects. Thus, in LDJT, the number of PRVs in a message is always smaller than the number of PRVs

in the largest parcluster. Unfortunately, for LDJT^{con}, the number of PRVs in a message can be higher than the number of PRVs in the largest parcluster.

If extending separators of LDJT^{con} does not result in a message with parfactors having more PRVs as argument than PRVs in the largest parcluster, then the complexity of message passing of LDJT^{con} would be the same as for LDJT. Further, as LDJT^{con} only performs an *inbound* message pass, LDJT^{con} only calculates $n_J - 1$ instead of $2 \cdot (n_J - 1)$ messages, without α and β message, for each time step. Then, the only difference would be in answering queries, where the largest parcluster would be extended with the number of corresponding PRVs from the conjunctive query. Unfortunately, we cannot guarantee that the number of PRVs in a message will never extend the lifted width of LDJT. Hence, we first define a lifted width for LDJT^{con} and then investigate the combined complexity of LDJT^{con} with a conjunctive query.

Definition 7.3.2. Let $(w_g^{0\cup\mathcal{Q}}, w_\#^{0\cup\mathcal{Q}})$ be the *lifted width* of J_0 with the PRVs of $\mathcal{Q}$ added to each parcluster of J_0 and let $(w_g^{t\cup\mathcal{Q}}, w_\#^{t\cup\mathcal{Q}})$ be the *lifted width* of J_t with the PRVs of $\mathcal{Q}$ added to each parcluster of J_t. The *lifted width* w_J^{cq} of a pair (J_0, J_t) and conjunctive query $\mathcal{Q}$ is a pair $(w_g^{cq}, w_\#^{cq})$, where $w_g^{cq} = \max(w_g^{0\cup\mathcal{Q}}, w_g^{t\cup\mathcal{Q}})$ and $w_\#^{cq} = \max(w_\#^{0\cup\mathcal{Q}}, w_\#^{t\cup\mathcal{Q}})$.

The *lifted width* w_J of LDJT only depends on the FO jtrees. For conjunctive queries, the *lifted width* w_J^{cq} also depends on a conjunctive query $\mathcal{Q}$. Let us now investigate the complexity of LDJT^{con}. LDJT^{con} uses the same evidence as LDJT does and therefore, does not need to enter evidence itself. In addition to the message passes of LDJT, LDJT^{con} now also needs to perform message passes for conjunctive queries. Let us again have a look at the best, average, and worst case complexity for o distinct conjunctive queries for each time step.

Lemma 7.3.1. *The worst case complexity of conjunctive query message passes is*

$$O(o \cdot T^2 \cdot n_J \cdot \log_2 n \cdot r^{w_g^{cq}} \cdot n_\#^{w_\#^{cq} \cdot r_\#}). \tag{7.1}$$

The average case complexity of conjunctive query message passes is

$$O(o \cdot T \cdot n_J \cdot \log_2 n \cdot r^{w_g^{cq}} \cdot n_\#^{w_\#^{cq} \cdot r_\#}). \tag{7.2}$$

The best case complexity of conjunctive query message passes is

$$O(o \cdot T \cdot n_J \cdot \log_2 n \cdot r^{w_g^{cq}} \cdot n_\#^{w_\#^{cq} \cdot r_\#}). \tag{7.3}$$

In the worst case, the earliest time step in a conjunctive query is the first time step and the latest is the last time step and there is such a query for each time step. Hence, LDJT^{con} performs a message pass over all time steps for each time step. Further, Eq. (7.1) now uses w_J^{cq}, because the message pass is dependent on a conjunctive query. Additionally, LDJT^{con} performs a message pass for each distinct conjunctive query for each time

step. In the best case, we only have conjunctive queries for the current time step and in the average case, the number of time steps in conjunctive queries is constant.

For each distinct query, LDJT^{con} can answer the conjunctive query for each time step for different representatives.

Lemma 7.3.2. *The complexity of answering a set of conjunctive queries* $\{\mathcal{Q}_k\}_{k=1}^{p}$ *for different representatives is*

$$O(p \cdot \log_2 n \cdot r^{w_g^{cq}} \cdot n_{\#}^{w_{\#}^{cq} \cdot r_{\#}}). \tag{7.4}$$

LDJT^{con} answers the p conjunctive queries with different representatives on the same parcluster, which is bounded by w_J^{cq}. To answer the same conjunctive query for different representatives, LDJT^{con} only needs to eliminate non-query terms from one parcluster. Thus, the difference to LJT^{con} is that the parcluster is extended only by the PRVs of the query terms instead of all PRVs of the subtree. Therefore, the number of PRVs that LDJT^{con} needs to eliminate for the second query is normally much lower.

We now combine the stepwise complexities to arrive at the complexity of LDJT^{con} by adding up the complexities in Eqs. (4.5) to (4.9) and (7.1) to (7.4).

Theorem 7.3.4. *The worst case complexity of LDJT^{con} is*

$$O((((T^2 + T) \cdot n_J + m) + (o \cdot T^2 \cdot n_J + o \cdot p)) \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{7.5}$$

The average case complexity of LDJT^{con} is

$$O(((T \cdot n_J + m) + (o \cdot T \cdot n_J + o \cdot p)) \cdot \log_2 n \cdot r^{w_g^{cq}} \cdot n_{\#}^{w_{\#}^{cq} \cdot r_{\#}}). \tag{7.6}$$

The best case complexity of LDJT^{con} is

$$O(((T \cdot n_J + m) + (o \cdot T \cdot n_J + o \cdot p)) \cdot \log_2 n \cdot r^{w_g^{cq}} \cdot n_{\#}^{w_{\#}^{cq} \cdot r_{\#}}). \tag{7.7}$$

The complexities of Eqs. (7.5) to (7.7) are the combination of Eqs. (4.10) to (4.12) and the additional workload to answer conjunctive queries. Thus, the overall complexity of LDJT^{con} consists of the complexity of LDJT and the complexity of conjunctive queries.

7.4 Evaluation

For the evaluation, we use G^{ex} as described in Section 5.1. Further, we vary the domain size for the logvar X, i.e., $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$, while setting $|\mathcal{D}(P)| = 3$. We compare the runtimes of LJT^{con} with an unrolled model, LJT^{con} with an unrolled FO jtree, and LDJT^{con}. For each run, we ask the following conjunctive queries $\{Hot_0(x_i), Hot_{T-1}(x_i), Hot_T(x_i)\}_{i=1}^{3}$ and $\{Hot_0(x_i), Hot_T(x_i)\}_{i=1}^{3}$, resulting in

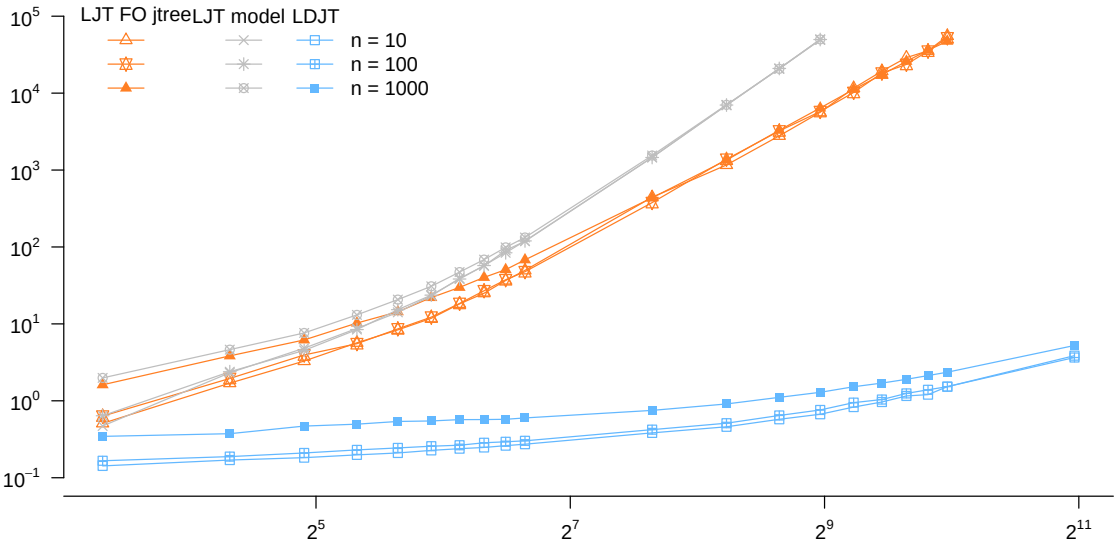


Figure 7.1: Conjunctive query runtimes [seconds, log], x-axis: time steps, log

6 conjunctive queries for each run. We also vary the maximum number of time steps T. The query terms of the second query are a subset of the query terms of the first query. Further, each conjunctive query always queries the first time step ($Hot_0(x_i)$) and the last time step ($Hot_T(x_i)$). We ask such queries, to evaluate our claim that LDJT^{con} will be faster starting with the second query.

Figure 7.1 shows the runtimes. We can see that LDJT^{con} is always significantly faster compared to LJT^{con}. After query-induced message passing, LDJT^{con} can answer the queries efficiently on a small parcluster. The work to answer the first query is roughly the same for LDJT^{con} and LJT^{con} with an unrolled FO jtree. However, for the second query, LJT^{con} with an unrolled FO jtree again has to many PRVs from huge subtree, basically spanning the complete FO jtree. Therefore, with more than one of such conjunctive queries, where LDJT^{con} can reuse the computations from its message pass, LDJT^{con} is significantly faster compared to LJT^{con}. One interesting aspect is that even though LJT^{con} with an unrolled model is always slower compared to LJT^{con} with an unrolled FO jtree, LJT^{con} with an unrolled model answers the queries faster. The message pass on the unrolled model just takes significantly longer, but LJT^{con} can reuse some of the computations, i.e., the subtree does not span nearly the complete FO jtree. Thus, LJT^{con} with an unrolled model is faster w.r.t. answering queries but takes significantly longer on message passing. Increasing the domain sizes, produces the to be expected behaviour for LJT^{con} and LDJT^{con}.

In summary, we can say that LDJT^{con} is significantly faster in case it can reuse the computations of the message pass. Further, LDJT^{con} again has the advantage of requiring less memory as it can work on one time step at a time and does not need to merge a

possibly huge subtree, making it efficient for temporal models.

7.5 Interim Conclusion

We present LDJT^{con} to answer conjunctive queries by avoiding eliminations of query terms (Contribution **4a**). To avoid eliminations, LDJT^{con} increases parclusters with query PRVs until all query PRVs are in one parcluster. LDJT^{con} efficiently answers multiple conjunctive queries for different representatives or if a query terms of a conjunctive query are a subset of query terms from another conjunctive query. Theoretical (Contribution **4b**) and empirical results show that extending can significantly save computations for multiple conjunctive queries compared to using LJT^{con} on an unrolled FO jtree.

Chapter 8

Assignment Queries

The second extension to the query language that we investigate is assignment queries. LDJT in its basic form answers marginal queries. However, one might also be interested in the most probable assignment of PRVs given evidence. In our publishing example, such a query could correspond to asking for the the most probable assignment for all PRVs, given that *bob* publishes in $aaai\_press$ in $t+2$ and attends a conference in $t+6$. In other domains, such as healthcare, one might be interested in the most probable assignment w.r.t conditions of patients given some test results.

The general idea of assignment queries compared to marginal queries is that one is interested in the most likely state, i.e., the most likely range value of a PRV, given some evidence. Thus, the general problem of assignment queries is to calculate the most probable values of some PRVs given evidence for other PRVs.

In this chapter, we introduce LDJT^{mpe} and LDJT^{map} to answer multiple temporal assignment queries efficiently. LJT^{mpe} and LJT^{map} already answer static assignment queries (Braun and Möller, 2018b; Braun, 2020). We propose to combine LDJT and LJT^{mpe} to answer temporal assignment queries efficiently. There are two types of assignment queries, namely most probable explanation (MPE), which asks for the assignment of all PRVs for which we do not have evidence, and maximum a posteriori (MAP), which asks for the assignment of some PRVs for which we do not have evidence.

In the following, we begin by presenting how LJT^{mpe} and LJT^{map} answer static assignment queries based on Braun and Möller (2018b). Afterwards, we introduce LDJT^{mpe} and LDJT^{map}. Lastly, we analyse LDJT^{mpe} and LDJT^{map} theoretically as well as empirically evaluate LDJT^{mpe} and LDJT^{map}.

This chapter is based on the following publication:

> Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Temporal Most Probable Explanation. In *Proceedings of the 24th International Conference on Conceptual Structures*, pages 72–85. Springer, 2019

In addition to the content of the publication, we also present a theoretical analysis and empirical results of LDJT^{mpe} and LDJT^{map}.

8.1 Most Probable Assignments in LJT

Based on Braun and Möller (2018b), we now define MPE queries and provide an intuition of how LJT^{mpe} solves the MPE problem.

Definition 8.1.1 (MPE problem)**.** Given a PM G and evidence $\mathbf{E}$, find the most probable assignment for all PRVs $\mathbf{V}$ in G for which there is no evidence given. Thus, the MPE problem is to solve the following expression: $\arg\max_{\mathbf{v}} P((\mathbf{V}=\mathbf{v})_{|C}|\mathbf{E}=\mathbf{e})$, $\mathbf{V}\cap\mathbf{E}=\emptyset$.

The basic idea of calculating an MPE, compared to answering marginal distribution queries, is to use a maximisation instead of a summation to eliminate PRVs. To efficiently calculate maximisations for relational probabilistic static models, Braun and Möller (2018b) propose a lifted maximisation for the current version of LVE (Taghipour *et al.*, 2013c) and also apply it to LJT (Braun and Möller, 2016), resulting in LJT^{mpe}.

LJT^{mpe} calculates a lifted solution to the MPE problem. Hence, LJT^{mpe} uses the fact that instances behave the same for assignments. Thus, for a PRV, LJT^{mpe} needs to store the number of instances for the range values of that PRV, which maximise the potential. Hence, LJT^{mpe} stores a histogram encoding the assignment.

Example 8.1.1 (Histograms in LJT^{mpe})**.** *Assume an MPE is that two people are doing research and one does not do research. Then, it is the same if either alice and bob, alice and eve, or bob and eve do research. Thus, LJT^{mpe} only needs to store a histogram with* $[2,1]$ *for the PRV DoR*(X)*.*

To compute an MPE, LJT^{mpe} does not only need to store assignments for PRVs, but also potentials, as we want the assignment that maximises the potential. Thus, to calculate an MPE, the function ϕ in a parfactor $\phi(\mathcal{A})_{|C}$ maps arguments to a pair of a potential and a set of histograms for already maxed out PRVs. By storing the potential and a set of histograms, LJT^{mpe} can read out the MPE after the last maxing out.

Example 8.1.2 (Parfactors in LJT^{mpe})**.** *LJT^{mpe} can answer an MPE query for G^{ex} shown in Fig. 2.1. Assume that we have no evidence. That means, LJT^{mpe} calculates the most probable assignment for all PRVs. After the last maxing out, a parfactor could look like this:* $\phi() \rightarrow (p, \{[6,0]_{Pub}, [3,0]_{DoR}, [3,0]_{Att}, [1,0]_{Hot}\})$*. LJT^{mpe} can use that parfactor to directly read out the most probable assignment, namely* $\{[6,0]_{Pub}, [3,0]_{DoR}, [3,0]_{Att}, [1,0]_{Hot}\}$*, as well as a corresponding potential, p.*

Knowing how histograms help us while computing an MPE, we now take a look at how LJT^{mpe} answers MPE queries efficiently. Algorithm 7 outlines the steps LJT^{mpe} performs. The first two steps are the same steps as for marginal queries. LJT^{mpe} first builds a corresponding FO jtree J from a PM G and enters evidence afterwards. The main difference compared to marginal queries is that LJT^{mpe} only performs an *inbound* message pass. LJT performs a complete message pass for marginal queries, to efficiently

Algorithm 7 LJT^{mpe} for PM G and Evidence $\mathbf{E}$

procedure LJT$^{mpe}(G, \mathbf{E})$
  Build FO jtree J using G
  Enter $\mathbf{E}$ in J
  Perform an *inbound* message pass on J
  Answer MPE query

answer multiple queries. However, only one MPE query exists for a given set of evidence, for which LJT^{mpe} needs to max out all PRVs without evidence. Thus, as long as one parcluster has all information, which a root has, after an *inbound* message pass, LJT^{mpe} can answer the MPE query using that parcluster. The other important difference is that LJT performs a maxing out compared to a summing out to eliminate a PRV. After the *inbound* message pass, LJT^{mpe} maxes out the remaining PRVs from the root cluster of the *inbound* message passing. Lastly, LJT^{mpe} can directly read out the MPE which lead to the highest potential.

Example 8.1.3 (Answering an MPE query with LJT^{mpe})**.** *Figure 2.2 depicts an FO jtree J for our example PM G^{ex}. After the evidence entering, LJT^{mpe} performs an* inbound *message pass with, e.g., $\mathbf{C}^1$ as root. Thus, LJT^{mpe} calculates the message m^{21}: LJT^{mpe} eliminates $DoR(X)$ from $\mathbf{C}^2$ by applying lifted maxing out. For each case where the range values of Hot and $Att(X)$ are the same and only the range value of $DoR(X)$ differs, LJT^{mpe} keeps the higher potential of the range value and saves the range value of $DoR(X)$ to max out $DoR(X)$. Hence, m^{21} contains a parfactor with 4 rows and each row contains the maximum potential and whether a true or false assignment of $DoR(X)$ leads to the potential. After LJT^{mpe} sends m^{21} to $\mathbf{C}^1$, $\mathbf{C}^1$ holds all state descriptions necessary to answer an MPE query. In $\mathbf{C}^1$, LJT^{mpe} still has to eliminate Hot, $Att(C)$, and $Pub(X, J)$ by applying lifted maxing out. Having eliminated all PRVs, LJT^{mpe} can return the most probable assignment given the evidence.*

Another assignment query is an MAP query, where we are interested in the most probable assignment only for a subset of PRVs. Again, we first define the MAP problem and then provide an intuition how LJT^{map} as a combination of LJT and LJT^{mpe} solves the MAP problem efficiently.

Definition 8.1.2 (MAP problem)**.** Given a PM G, evidence $\mathbf{E}$, and a set $\mathbf{U}_{|C'} \subseteq \mathbf{V}_{|C}$, an *MAP problem* refers to the problem of finding an assignment for $\mathbf{U}_{|C'}$ with the highest probability w.r.t. P_G, i.e., given $\mathbf{S}_{|C''} = \mathbf{V}_{|C} \setminus \mathbf{U}_{|C'}$: The MAP problem is to solve $\arg\max_{\mathbf{u}} P((\mathbf{U} = \mathbf{u})_{|C'} | \mathbf{E} = \mathbf{e}) = \arg\max_{\mathbf{u}} \sum_{\mathbf{s} \in \mathbf{S}_{|C''}} P((\mathbf{U} = \mathbf{u})_{|C'}, (\mathbf{S} = \mathbf{s})_{|C''} | \mathbf{E} = \mathbf{e})$.

Unfortunately, summing out and maxing out PRVs are not commutative. This non-commutativity leads to a restriction of the elimination order. For an MAP query, an

algorithm needs to sum out PRVs before maxing out query PRVs (Braun and Möller, 2018b). Hence, the problem of solving an MAP is in general harder compared to solving the MPE problem (Sharma *et al.*, 2018). To compute a *lifted* solution to the MAP problem, lifting imposes additional restrictions on the elimination order, making the problem even harder than in the propositional case.

LJT^{map} is a combination of LJT and LJT^{mpe}. Even though the MAP problem is in general harder, for LJT^{map} we can identify two types of harmless MAP queries. Harmless, in this case, means that LJT^{map} can answer an MAP query without grounding. If an MAP queries is over all PRVs from a parcluster or connected parclusters, LJT can use the message pass from marginal queries, in which LJT sums out PRVs. These MAP queries are harmless in case the message passing of LJT does not induce groundings. Based on the elimination order induced by separators, all other PRVs are summed out and LJT^{mpe} can calculate an MPE for the PRVs from the (connected) parcluster(s).

Example 8.1.4 (Harmless MAP query)**.** *Assume that we are interested in the assignment of Hot, $Att(X)$, and $Pub(X, J)$. Hence, LJT needs to sum out $DoR(X)$ and LJT^{mpe} needs to max out Hot, $Att(X)$, and $Pub(X, J)$. To answer the query, LJT^{mpe} can use $\mathbf{C}^1$. During the message pass for marginal queries, LJT calculates the message m^{21} for which it eliminates $DoR(X)$. With m^{21}, $\mathbf{C}^1$ holds all state descriptions necessary to answer the assignment query. Thus, LJT^{mpe} calculates an MPE on $\mathbf{C}^1$ to answer the MAP query.*

Unfortunately, LJT^{mpe} and LJT^{map} do not efficiently handle temporal aspects of PDMs. Thus, we now introduce LDJT^{mpe} and LDJT^{map} to efficiently answer assignment queries for relational temporal models.

8.2 Most Probable Assignments in LDJT

In this section, we investigate how MPE and MAP queries can be solved efficiently for temporal relational models and discuss the need of a temporal approach.

8.2.1 MPE Queries

We now define temporal assignment queries, introduce LDJT^{mpe}, and investigate how LDJT^{mpe} efficiently answers temporal MPE queries.

Definition 8.2.1 (Temporal MPE problem)**.** Given a PDM G and evidence $\mathbf{E}_{0:T}$ for all time steps, we are interested in the most probable assignment for all PRVs $\mathbf{V}$ in G for which there is no evidence specified. Thus, the temporal lifted MPE problem is to solve the following expression $\arg\max_{\mathbf{v}} P((\mathbf{V} = \mathbf{v})_{|C} | \{\mathbf{E}_i = \mathbf{e}_i\}_{i=0}^{T})$, $\mathbf{V} \cap \mathbf{E} = \emptyset$.

The basic idea of solving the temporal lifted MPE problem is also to use a maximisation instead of summation, which LDJT uses for marginal queries. The Viterbi algorithm is

Algorithm 8 LDJT^{mpe} for PDM G and Evidence $\mathbf{E}_{0:T}$

procedure LDJT$^{mpe}(G, \mathbf{E}_{0:T})$
  Build FO jtree J_0 and J_t using G
  $t := 0$
  while $t \neq T + 1$ **do**
    Recover previous state from temporal assignment message γ_{t-1}, $\gamma_0 = \emptyset$
    Enter $\mathbf{E}_t$ in J_t
    Perform an *inbound* message pass on J_t with *out-cluster* as root
    Calculate temporal assignment message γ_t based on *out-cluster* of J_T
  Answer MPE query using *out-cluster* of J_T

one approach to solve the temporal propositional MPE problem by applying a max-product algorithm instead of a sum-product algorithm (Russell and Norvig, 1995)

Algorithm 8 outlines how LDJT^{mpe} efficiently solves the temporal lifted MPE problem. The basic idea for LDJT^{mpe} is a combination of LDJT and LJT^{mpe}. The first step in Alg. 8 is to construct the FO jtree structures J_0 and J_t as described in Alg. 1. Using the structures, LDJT^{mpe} enters a loop where for every time step, it recovers the previous state, enters evidence for the current time step, performs an *inbound* message pass with the *out-cluster* as root, and calculates γ_t to preserve the current time step. The main differences are that LDJT^{mpe} calculates γ_t by maxing out instead of α_t by summing out and that LDJT^{mpe} only performs an *inbound* message pass with the *out-cluster* as root. After LDJT^{mpe} propagates all the information to the last time step and leaves the loop, LDJT^{mpe} answers the MPE query.

To calculate γ_t, LDJT^{mpe} maxes out all non-interface PRVs from the *out-cluster* of J_t. As LDJT^{mpe} only needs to answer one MPE query, over all time steps, it suffices to always perform an *inbound* message pass with the *out-cluster* as root. This way, the *out-cluster* has all information required to calculate γ_t. For the last time step, LDJT^{mpe} uses the *out-cluster* to answer the MPE query. LDJT^{mpe} answers the MPE query by maxing out the PRVs of the *out-cluster* and reading out the most probable assignment to all PRVs for which there is no evidence.

Example 8.2.1 (Answering temporal MPE queries with LDJT^{mpe})**.** *For our example PDM G^{ex}, LDJT^{mpe} first builds FO jtree structures J_0^{ex} and J_t^{ex}, which are the same structures as for marginal queries. For time step* 0*, LDJT^{mpe} does not have a previous state to recover. Thus, LDJT^{mpe} directly enters evidence for time step* 0 *in J_0^{ex}. On J_0^{ex}, LDJT^{mpe} performs an* inbound *message pass with the* out-cluster *as root and calculates γ_0 by maxing out all non-interface PRVs from the* out-cluster*. For time step* 1*, LDJT^{mpe} instantiates J_1^{ex} from J_t^{ex}. Figure 3.7 depicts the FO jtree instantiations for time step* 3 *and* 4*. LDJT^{mpe} adds γ_0 to the* in-cluster*, $\mathbf{C}_1^1$, of J_1^{ex}, enter evidence for time step* 1*, and performs an* inbound *message pass with $\mathbf{C}_1^2$ as root. Hence, LDJT^{mpe} calculates*

two messages, namely m_1^{12} *and* m_1^{32}*. With* m_1^{12} *and* m_1^{32}*,* $\mathbf{C}_1^2$ *holds all the information necessary to calculate* γ_1*, by maxing out* $Pub_1(X, J)$*. LDJT*^{mpe} *proceeds in this fashion until it reaches the last time step* T*. In* J_T^{ex}*, the* out-cluster $\mathbf{C}_T^2$ *holds all the information to answer the MPE query. After maxing out* Hot_T*,* $Att_T(X)$*, and* $Pub_T(X, J)$*, LDJT*^{mpe} *can directly read out the assignments for all PRVs from each time step.*

Now, we investigate temporal lifted MAP queries.

8.2.2 MAP Queries

First, we define temporal MAP queries and then investigate how LDJT^{map} solves the temporal lifted MAP problem. LDJT^{map} is a combination of LDJT, for lifted summing out, and LDJT^{mpe}, for lifted maxing out.

Definition 8.2.2 (Temporal MAP problem)**.** Given a PM G, evidence $\mathbf{E}_{0:T}$, and a set $\mathbf{U}_{|C'} \subseteq \mathbf{V}_{|C}$, an *MAP problem* refers to the problem of finding an assignment for $\mathbf{U}_{|C'}$ with the highest probability w.r.t. P_G, i.e., given $\mathbf{S}_{|C''} = \mathbf{V}_{|C} \setminus \mathbf{U}_{|C'}$: The temporal lifted MAP problem is to solve the following expression $\arg\max_{\mathbf{u}} P((\mathbf{U} = \mathbf{u})_{|C'} | \{\mathbf{E}_i = \mathbf{e}_i\}_{i=0}^T) = \arg\max_{\mathbf{u}} \sum_{\mathbf{s} \in \mathbf{S}_{|C''}} P((\mathbf{U} = \mathbf{u})_{|C'}, (\mathbf{S} = \mathbf{s})_{|C''} | \{\mathbf{E}_i = \mathbf{e}_i\}_{i=0}^T)$.

LDJT^{map}, a combination of LDJT and LDJT^{mpe}, answers MAP queries efficiently. an MAP query over a parcluster is also harmless for LDJT^{map}. One feature of LDJT is that an α message separates one time step from the next. Thus, LDJT^{map} can efficiently answer MAP queries over complete time steps and reuse α messages computed while answering marginal queries. In the case of MAP queries over complete time steps, the non-commutativity of summing out and maxing out does not lead to a restriction of the elimination order. Hence, using LDJT^{map}, we can easily identify that MAP queries over complete time steps are harmless. Additionally, LDJT^{map} reuses computations from marginal queries to answer MAP queries over complete time steps.

Example 8.2.2 (Harmless temporal MAP queries)**.** *Assume we are only interested in the assignment of all PRVs from the last* 20 *time steps. Then, LDJT*^{mpe} *can instantiate* J_{T-20}*, add* α_{T-21}*, start solving an MPE, and read out the most probable assignments for the last* 20 *time steps at the* out-cluster *of* J_T*. The* α_{T-21} *message includes all necessary state descriptions corresponding to the summing out of all PRVs from time step* 0 *to* $T - 21$ *(without the interface variables being summed out for time step* $T - 21$*).*

By using α messages to separate time steps and identify harmless MAP queries, LDJT^{map} actually does not return the assignments of the queries for the last t time steps, but also for the interface variables of time step $t-1$. However, using α messages is convenient as LDJT calculates them anyhow and they make it easy to identify harmless MAP queries.

8.2.3 Discussion

This section discusses how LDJT^{mpe} and LDJT^{map} efficiently handle temporal aspects compared to LJT^{mpe} and LJT^{map} for MPE and MAP queries.

MPE Queries If we unroll a PDM into a PM and use LJT^{mpe}, the FO jtree would not necessarily be constructed in a way to handle the temporal aspects efficiently and thus, would have an impact on the performance. However, if we unroll an FO jtree based on the structures J_0 and J_t from LDJT^{mpe}, then from a computational perspective, LJT^{mpe} and LDJT^{mpe} would perform the same calculations possibly in a different order. By selecting the *out-cluster* of the last time step as the root for LJT^{mpe}, both algorithms would actually compute exactly the same messages.

Nonetheless, there still is a difference in the memory consumption. LJT^{mpe} needs to store the complete unrolled FO jtree. Thus, all messages, evidence, and local models for all time steps need to be stored in memory, which might not always be feasible for a high number of maximal time steps. Further, the search space for the next operation to perform is significantly smaller. Overall, from a computational point of view, LJT^{mpe} only performs as well as LDJT^{mpe} does if using the FO jtree construction of LDJT^{mpe}. Irregardless, the memory consumption of LDJT^{mpe} is always significantly lower due to efficiently handling temporal aspects.

MAP Queries As we have already mentioned, assignment queries over complete time steps are safe for LDJT^{map} either the last few time steps using an α message or time steps in between other time steps using an α and a β message. Additionally to being safe, these queries could also be of high interest as often one might not be interested in the assignment of all PRVs, but only in the PRVs of the last few time steps. Further, while answering marginal queries with LDJT, LDJT^{map} can simply store the calculated α messages for assignment queries. Thus, for MAP queries over complete time steps, LDJT^{map} reuses computations and only needs to store one FO jtree at a time.

To answer MAP queries over complete time steps with LJT^{map}, we again could provide it with an unrolled FO jtree, analogous to MPE queries. Here, we would again have overhead on the memory consumption. Nonetheless, also with LJT^{map}, we could use the message pass of LJT, which includes a complete message pass with *inbound* and *outbound* phase, and apply LJT^{mpe} on the time steps for which we want to know the assignment. However, LDJT normally computes fewer messages compared to LJT for temporal models. Hence, LDJT^{map} significantly outperforms LJT^{map} for MAP queries.

8.3 Theoretical Analysis

This section investigates soundness, completeness, and complexity of LDJT^{mpe} and LDJT^{map}. The results are highly similar to the results of LDJT.

8.3.1 Soundness

To show soundness of LDJT^{mpe}, we show that LDJT^{mpe} and LJT^{mpe} with an unrolled model perform the same calculations and therefore, compute equivalent results.

Theorem 8.3.1. *LDJT^{mpe} is sound, i.e., LDJT^{mpe} computes the same results as LJT^{mpe} does for an unrolled PDM.*

Proof. LJT^{mpe} is sound. Basically, LDJT^{mpe} unrolls an FO jtree for T time steps and performs an *inbound* message pass. Thus, LDJT^{mpe} produces the same result as unrolling an FO jtree, providing it to LJT^{mpe}, and LJT^{mpe} performing an inbound message pass with one special parcluster as root. The calculations are equivalent. Hence, as LJT^{mpe} is sound, LDJT^{mpe} is also sound. □

LDJT^{map} is a combination of LDJT and LDJT^{mpe}.

Theorem 8.3.2. *LDJT^{map} is sound, i.e., LDJT^{map} computes the same results as LJT^{map} does for an unrolled PDM.*

Proof. Given both LDJT and LDJT^{mpe} are sound, soundness of the combination of both for queries over time steps follows. □

8.3.2 Completeness

We begin by showing that the completeness results of LDJT are transferable to LDJT^{mpe}. For LDJT^{map}, we only show completeness for assignment queries over complete time steps, i.e., queries where summing out turns to maxing out are after an α message.

The main difference between LDJT and LDJT^{mpe} is whether the approach uses lifted summing out or lifted maxing out. These two operators have the same preconditions from a lifting point of view, i.e., in case one of the two operators can be applied without having to ground the other operator can also be applied without having to ground.

Theorem 8.3.3. *LDJT^{mpe} is complete for the same models as LDJT.*

Proof. In case lifted summing out can be applied, lifted maxing out can also be applied. Thus, LDJT and LDJT^{mpe} can obtain a lifted solution for the very same models. □

LDJT^{map} is a combination of LDJT and LDJT^{mpe}. Unfortunately, summing out and maxing out are not commutative. Thus, additional restrictions on the elimination order and thereby, completeness exists. We focus on showing completeness for LDJT^{map} when asking for assignments of PRVs from complete time steps.

Theorem 8.3.4. *LDJT^{map} is complete for the same models as LDJT and LDJT^{mpe}, in case we ask assignment queries over complete time steps.*

Proof. With assignment queries over complete time steps, LDJT^{map} can use the fact that LDJT has already summed out all other variables to calculate the corresponding α and β message. Thus, LDJT^{map} consists for these queries of performing LDJT for the first time steps and start to perform LDJT^{mpe} on a newly instantiated FO jtree for the remaining time steps. Thereby, assignment queries over complete time steps do not impose additional restrictions on the elimination order. Hence, the completeness results of LDJT and LDJT^{mpe} can also be applied to LDJT^{map} for assignment queries over complete time steps. □

8.3.3 Complexity

There are two main differences between LDJT and LDJT^{mpe}, namely lifted summing out and a complete message pass for each FO jtree against lifted maxing out and only an *inbound* message pass for each FO jtree. However, these differences do not change the complexity. Lifted summing out and lifted maxing out have the same complexity. Both operators need to eliminate PRVs and the complexity depends on the number of rows. Performing a complete message pass compared to only an *inbound* message pass differs only in a constant factor. Further, Lemma 4.3.5 already does not include the $2 \cdot (n_J - 1)$ for the complete message pass, but only n_J as all other factors are constant. Therefore, the complexity results of LDJT can be directly transferred to LDJT^{mpe}.

LDJT^{map} is a combination of LDJT and LDJT^{mpe}. From a complexity point of view, LDJT and LDJT^{mpe} are the same. Thus, the complexity results from LDJT and LDJT^{mpe} can also be directly transferred to LDJT^{map}.

8.4 Evaluation

For the evaluation, we use G^{ex} as described in Section 5.1. Further, we vary the domain size for the logvar X, i.e., $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$, while setting $|\mathcal{D}(P)| = 3$. Additionally, we vary the maximum number of time steps T from 10 to 10000. Figure 8.1 shows the runtimes for the corresponding MPE queries as well as MAP queries always asking for the assignment of the last 20 time steps. We only present runtimes for LDJT as LJT and LDJT perform the very same calculations to answer MPE queries, because both perform only an *inbound* message pass. The only difference between LJT and LDJT w.r.t. MPE queries is the memory consumptions but not calculations. For MAP queries, the benefit of LDJT over LJT from Chapter 5 still hold. LDJT answers MAP queries faster compared to LJT, but we would only reproduce the results from Chapter 5. Thus, we only present runtimes for LDJT.

In Fig. 8.1, we can see that the runtimes for the MAP queries are as to be expected linear w.r.t. the maximum number of time steps. The runtimes of MPE queries up to about 1.000 are also as expected linear w.r.t. the maximum number of time steps. However, for larger T, the runtimes of LDJT are no longer linear. By increasing T,

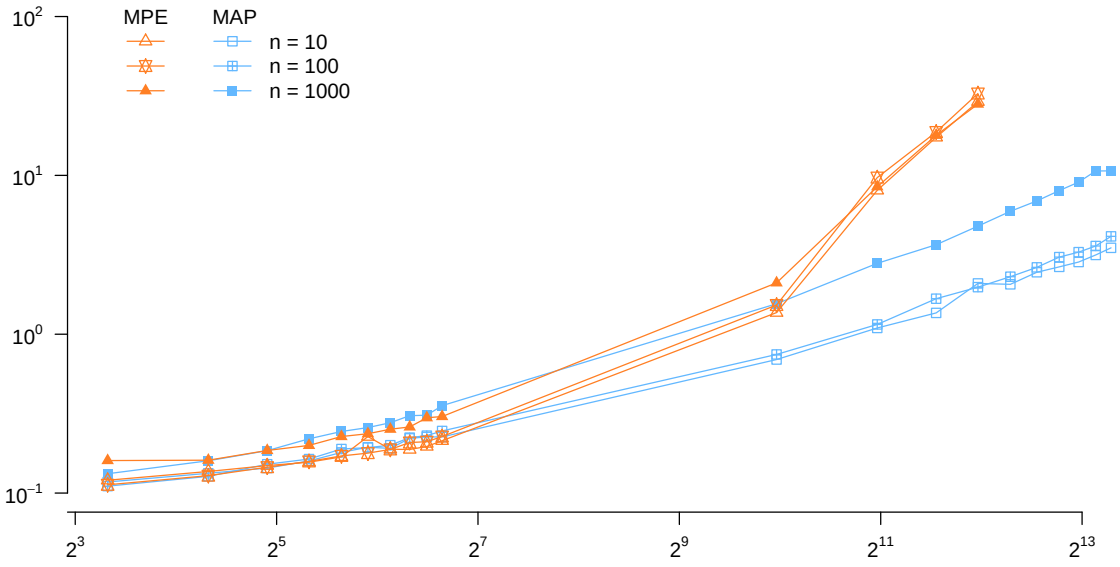


Figure 8.1: MPE and MAP runtimes [seconds, log], x-axis: time steps, log

LDJT needs to store assignments for an increasing number of PRVs. The assignments are part of the messages. Thus, the size of messages grow w.r.t. T as more assignments need to be stored. The growing message size and the addition of even more assignments is also the reason for the behaviour of MPE queries. Even though, compared to LJT, LDJT is memory efficient as it only keeps one time step in memory, LDJT runs into memory problems for 5000 and more time steps.

Overall, we can say that the runtimes for the MAP are as to be expected. However, as the number of stored assignments grow with the maximum number of time steps, LDJT is not linear w.r.t. the maximum number of time steps for MPE queries.

8.5 Interim Conclusion

We present LDJT^{mpe} to efficiently solve the temporal MPE problem for temporal probabilistic relational models (Contribution **5a**). The idea is to use lifted maxing out instead of lifted summing out, which LDJT uses, to eliminate PRVs. Additionally, we show that LDJT^{map} (Contribution **5b**) can efficiently answer MAP queries over the last x time steps (Contribution **5c**). Further, by comparing LDJT and LDJT^{mpe} against LJT and LJT^{mpe} on a theoretical level, we show that an efficient handling of temporal aspects is necessary and that LDJT and LDJT^{mpe} significantly outperform LJT and LJT^{mpe} for temporal models.

Chapter 9

Maximum Expected Utility

The last extension to the query language that we investigate deals with expected utility queries. With the last extension, we take the first steps towards enabling LDJT to also support decision making. LDJT in its basic form cannot reason about optimal actions w.r.t. some predefined rewards or utilities. LDJT can answers marginal queries and from these results, one could possibly also determine which action one probably wants to perform next. However, evaluating all marginals for at least one PRV by hand can be quite cumbersome. Therefore, we now extend PMs and PDMs with actions and utilities, to directly query what the best action is. For example, one could ask which option is better to improve ones reputation with the options being attending a conference now without a publication and spending time on doing research to possibly later attend a conference with a publication.

The general idea of expected utility queries is similar to marginal queries. The main difference is that in addition to obtaining the current belief state of PRVs, these belief states are multiplied with corresponding predefined utility values. By selecting an action that maximises the expected utility, one can solve the MEU problem. Thus, the general problem of expected utility queries is to calculate the best action, i.e., the action that maximises the expected utility given evidence.

In this chapter, we present parameterised probabilistic decision models (PDecMs) and parameterised probabilistic dynamic decision models (PDDecMs). Further, we introduce LJT^{meu} to answer multiple marginal and expected utility queries efficiently for static probabilistic relational models and LDJT^{meu} to answer multiple marginal and expected utility queries efficiently for temporal probabilistic relational models. By calculating an exact solution to the MEU problem, LJT^{meu} and LDJT^{meu} can be used to explain the suggested decision. For an exact solution, the effect of each possible action needs to be checked. Thus, LJT^{meu} and LDJT^{meu} check all influences of each action, making it explainable why the algorithms suggest an action. Further, asking additional marginal queries allows for making the decisions explorable. In this context, explorable means that one can ask additional queries to better understand the suggested action. Hence, LJT and LDJT answering multiple marginal queries efficiently allows for making the decisions efficiently explorable.

For static models, Nath and Domingos (2009) introduce Markov logic decision net-

works (MLDNs), which include action and utility nodes. Nath and Domingos calculate approximate solutions to the static MEU problem in a completely grounded way (Nath and Domingos, 2010b) based on MLDNs. Another approach of Nath and Domingos is approximate (Nath and Domingos, 2010a). Further, Apsel and Brafman (2011) propose an exact lifted solution to the MEU problem based on the work by Nath and Domingos (2009). These approaches are designed to handle single queries. However, we propose to answer multiple queries efficiently, i.e., a combination of marginal and expected utility queries. Van den Broeck *et al.* (2010) propose an approach that in theory could answer multiple queries efficiently. Even though the approach only defines utility queries, the compiled structure could in theory be used to also also answer marginal queries. The approach of Van den Broeck *et al.* (2010) provides exact and approximate solutions to the MEU problem. However, they do not employ any lifting techniques. Unfortunately, to obtain an exact solution an algorithm has to iterate over all actions. While iterating over all possible actions to calculate an exact solution to the MEU problem, our approach can reuse the FO jtree structures and computations

Additional research focuses on sequential decision making by investigating first-order (partially observable) Markov decision processes (FO (PO)MDPs) (Sanner and Boutilier, 2007; Joshi *et al.*, 2009; Sanner and Kersting, 2010). In contrast to FO POMDPs, which support *offline* decision making, we propose to support probabilistic *online* decision making, which allows for reacting to observations as well as for query answering. Additionally, our approaches allow for asking additional marginal queries.

In the following, we present PDecMs and LJT^{meu}. Afterwards, we show how LDJT^{meu} solve the temporal lifted MEU problem with PDDecMs. Lastly, we theoretically and empirically evaluate LJT^{meu} and LDJT^{meu} and conclude by looking at possible extensions.

This chapter is based on the following publications:

> Marcel Gehrke, Tanya Braun, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhäuser. Towards Lifted Maximum Expected Utility. In *Proceedings of the Joint Workshop on Artificial Intelligence in Health in Conjunction with the 27th IJCAI, the 23rd ECAI, the 17th AAMAS, and the 35th ICML*, pages 93–96. CEUR-WS.org, 2018
>
> Marcel Gehrke, Tanya Braun, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhäuser. Lifted Maximum Expected Utility. In *Proceedings of Artificial Intelligence in Health*, pages 131–141. Springer International Publishing, 2019
>
> Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Temporal Maximum Expected Utility. In *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*, pages 380–386. Springer, 2019

In addition to the contributions of the publications, we also present a theoretical analysis and also empirical results for LJT^{meu} and LDJT^{meu} application.

9.1 Lifted Maximum Expected Utility

In this section, we introduce actions and utilities for PMs forming PDecMs, define the MEU problem for PDecMs, and show how LJT^{meu} solves the MEU problem.

9.1.1 Parameterised Probabilistic Decision Models

Let us extend PMs with action and utility nodes, resulting in PDecMs.

Definition 9.1.1 (PDecM)**.** Let Φ^u be a set of utility factor names. The range of action PRVs is a set of disjoint actions and the range of utility PRVs is $\mathbb{R}$. A parfactor with a utility PRV U is a *utility parfactor* u. We denote u with $\mu(\mathcal{A})_{|C}$, where $U \in \mathcal{A}$ and C a constraint on $lv(\mathcal{A})$. Function $\mu : \times_{A^i \in (\mathcal{A} \setminus \{U\})} \mathcal{R}(A^i) \mapsto \mathbb{R}$, $\mu \in \Phi^u$, is identical for $gr(\mathcal{A}_{|C})$. Its output is the additive change of U's value and we only have one U. The default initial utility value is 0. A *PDecM* G is a PM that also contains utility parfactors. Let G^u refer to the utility parfactors in G and $rv(G^u)$ refer to all probability PRVs in G^u. G^u represents the combination of all utilities $U_G = \sum_{f \in gr(G^u)} f$.

μ functions return a utility, i.e., a scalar, which makes comparing utility values easy. After the evaluation of a μ function, the initial value $U = i$ is changed by the output value, j, resulting in the new value $U = i + j$. As a utility parfactor contains a utility PRV, the functions does not use the utility PRV as input but solely as output.

Example 9.1.1 (Utilities)**.** *With utilities we can model influences of doing research, publishing, and attending a conference w.r.t. a topic being hot. If an individual only does research without publishing and attending conferences, we can model that the individual gets a negative utility. However, if an instance publishes and attends conferences, then that individual gets a positive (relatively high) utility. Only attending conferences could also have a slightly positive influence on the utility of the instance and whether a topic is hot. Similar for all other possible combinations we have to assign a utility value.*

With utilities incorporated, we look at actions. To model actions, we introduce an action PRV with the actions in its range. Hence, we have one PRV modeling disjoint actions. To execute an action, we set the value of the action PRV to the action, which we want to perform, similarly to providing evidence for marginal queries. Thus, the range of an action PRV $B(X)$ consists of different actions, lets say $b^1, ..., b^n$, and by setting $B(X)$ to the action, let us say b^1, i.e., $B(X) = b^1$, we can select an action.

Example 9.1.2 (Actions and utilities)**.** *We extend the example with actions and utilities. In Fig. 9.1, we can see one action node (square), one utility node (diamond), and one utility parfactor (crosses). The action PRV $B(X)$ has two actions in its range, namely b^1, submit a paper, and action b^2, keep on researching for writing a paper later on.*

The state of a current paper from a researcher and $B(X)$ influence the utility. There are two possibilities, namely submitting the paper if one feels the paper is ready for publication

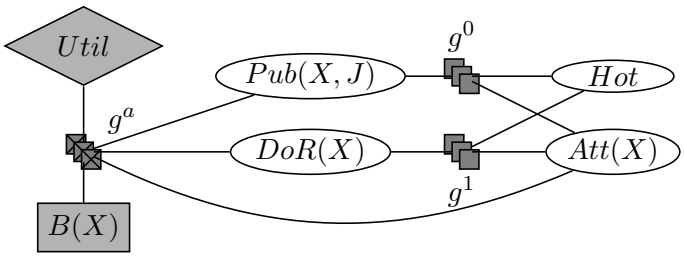


Figure 9.1: PDecM of G^{ex}

or keeping on working on the paper to improve it. One can decide to keep working on the paper, even though the paper might be ready for publication. By doing so, it is possible that other researchers come up with a similar idea, rendering the own paper outdated. Thus, one always needs to consider that submitting a paper too early might delay the publication of that paper unnecessarily and submitting a paper too late can render that paper obsolete.

Now, we take a look at how one can calculate a best action w.r.t. a PM and given utilities.

9.1.2 Maximum Expected Utility

To select the best action, we define expected utility queries on a PDecM.

Definition 9.1.2 (Expected utility query)**.** Given a PDecM G, a query term Q, and events $\mathbf{E}$, the expression $P(Q|\mathbf{E})$ denotes a *probability query* w.r.t. P_G. Given an assignment $\mathbf{a}$ for an action in G, the expression $U(Q, \mathbf{E}, \mathbf{a})$ refers to a *utility* w.r.t. U_G. The *expected utility* of G is defined by

$$eu(G|\mathbf{E}, \mathbf{a}) = \sum_{v \in range(rv(G^u))} P(v|\mathbf{E}, \mathbf{a}) \cdot U(v, \mathbf{E}, \mathbf{a}) \tag{9.1}$$

The inner part of the summation in Eq. (9.1) calculates a belief state $P(v|\mathbf{E}, \mathbf{a})$ and combines it with corresponding utilities $U(v, \mathbf{E}, \mathbf{a})$. By summing over all randvars from G^u, one obtains a scalar representing the expected utility. LVE allows for exactly computing an expected utilities. Based on expected utility, we define the MEU as follows.

Definition 9.1.3 (MEU problem)**.** Given a PDecM G and evidence $\mathbf{E}$, the MEU problem is given by

$$meu[G, \mathbf{E}] = \left(\arg\max_{\mathbf{a}} eu(G|\mathbf{E}, \mathbf{a}), \max_{\mathbf{a}} eu(G|\mathbf{E}, \mathbf{a}) \right) \tag{9.2}$$

Equation (9.2) suggests a naive algorithm for calculating an MEU, namely by iterating over all possible action configurations, computing an expected utility for each configuration using LVE, an iteration that one cannot avoid if asking for an exact solution. The action assignment that maximises the expected utility is selected. As the utility value is a scalar, the expected utility w.r.t. configurations can be easily compared. Therefore, we also can easily determine configurations whose expected utility lie within an ϵ margin, making the actions hardly discriminable w.r.t. utilities.

Example 9.1.3 (Maximum Expected Utilities)**.** *The action PRV in G^{ex} has two possible actions. By setting $B(X) = b^1$, we turn on B^1. By setting $B(X) = b^2$, we turn on B^2. Thus, in our example to calculate the MEU, we need to iterate over two action assignments. For each expected utility, we obtain a scalar, allowing us to easily compare them and return the action with the MEU and the expected utility value. If all researcher/paper behave the same, we only need to iterate over two actions. In case we obtain different evidence for, say, two groups of researchers, X^1 and X^2, we need to iterate over the actions for both groups. Hence, we would need to iterate over $\{B(X^1) = b^1, B(X^2) = b^1\}$, $\{B(X^1) = b^2, B(X^2) = b^1\}$, $\{B(X^1) = b^1, B(X^2) = b^2\}$, and $\{B(X^1) = b^2, B(X^2) = b^2\}$. In general, we need to iterate over r^n actions, where r is the number of actions in the range of an action PRV and n the number of different groups. Assuming, we have ten researchers in two groups and two possible actions. Solving the MEU in a lifted way, we need to iterate over $2^2 = 4$ actions. Without the lifting idea, we would need to iterate over $2^{10} = 1024$ actions. Therefore, solving the MEU problem in a lifted way makes the problem manageable.*

To solve the MEU problem, we can extend LJT to meuLJT. Allowing utility parfactors in parclusters is straightforward. Basically, we only need to consider that G is not a PM anymore but a PDecM and that the local models are composed of utility and probability parfactors of G. While constructing the FO jtree, meuLJT treats the utility parfactor in the same way as probability parfactors. With the parclusters, meuLJT distributes state descriptions of all parfactors by message passing. In this case meuLJT excludes the utility parfactor from the message pass. As only one utility parfactor exists in a PDecM, meuLJT also does not need to perform a message pass for utility values. Then,

Algorithm 9 meuLJT for a PDecM G, Queries $\{\mathbf{Q}\}$, Evidence $\{\mathbf{E}\}$

procedure MEULJT($G, \{\mathbf{Q}\}, \{\mathbf{E}\}$)
  $J :=$ FO-JTREE(G)
  $J :=$ EnterEvidence($J, \mathbf{E}$)
  $J :=$ PassProbMessages(J)
  AnswerMEUQuery(J)
  AnswerQueries($J_t, \mathbf{Q}_t$)

the parcluster with the utility parfactor in its local model can answer utility queries. Algorithm 9 outlines the described idea of meuLJT.

Example 9.1.4 (meuLJT)**.** *In our example. the action PRV $B(X)$ is only connected to the utility parfactor. Thus, $B(X)$ does not influence the belief state of $Att(X)$, $DoR(X)$, $Pub(X, J)$, or Hot. In such a case, the efficiency of meuLJT can benefit from the underlying FO jtree. For example, the FO jtree has one parcluster with only the utility parfactor in the local model, then meuLJT passes the probability belief state to that parcluster during message passing. Additionally, selecting another action does not change that belief state. Hence, the belief state still holds when selecting a new action and meuLJT does not have to perform a message pass again. Thus, the effort to calculate a new expected utility boils down to answer a new query without having to perform another message pass.*

Knowing how meuLJT can calculate the best action, let us extend the problem to the temporal case and present meuLDJT.

9.2 Lifted Temporal Maximum Expected Utility

We define PDDecMs with action and utility nodes to support temporal decision making. Finally, we define the temporal MEU problem for PDDecMs and introduce meuLDJT.

9.2.1 Parameterised Probabilistic Decision Models

For a PDDecM, which extends a PDecM to the temporal case, we define a utility transfer function to connect two utility PRVs from different time steps.

Definition 9.2.1 (PDDecM)**.** A utility transfer function λ has utility PRVs $\mathbf{U}$ as input and one utility PRV U_o as output. Additionally, λ can have non utility PRVs as input. λ specifies how the value of U_o is additively changed when transferring from time step t to $t+1$. A *PDDecM* is a pair of PDecMs $(G_0, G_{\rightarrow})$ with $G^u_{\rightarrow}$ also possibly containing utility transfer functions. Given a PDDecM G, a *temporal MEU query* asks for the action sequence (range values for each action PRV in G over time) that maximises the overall expected utility in G up to the current time step.

Example 9.2.1 (PDDecM)**.** *Figure 9.2 shows a PDDecM for our example. The main difference to Fig. 9.1 is that now, we can also see a utility transfer parfactor g^U in black. Further, as the action influences whether one keeps on working on the current paper, the action now also influences $DoR(X)$ in the next time step. PDDecMs encode trade-offs of performing actions, e.g., submitting now or keeping to work on the paper, in transfer utility parfactors. Connecting $Util_{t-1}$ and $Util_t$ with a utility transfer parfactor g^U makes utility PRVs time-dependent and allows for discounting. For example, g^U specifies that the value of $Util_{t-1}$ is reduced by 5 and then added to $Util_t$.*

Algorithm 10 meuLDJT for a PDDecM G, Queries $\{\mathbf{Q}\}_{t=0}^{T}$, Evidence $\{\mathbf{E}\}_{t=0}^{T}$, and Horizon h

procedure MEULDJT($G_0, G_{\rightarrow}, \{\mathbf{Q}\}_{t=0}^{T}, \{\mathbf{E}\}_{t=0}^{T}, h$)
  $(J_0, J_t, \mathbf{I}_t)$:= DFO-JTREE($G_0, G_{\rightarrow}$)
  while $t \neq T + 1$ **do**
    J_t := LJT.EnterEvidence($J_t, \mathbf{E}_t$)
    J_t := LJT.PassProbMessages(J_t)
    J_t := LJT.PassUtilMessages(J_t)
    AnswerQueries($J_t, \mathbf{Q}_t$)
    AnswerMEUQuery(J_t, h)
    $(J_t, t, \alpha[t-1])$:= ForwardPassJ_t, t)

Equation (9.2) defines how to calculate the MEU for a PDecM and a PDDecM. Using a utility transfer function, we see the problem as an iterative filtering problem. The expected utility is calculated for one time step and then the utility value is transferred to the next time step. Therefore, the utility value of the latest time step is the overall utility value. Due to the inherent uncertainty of PDDecMs, calculating the best actions is only feasible for a finite horizon as one needs to iterate over all possible action assignments.

Next, we investigate how meuLDJT can solve the temporal lifted MEU problem.

9.3 Solving the MEU Problem with meuLDJT

We illustrate how meuLDJT incorporates utilities and solves the MEU problem for one utility PRV and parfactor for each time step.

Including Utilities Algorithm 10 outlines how meuLDJT includes utilities. Similar to LDJT for PDMs, meuLDJT first builds FO jtree structures for a PDDecM. As mentioned earlier, allowing utility parfactors in parclusters is straightforward. While construct-

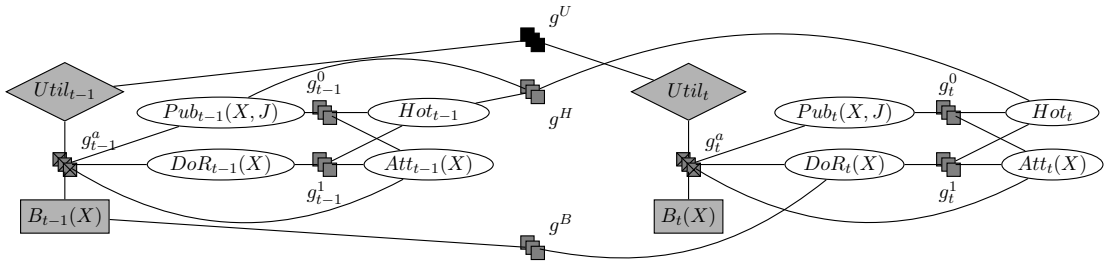


Figure 9.2: PDDecM of G^{ex}

ing the FO jtree, meuLDJT treats the utility parfactor in the same way as probability parfactors. With the parclusters, meuLDJT distributes local information by message passing. To calculate the probability messages, meuLDJT excludes utility parfactors as they do not influence the probability distributions. Using the probability messages, meuLDJT can calculate the current utility value and can distribute the value as long as the utility PRV is in the separator. To calculate utilities, utility parclusters need to know the probability distributions, which are distributed to each parcluster during message passing. Using the probability distributions, for each group meuLDJT calculates a utility value and multiplies the value by the number of groundings. The new utility value is then added to the old utility value. Further, the utility transfer function ensure the transfer of utility values, while the behaviour of LDJT ensures preserving the current state.

Answering MEU Queries meuLDJT can answer MEU queries for a finite horizon. The horizon defines how far meuLDJT predicts the future. For a given horizon, meuLDJT tests all action sequences to find the best action sequence. To do so, meuLDJT constructs all action sequences for a horizon. meuLDJT constructs r_a^{h+1} action sequences, where r_a is the overall range of action PRVs, i.e., number of actions and h the horizon. For each action sequence, meuLDJT enters the sequence as evidence and answers the expected utility query for that sequence. Finally, after having tested all action sequences, meuLDJT returns the best action sequence and the expected utility value.

Example 9.3.1 (Calculating utilities over time)**.** *Assume that* $t = 3$ *and we only have a horizon of* 1 *to answer an MEU query. First, meuLDJT constructs the action sequences. With the two range values of our action PRV* $B(X)$*, there are four action sequences. For example, the first action sequence is* $B_3(X) = b^1$ *and* $B_4(X) = b^1$*. Then, meuLDJT enters* $B_3(X) = b^1$ *in the FO jtree for time step* 3*. Message passing on the FO jtrees is performed in two steps. The first step is to calculate probability messages. The second step is to calculate utility messages. meuLDJT uses the probability messages and the evidence, which includes the current action, and distributes the utility through the FO jtree. After message passing, each parcluster can answer queries about its PRVs. Thus, meuLDJT can answer multiple probability and utility queries efficiently as it can reason over representatives. To proceed in time, meuLDJT uses the* out-cluster *to calculate an* α *message over the interface PRVs, which now includes* $Util_t$*. Hence, the current belief state and utility value is stored in the* α *message and then added to the* in-cluster *of the next time step. Using the utility transfer function, the current FO jtree contains the overall utility value. Next, meuLDJT enters* $B_4(X) = B^1$ *in the FO jtree for time step* 4 *and performs the two message passes. Last, meuLDJT calculates the expected utility value, stores it to for comparison, and proceeds with the next action sequence.*

For the next sequence, $B_3(X) = B^1$ *and* $B_4(X) = B^0$*, meuLDJT can reuse previous calculations. meuLDJT has already calculated the expected utility and probabilities for*

the subsequence $B_3(X) = B^1$. Thus, meuLDJT can directly continue with $B_4(X) = B^0$ by using the previously calculated α_3 for $B_3(X) = B^1$. meuLDJT calculates the expected utility value for the sequence $B_3(X) = B^1$ and $B_4(X) = B^0$ and proceeds until the expected utility values for all four sequences are calculated. Finally, returns the action sequence with the MEU as well as the corresponding utility value.

As seen in the example, meuLDJT does not need to start from scratch for each utility query, but can reuse previous calculations, which can be significant for large subsequences. Thereby, meuLDJT helps to make the combinatorial problem behind solving the temporal MEU problem more manageable. Additionally, by calculating an exact solution to the temporal MEU problem, the proposed best action sequence of meuLDJT is explainable. meuLDJT calculates the expected utility for each action sequence for a finite horizon. Thus, the output is the action sequence with the highest expected utility. To make the proposed action sequence even more explainable, the sequence can be explored by asking additional marginal queries. As meuLDJT is based on LDJT, meuLDJT can efficiently answer multiple marginal queries. The marginal queries can be used to check a gut feeling against the best action sequence. Further, one could also ask marginal queries for action sequences that one would expect to be good action sequences. Hence, with meuLDJT, one can explain and explore action sequences. Now, let us have a look at the theoretical analysis of meuLDJT.

9.4 Theoretical Analysis

This section investigates soundness, completeness, and complexity of meuLDJT.

9.4.1 Soundness

To investigate the soundness of meuLDJT, we first look into meuLJT and show that meuLJT is sound for one utility parfactor.

Theorem 9.4.1. *meuLJT is sound, i.e., it produces the same best action as any exact ground algorithm, for one utility parfactor.*

Proof. LJT is sound and meuLJT uses LJT for the probability calculations. For the message pass, meuLJT only takes probability parfactors into account. Leading to the very same messages as if the utility parfactor would not be in the model. Thus, marginal queries are still sound. To calculate an expected utility, meuLJT needs to calculate a belief state of the PRVs from the utility parfactor. As the marginal queries are sound, the belief state of the PRVs is also sound. Hence, by multiplying the belief state with the utility parfactor, accounting for groundings by eliminating logvars, which LJT does, and summing over the range values, meuLJT soundly calculates an expected utility. By iterating over all possible actions, meuLJT solves the temporal MEU problem. Further,

meuLJT calculates the same result as any ground algorithm, as meuLJT is based on sound computations of LJT. □

One difference between meuLJT and meuLDJT is that meuLDJT has more than one utility parfactor because meuLDJT has one utility parfactor for each time step. Thus, for meuLDJT, we also show that passing utility values is sound.

Theorem 9.4.2. *meuLDJT is sound, i.e., it produces the same best action sequence as any exact ground algorithm.*

Proof. LDJT is sound and meuLDJT uses LDJT for the probability calculations. Thus, marginal *hindsight*, *filtering*, and *prediction* queries are still sound. To calculate the new utility value, meuLDJT calculates the utility value and adds it to the old utility value. While calculating the utility value, meuLDJT accounts for the groundings, namely it calculates the utility value for one representative and multiplies it by the number of groundings. As all instances behave the same, each would contribute the same utility value in a ground model. Hence, by calculating a utility for one representative and multiplying the utility by the number of groundings, meuLDJT obtains the same result a ground algorithm would obtain. Additionally, the message passing inside of an FO jtree ensures that the current utility value is known at all relevant parclusters and the transfer function preserves the value over time. □

Knowing that meuLDJT is sound, we now look at the completeness of meuLDJT.

9.4.2 Completeness

To show completeness of meuLDJT, we use the completeness of LDJT. By not distinguishing between utility and probability parfactors, i.e., by treating them all the same for the completeness analysis, we can transfer completeness results of LDJT to meuLDJT. Hence, the interface PRVs also account for the PRVs in utility transfer parfactors.

Theorem 9.4.3. *meuLDJT is complete for the same models as LDJT.*

Proof. Inside a time step, meuLDJT first performs a probability message pass, where the utility parfactors are not accounted for. Thus, the completeness of LDJT still holds for this message pass. The utility message pass only distributes the current utility value, but does not eliminate any PRVs. Therefore, the utility message pass does not change anything w.r.t. completeness. As the utility PRV can also be parameterised, we need to account for it while proceeding in time. The interface PRVs account for the PRVs in the utility transfer parfactors. Thus, for temporal messages, which now also includes the utility value, completeness results of LDJT also apply for meuLDJT. Therefore, meuLDJT is complete for the same models as LDJT. □

Thus, the completeness results of LDJT can be transferred to meuLDJT. Now, we take a look at the complexity of meuLDJT.

9.4.3 Complexity

For the complexity of meuLDJT, we investigate the complexity of answering one MEU query with meuLDJT. In general, meuLDJT constructs r_a^{h+1} action sequences, where r_a is the overall range of action PRVs, i.e., number of actions and h the horizon. Thus, in the worst case, meuLDJT would need to go over r_a^{h+1} action sequences, leading to r_a^{h+1} message passes for each of the $h+1$ FO jtrees as meuLDJT computes the message passes of $h+1$ time steps. However, as we mentioned, meuLDJT can reuse computation. By reusing computations, meuLDJT needs to compute $\sum_{i=1}^{h+1} r_a^i$ message passes and expected utility queries. For the current time step, there are only r_a^1 actions possible. For the next time step, there are r_a^2 actions possible because, for each action from the current time step, there are r_a^1 actions possible in the next time step and so on. Therefore, meuLDJT needs to compute $\sum_{i=1}^{h+1} r_a^i$ message passes and expected utility queries. Further, the geometric series $\sum_{i=1}^{h+1} r_a^i$ can be rewritten as $\frac{r_a \cdot (1 - r_a^{h+1})}{1 - r_a} = \frac{r_a \cdot (r_a^{h+1} - 1)}{r_a - 1}$, which is from a complexity perspective bounded by r_a^{h+1}. Hence, from a complexity perspective, meuLDJT only needs to perform r_a^{h+1} message passes on a small FO jtree, while meuLJT needs to perform r_a^{h+1} message passes on an FO jtree with a higher lifted width and more parclusters.

Now, we take a look at the step wise complexity of meuLDJT and then combine the step wise complexities to the overall complexity of meuLDJT. Again, we use the notion from Section 4.3.2. Further, the lifted width now also includes the utility parfactors.

Evidence entering consists of absorbing evidence at each applicable node.

Lemma 9.4.1. *The complexity of absorbing an evidence parfactor is*

$$O(r_a^{h+1} \cdot n_J \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{9.3}$$

For each FO jtree, meuLDJT needs to set the current action as evidence. Thus, to answer one MEU query, meuLDJT needs to set evidence in $\sum_{i=1}^{h+1} r_a^i$ FO jtrees.

Passing messages consists of calculating messages with LJT for every action sequence.

Lemma 9.4.2. *The complexity of passing messages is*

$$O(r_a^{h+1} \cdot n_J \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{9.4}$$

As we mentioned, meuLDJT can reuse some calculations leading to $\sum_{i=1}^{h+1} r_a^i$ instead of $(h+1) \cdot r_a^{h+1}$ message passes.

For each different action in the action sequences, meuLDJT also needs to compute an expected utility query.

Lemma 9.4.3. *The complexity of answering an MEU query is*

$$O(r_a^{h+1} \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{9.5}$$

Answering an expected utility query or a marginal query is the same from a complexity perspective. Overall, for r_a^{h+1} action sequences, meuLDJT has $\sum_{i=1}^{h+1} r_a^i$ different actions, leading to $\sum_{i=1}^{h+1} r_a^i$ expected utility queries. In addition to the expected utility queries, meuLDJT also needs to compute $\sum_{i=1}^{h+1} r_a^i$ α messages, but that only leads to a multiplication by 2 in Eq. (9.5).

We now combine the stepwise complexities to arrive at the complexity of LDJT by adding up the complexities in Eqs. (9.3) to (9.5).

Theorem 9.4.4. *The overall complexity of meuLDJT to answer one MEU query with a horizon of h is*

$$O(r_a^{h+1} \cdot n_J \cdot \log_2 n \cdot r^{w_g} \cdot n_\#^{w_\# \cdot r_\#}). \tag{9.6}$$

9.5 Evaluation

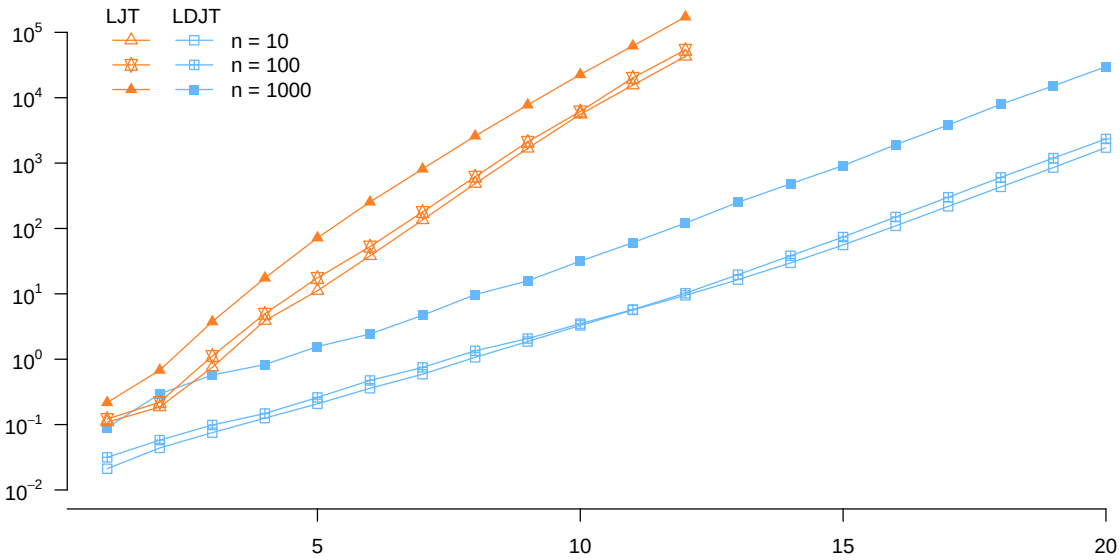


Figure 9.3: Maximum expected utility queries for two possible actions [seconds, log], x-axis: horizon

Let us now evaluate meuLJT and meuLDJT. For the evaluation, we use the example model from this chapter, but again with the PRV *Hot* being parameterised with X. We provide meuLJT with the unrolled model and compare it against meuLDJT while increasing the horizon. We vary the domain size for the logvar X, i.e., $|\mathcal{D}(X)| = 10$, $|\mathcal{D}(X)| = 100$, and $|\mathcal{D}(X)| = 1000$, while setting $|\mathcal{D}(P)| = 3$. Additionally, we evaluate our example model with two actions, i.e., there are two range values for the action PRV, as well as with three actions, i.e., there are three range values for the action PRV.

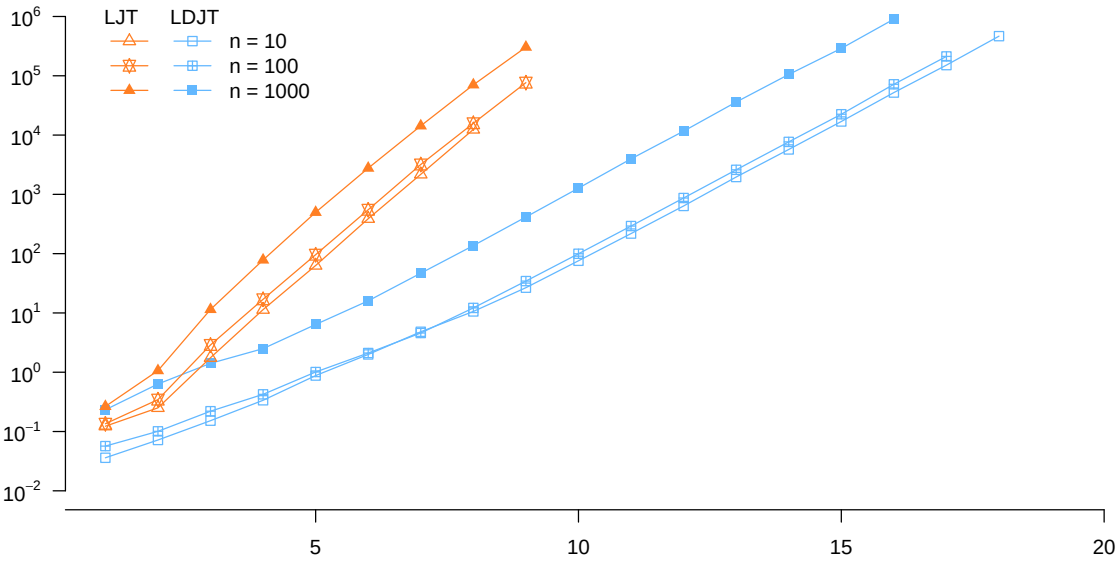


Figure 9.4: Maximum expected utility queries for three possible actions [seconds, log], x-axis: horizon

Figure 9.3 shows the runtimes for two possible action with a horizon up to 20 time steps. For meuLDJT, we can see that increasing the horizon by one doubles the runtime. The behaviour can be explained with the r_a^{h+1} from the complexity results, as r_a is 2. Further, as expected meuLDJT is always faster compared to meuLJT. meuLJT needs to compute the same number of message passes, just on the FO jtree constructed from the unrolled model instead of a small FO jtree encoding a single time step.

Figure 9.4 shows the runtimes with a horizon up to 18 time steps for three actions. Here, we stopped some runs as answering the MEU for a horizon of 16 with $|\mathcal{D}(X)| = 1000$ took roughly 10 days. Nonetheless, we can also see that the runtimes increase by a factor of three if we increase the horizon by one. For this setting, r_a is 3. Hence, the runtimes depicted in Figs. 9.3 and 9.4 can be explained quite well by the results of the complexity analysis. In both cases increasing the domain sizes to 1000 significantly increases the runtimes. However, without lifting calculating an MEU for so many randvars with such a horizon would be infeasible.

9.6 Interim Conclusion

We present PDDecMs, an extension to PDMs, and meuLDJT (Contribution **6a** and **6b**) for sequential probabilistic online decision support by calculating a solution to the lifted temporal MEU problem. By maximising the expected utility, meuLDJT can calculate a best action sequence. We also show that using meuLDJT is more efficient than using

meuLJT with an unrolled model (Contribution **6c**). Further, meuLDJT can efficiently answer a combination of *expected utility*, *hindsight*, *filtering*, and *prediction* queries. Thus, meuLDJT can support decision support and can help to understand the suggested decision by also efficiently answering multiple marginal queries.

We currently check whether meuLDJT can reuse computations from previous expected utility calculations by, e.g., identifying dominant actions for belief state regions.

Part III

Extending Evidence Handling

Chapter 10

Uncertain Evidence

In this part of the dissertation, we focus on evidence in temporal probabilistic relational models. We begin by investigating uncertain evidence, i.e., observations that are not either true or false, but with a probability p true and with $1 - p$ false. Additionally, we would like to investigate that evidence can slowly ground a temporal probabilistic relational model. To benefit from lifted calculations, we propose a method to approximate symmetries to restore a lifted representation after evidence has slowly grounded a model.

So far lifted inference approaches have only dealt with *certain* evidence, i.e., evidence parfactors where exactly one range value is mapped to 1 and all other range values are mapped to 0. In this chapter, we investigate *uncertain* evidence, i.e., evidence that is assigned more than one range value with a potential greater than 0.

For Bayesian networks, work on uncertain evidence exists, sometimes called soft evidence in contrast to certain, i.e., hard evidence (Chan and Darwiche, 2005; Pearl, 2001, 2014; Jeffrey, 1990; Peng *et al.*, 2010). To the best of our knowledge, lifted inference algorithms only handle certain evidence. In this chapter, we interpret uncertain evidence in the sense of a priori distributions, which is closely related to Pearl's method of virtual evidence (Pearl, 2014). This chapter includes two main contributions, (i) an algorithm, LVE^{evi}, for handling uncertain evidence w.r.t. probabilistic relational models and (ii) LJT^{evi}, for handling uncertain evidence for multiple queries. Additionally, we show soundness and completeness results for LVE^{evi} and LJT^{evi} and a brief empirical case study.

The remainder of this chapter is structured as follows: First, we recapitulate how LVE handles certain evidence and then present LVE^{evi} to handle uncertain evidence. Second, we recapitulate how LJT handles certain evidence and then present LJT^{evi} to handle uncertain evidence. Then, we present an empirical case study and show that from a runtime perspective certain and uncertain evidence roughly put the same overhead on the execution time. Last, we conclude with upcoming work.

This chapter is based on the following publication:

> Marcel Gehrke, Tanya Braun, and Ralf Möller. Uncertain Evidence for Probabilistic Relational Models. In *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*, pages 80–93. Springer, 2019

10.1 LVE for Uncertain Evidence

Currently, evidence in LVE, and therefore, LJT as well as LDJT, is always certain. However, sensors often are not completely reliable or some test may be more precisely performed in a hospital compared to a test in a general practice (Steinhäuser and Kühlein, 2015). Before we incorporate uncertain evidence in LVE, we take a closer look at how LVE handles certain evidence.

10.1.1 Evidence in LVE

Evidence displays symmetries if observing the same value for n instances of a PRV (Taghipour *et al.*, 2013c). In a parfactor $g^E = \phi^E(R(\mathbf{X}))_{|C^E}$, a potential function ϕ^E and constraint C^E encode the observed values and instances for PRV $R(\mathbf{X})$.

Example 10.1.1 (Entering evidence)**.** *Assume we observe the value true for ten randvars of the PRV Att(X). The corresponding parfactor is* $\phi^E(Att(X))_{|C^E}$*.* C^E *represents the domain of X restricted to the 10 instances and* $\phi^E(true) = 1$ *and* $\phi^E(false) = 0$*.*

A technical remark: To *absorb* evidence, we split all parfactors g^i that cover $R(X)$, called shattering (Braz *et al.*, 2005), restricting C^i to those tuples that contain $gr(R(X)_{|C^E})$ and a duplicate of g^i to the rest. g^i absorbs g^E.

To understand evidence entering, let us have a look at lifted absorption, which is outlined in Alg. 11, without including CRVs for ease of explanation. The operator uses a count function defined as follows.

Definition 10.1.1 (Count function)**.** Given a constraint $C = (\mathcal{X}, C_{\mathbf{X}})$, for any $\mathbf{Y} \subseteq \mathbf{X}$ and $\mathbf{Z} \subseteq \mathbf{X} \setminus \mathbf{Y}$, the function $\text{COUNT}_{\mathbf{Y}|\mathbf{Z}} : C_{\mathbf{X}} \to \mathbb{N}$ is defined by

$$\text{COUNT}_{\mathbf{Y}|\mathbf{Z}}(t) = |\pi_{\mathbf{Y}}(C_{\mathbf{X}} \bowtie_{\mathbf{Z}} \pi_{\mathbf{Z}}(t))|.$$

i.e., for a tuple $t \in C_{\mathbf{X}}$, it outputs how many constants for $\mathbf{Y}$ co-occur with the value of $\mathbf{Z}$ in t. We define $\text{COUNT}_{\mathbf{Y}|\mathbf{Z}}(t) = 1$ for $\mathbf{Y} = \emptyset$. $\mathbf{Y}$ is *count-normalised* w.r.t. $\mathbf{Z}$ in C iff

$$\exists n \in \mathbb{N} : \forall t \in C_{\mathbf{X}} : \text{COUNT}_{\mathbf{Y}|\mathbf{Z}}(t) = n.$$

If n exists, we call it the conditional count of $\mathbf{Y}$ given $\mathbf{Z}$ in C, denoted by $\text{COUNT}_{\mathbf{Y}|\mathbf{Z}}(C)$.

Before we take a closer look at the operator, we illustrate the count function.

Example 10.1.2 (Count)**.** *Consider the constraint* $C = ((X, J), \{(eve, springer), (alice, aaai\_press), (alice, springer), (bob, aaai\_press), (bob, springer)\})$*. With* $\mathbf{X} = \{X, J\}$*,* $\mathbf{Y} = \{J\}$*, and* $\mathbf{Z} = \{X\}$*, the count function calculates the following for tuple* $(eve, springer)$*: First, it projects* $(eve, springer)$ *onto* $\{X\}$*, which leaves* (eve)*. Then, it joins eve with the tuples from C, i.e.,* $(eve, springer)$*, and projects the tuples onto* $\{J\}$*,*

Algorithm 11 Lifted Absorption (Taghipour *et al.*, 2013c).

Operator ABSORB

Inputs:

(1) $g = \phi(\mathcal{A})_{|C}$: a parfactor in G
(2) $A^i \in \mathcal{A}$ with $A^i = R(\mathbf{X})$
(3) $g^E = \phi^E(R(\mathbf{X}))_{|C^E}$: an evidence parfactor
Let $\mathbf{X}^{excl} = \mathbf{X} \setminus lv(\mathcal{A} \setminus A^i)$;
$L' = lv(\mathcal{A}) \setminus \mathbf{X}^{excl}$;
o = the observed value for $R(\mathbf{X})$ in g^E

Preconditions:

(1) $gr(A^i{}_{|C^i}) \subseteq gr(A^i{}_{|C^E})$
(2) $\mathbf{X}^{excl}$ is count-normalised w.r.t. L' in C.

Output: $\phi'(\mathcal{A}')_{|C'}$, with

(1) $\mathcal{A}' = \mathcal{A} \setminus A^i$
(2) $C' = \pi_{lv(C)\setminus\mathbf{X}^{excl}}(C)$
(3) $\phi'(\dots, a^{i-1}, a^{i+1}, \dots) = \phi(\dots, a^{i-1}, o, a^{i+1}, \dots)^r$ with $r = \text{COUNT}_{\mathbf{X}^{excl}|L'}(C)$

Postcondition: $G \cup \{g^E\} \equiv G \setminus \{g\} \cup \{g^E, \text{ABSORB}(g, A^i, g^E)\}$

which results in a set with one element, $(springer)$*. Last, it outputs the cardinality of the set, here* 1*. For* $(alice, aaai\_press)$*, the first projection yields* $(alice)$*, with the join resulting in* $(alice, aaai\_press)$ *and* $(alice, springer)$ *and the second projection resulting in* $(aaai\_press)$ *and* $(springer)$*, yielding a cardinality of* 2*. Thus, there does not exist a unique* n *for all tuples in* C*, that is,* J *is not count-normalised w.r.t.* X *in* C*.*

Now, consider the constraint $C' = ((X, J), \{(alice, aaai\_press), (alice, springer), (bob, aaai\_press), (bob, springer)\})$*. Here, each tuple leads to a count of* 2 *given* $\mathbf{X} = \{X, J\}$*,* $\mathbf{Y} = \{J\}$*, and* $\mathbf{Z} = \{X\}$ *and thus,* J *is count-normalised w.r.t.* X *in* C'*. The conditional count of* J *given* X *in* C' *is* 2*. In case, alice and bob would additionally publish another journal, the count would be* 3*. The count in this case is important as absorbing evidence eliminates as many instances as the count function yields, and thus, LVE needs to exponentiate the result with the count.*

ABSORB has as inputs an evidence parfactor g^E with evidence for a PRV A^i and a parfactor g, which contains A^i. As a precondition, A^i covers at most the randvars of g^E in g. Thus, LVE often performs a shattering before absorption to split parfactors into parts with and without evidence. The other precondition relates to logvars being eliminated during absorption. For the output parfactor, the operator deletes A^i from g, reducing the dimensions in g. The operator also projects the constraint C of g onto the remaining logvars. Lastly, it collects all potentials that agree with the evidence, i.e., where $A^i = o$, and exponentiates them accordingly. As the operator performs a

dimension reduction by deleting A^i from the argument sequence, rather than keeping the argument and setting all potentials where $A^i \neq o$ to 0, LVE has to apply the absorption operator to each parfactor that contains A^i.

Example 10.1.3 (Absorb operation)**.** *To illustrate the* ABSORB *operator, assume that eve attends a conference, i.e., $Att(eve) = true$. LVE builds an evidence parfactor $g^E = \phi^E(Att(X))_{|C^E}$, with $C^E = (X, \{eve\})$. As g^0 and g^1 contain $Att(X)$, both need to absorb g^E. To absorb g^E in g^0, LVE first splits g^0 into $g^{0'}$ for eve and $g^{0''}$ for all other instances, i.e., alice and bob. With g^E and $g^{0'}$ as inputs, the first precondition holds as both g^E and $g^{0'}$ have X restricted to eve. Since $\mathbf{X}^{excl} = X \setminus lv(Hot, Pub(X, J)) = \emptyset$, i.e., no logvars are eliminated, $\mathbf{X}^{excl}$ is count-normalised and $r = 1$. Hence, the operator can proceed. It removes $Att(X)$ from $g^{0'}$. The constraint remains unchanged. Lastly, all potentials that agree with $Att(eve) = true$ remain and are exponentiated to the power of 1. Similarly, g^E gets absorbed in g^1.*

One could also perform lifted absorption by multiplying g^E into g, which leads to potentials of 0 whenever $A^i \neq o$. Afterwards, one could drop the mappings with potentials of 0 and then eliminate A^i from the argument sequence as after dropping the mappings, $A^i = o$ in all remaining mappings, holding no further information. However, absorption as in Alg. 11 only works for certain evidence.

10.1.2 Uncertain Evidence in LVE^{evi}

The main differences to certain evidence and its handling are to be found in in specifying evidence, constructing evidence parfactors, and handling evidence parfactors within LVE. Currently, one event has a potential of 1, while all others have a potential of 0 in an evidence parfactor. With uncertain evidence, we need to be able to specify potentials different from 0 and 1 for possible events of a PRV. However, evidence should not incur a scaling factor. Therefore, individual events of a PRV A have assigned a potential p with $p \in [0, 1]$ and the potentials of all possible events of A add up to 1. We allow for two options to specify potentials for events. The first option is to specify the potential for each possible event of a PRV A^i with the sum of the potentials being 1. LVE^{evi} then constructs an evidence parfactor $g^E = \phi^E(A^i)_{|C^E}$ accordingly. The second option is to specify a subset of the events with the sum of the potentials s being at most 1. LVE^{evi} constructs an evidence parfactor $g^E = \phi^E(A^i)_{|C^E}$, distributing the residual potential $1 - s$ on the remaining range values in a max-entropy style (Thimm and Kern-Isberner, 2012). Constructing evidence parfactors in such a way ensures that all range values have a potential and that the potentials add up to 1, and thereby, ensuring the requirements of uncertain evidence parfactors.

Example 10.1.4 (Uncertain evidence)**.** *Assume the potential of eve attending a conference is* 0.9*. We may specify the evidence using a complete distribution, $Att(eve) =$*

Algorithm 12 Evidence Handling in LVE^{evi}

```
1: procedure addEvidence(G, g^E)
2:     if g^E is uncertain then
3:         Add g^E to G
4:     else
5:         Absorb g^E in G
```

$((true, 0.9), (false, 0.1))$. *The other option is to only specify* $Att(eve) = (true, 0.9)$*, a subset of the distribution. Then, LVE*^{evi} *would distribute the remaining* 0.1 *max-entropy alike on the remaining range values, while constructing the evidence parfactor. With* $Att(X)$ *being boolean, there is only one other range value, namely* $false$*, which would be assigned a potential of* 0.1*. In case of another range value, e.g.,* $workshops\_only$*, then both would be assigned a potential of* 0.05*. Assigning a distribution to evidence still allows for specifying certain evidence. Given* $Att(eve) = ((true, 1))$*, all other range values would be assigned the potential* 0*, which is identical to the evidence so far in LVE.*

We now present LVE^{evi} to handle uncertain evidence while answering a query. The workflow of LVE^{evi} is identical to LVE. Instead of absorbing all evidence $\mathbf{E}$ in affected parfactors, a case distinction occurs, which is specified in Alg. 12, for each evidence parfactor g^E constructed for $\mathbf{E}$. If g^E contains certain evidence, g^E is absorbed in G as before. If g^E is uncertain evidence, g^E is added to G. During query answering, the uncertain evidence is then properly accounted for since g^E is multiplied into the model at one point and therefore, influences a queried distribution accordingly. Next, we discuss theoretical results of LVE^{evi}.

10.1.3 Theoretical Analysis

In this section, we investigate soundness and completeness of LVE^{evi}. We do not consider complexity aspects as uncertain evidence leads to an additional multiplication of evidence parfactors instead of absorptions. Thus, from a complexity perspective, there hardly is any difference. First, let us have a look at the soundness of LVE^{evi}.

Theorem 10.1.1. *LVE*^{evi} *is sound, i.e., computes a correct result for a query* $\mathcal{Q}$ *given an input model* G *and evidence* $\mathbf{E}$*.*

Proof. Since both LVE^{evi} and LVE handle certain evidence in the same way and LVE is sound (Taghipour *et al.*, 2013c), LVE^{evi} is sound w.r.t. certain evidence. We interpret uncertain evidence as an a priori distribution for events. LVE^{evi} simply adds evidence parfactors of uncertain evidence to a model. During query answering, LVE^{evi} then handles these parfactors as part of the model, multiplying evidence parfactors into other

parfactors accordingly, thus, accounting for evidence as a form of an a priori distribution. LVE is sound (Taghipour *et al.*, 2013c) w.r.t. its operation and therefore, also multiplication, thus, LVE^{evi} is sound for uncertain evidence. □

Second, let us have a look at the completeness of LVE^{evi}.

Theorem 10.1.2. *LVE^{evi} is complete for unary evidence, i.e., the runtime complexity is polynomial in the domain sizes of the model logvars.*

Proof. Given certain, unary evidence, i.e., evidence which can be represented in a parfactor with an evidence PRV using one-logvar, LVE is complete (Van den Broeck and Davis, 2012; Taghipour *et al.*, 2013d). Replacing certain, unary evidence with uncertain, unary evidence with a given distribution leads to the same number of splits during shattering and the number of splits is linear per evidence PRV and model parfactor (Taghipour *et al.*, 2013c). Thus, LVE^{evi} still has a time complexity polynomial in the domain sizes of the model logvars given uncertain, unary evidence and the completeness results for unary evidence from LVE also hold for LVE^{evi}. □

Let us now have a look at the implication of uncertain evidence for LJT.

10.2 LJT for Uncertain Evidence

LVE^{evi} handles uncertain evidence efficiently for single queries. To handle multiple queries efficiently, we incorporate uncertain evidence into LJT based on the same principles that have guided the adaptation of LVE to handle uncertain evidence. Before we present LJT^{evi}, we first take a closer look at how LJT handles evidence.

10.2.1 Evidence in LJT

Evidence handling in LJT generally works by performing the following steps: (i) Construct evidence parfactors. (ii) Enter evidence parfactors into FO jtree. (iii) Shatter local models on entered evidence parfactors. (iv) Absorb evidence parfactor in local models. Basically, LJT handles evidence in each local model as LVE does in its input model. In each parcluster that covers an evidence PRV, LJT tests each parfactor for evidence absorption. If a parfactor in a local model covers the evidence PRV, LJT shatters the parfactor on the evidence and lets the affected parfactor absorb the evidence parfactor. Whenever a separator no longer covers an evidence PRV, LJT can omit checking the subtree beyound the neighbour associated with the separator based on the running intersection property.

Example 10.2.1 (Evidence entering)**.** *Again, assume that eve attends a conference as certain evidence with an evidence parfactor $g^E = \phi^E(Att(X))_{|C^E}$, with $C^E = (X, \{eve\})$.*

Then, LJT enters g^E in J^{ex}. The PRV $Att(X)$ occurs in $\mathbf{C}^1$ and $\mathbf{C}^2$. LJT shatters the local models G^1 and G^2, i.e., g^0 and g^1. LJT splits g^0 into $g^{0'}$ for eve and $g^{0''}$ for all other instances, in this case, alice and bob. Analogously, LJT splits g^1 into $g^{1'}$ and $g^{1''}$. Finally, LJT absorbs g^E in $g^{0'}$ and $g^{1'}$. After the absorption, all local models encode information about certain evidence and the overall model.

Knowing how LJT handles certain evidence, we present LJT^{evi} to efficiently handle uncertain evidence for multiple queries.

10.2.2 Uncertain Evidence in LJT

LJT^{evi} is based on LJT and is able to handle uncertain evidence as well. Evidence may be specified in the same manner as for LVE^{evi}, which allows for certain evidence as well as uncertain evidence, partially or fully specified with distributions, whose potentials add up to 1. LJT^{evi} has the same workflow as LJT. Algorithm 13 describes the steps to enter an evidence parfactor g^E in an FO jtree J. Again, a case distinction occurs. If g^E encodes certain evidence, LJT^{evi} works as LJT, absorbing g^E in the affected parfactors of all parclusters that cover the evidence PRV. If g^E encodes uncertain evidence, LJT^{evi} adds g^E to one local model of a parcluster that covers the evidence PRV. During message passing, the information about the evidence is distributed to all other parclusters, which makes it apparent why uncertain evidence should only be added to one local model. In case LJT^{evi} would add the uncertain evidence parfactor to all parclusters containing the evidence PRV, then the evidence would be distributed during message passing and accounted for multiple times. One could directly shatter a local model of the chosen parcluster and multiply g^E into it. But, the operations are optional: LJT^{evi} uses LVE for its calculations, which is able to handle g^E accordingly and multiply g^E into other parfactors when necessary, resulting in more efficient multiplications.

Example 10.2.2 (Uncertain evidence with LJT^{evi})**.** *Assume that eve is attending a conference with a potential of* 0.9*. So, LJT^{evi} builds an evidence parfactor g^E =*

Algorithm 13 Evidence Handling in LJT^{evi}

```
1: procedure enterEvidence(J, g^E)
2:     if g^E is uncertain then
3:         Add g^E to the local model of one parcluster, which contains the PRV of g^E
4:         Shatter local model (optional)
5:         Multiply g^E into local model (optional)
6:     else
7:         Enter g^E in all parclusters, which contain the PRV of g^E
8:         Shatter local models
9:         Absorb g^E
```

$\phi_E(Att(X))_{|C^E} = ((true, 0.9)), (false, 0.1)$, *with* $C^E = (X, \{eve\})$, *as would* LVE^{evi}. *Now,* LJT^{evi} *only needs to find one parcluster containing* $Att(X)$, *instead of all parclusters containing* $Att(X)$. *Both parclusters* $\mathbf{C}^1$ *and* $\mathbf{C}^2$ *from Fig. 2.2 contain* $Att(X)$. LJT^{evi} *randomly chooses to add* g^E *to* $\mathbf{C}^2$. *As the remaining part is optional, we opt against it for efficiency reasons. Evidence entering now is complete.*

During message passing, LJT^{evi} *sends* m^{21} *from* $\mathbf{C}^2$ *to* $\mathbf{C}^1$. *To calculate* m^{21}, LJT^{evi} *splits* g^1 *into* $g^{1'}$ *for eve and* $g^{1''}$ *for all other instances. Then,* LJT^{evi} *eliminates* $DoR(X)$ *from* $g^{1'}$ *and* $g^{1''}$. *Afterwards,* LJT^{evi} *sends* m^{21}, *which contains* g^E, $g^{1'}$, *and* $g^{1''}$, *to* $\mathbf{C}^1$. *In* m^{21}, *we can easily see that* LJT^{evi} *propagates evidence to all parclusters containing the PRV of the evidence parfactor as it is an explicit part of the message.*

Knowing how LJT^{evi} handles uncertain evidence, let us discuss theoretical implications of LJT^{evi} compared to LJT.

10.2.3 Theoretical Analysis

This section investigates soundness and completeness of LJT^{evi}. Similar to LVE^{evi} it also holds for LJT^{evi} that the complexity remains roughly the same as for LJT, as the main difference between LJT^{evi} and LJT is to perform a multiplication of an evidence parfactor compared to absorption. First, let us have a look at the soundness of LJT^{evi}.

Theorem 10.2.1. LJT^{evi} *is sound, i.e., computes a correct result for a query* $\mathcal{Q}$ *given an input model* G *and evidence* $\mathbf{E}$.

Proof. For certain evidence, LJT^{evi} computes the same result as LJT since they perform the same steps. Given that LJT is sound (Braun, 2020), LJT^{evi} is sound. For uncertain evidence, LJT^{evi} adds evidence parfactors once to a local model of one parcluster. During message passing and query answering, LJT^{evi} then properly accounts for the evidence as an a priori distribution for the given events. For the message passing and multiplication of the uncertain evidence with other parfactors, LJT^{evi} uses LVE. As LVE is sound, LJT^{evi} is in turn also sound for uncertain evidence. □

Second, let us have a look at the completeness of LJT^{evi}.

Theorem 10.2.2. LJT^{evi} *is complete for unary evidence, i.e., the runtime complexity is polynomial in the domain sizes of the model logvars.*

Proof. The completeness results for unary evidence and LVE (Van den Broeck and Davis, 2012; Taghipour *et al.*, 2013d) extend also to LJT. Following the same argument as in the proof of completeness for LVE^{evi}, Section 10.1.3, the change from certain to uncertain evidence over one distribution does not lead to groundings, which means that the runtime complexity is still polynomial in the domain sizes of the model logvars and the completeness results extend to LJT^{evi}. □

For uncertain evidence, we do not need to handle anything special in the temporal case. Thus, to also have uncertain evidence for temporal models, LDJT, or more precisely LDJT^{evi} can simply use LJT^{evi} as a subroutine to handle uncertain evidence efficiently. Finally, we have a look at runtimes of LVE^{evi} and LJT^{evi} compared to LVE and LJT.

10.3 Empirical Case Study

We have implemented a prototype version of LJT^{evi} and adapted an LVE implementation by Taghipour (`https://dtai.cs.kuleuven.be/software/lve`) for uncertain evidence. Given the changes from certain to uncertain events in LVE and LJT and their effects on completeness, we expect implementations of the algorithms to accomplish similar runtimes for certain and uncertain evidence given that certain evidence does not cancel out a majority of the model. If certain evidence exists for a majority of the PRVs in a model, the dimension reduction during absorption leaves a very small model, enabling fast query answering. Thus, we use the running example with a domain size of 1000 and add certain evidence $Att(X) = ((true, 1))$ as well as uncertain evidence $Att(X) = ((true, 0.8), (false, 0.2))$, covering 0% to 100% of $gr(Att(X))$ in 10% steps. The query term is $DoR(x_{1000})$. We look at two aspects, (i) runtimes for answering a single query with LVE^{evi} and LJT^{evi} and (ii) runtimes of the LJT^{evi} steps.

Figure 10.1 shows runtimes in milliseconds [ms] for answering a single query with LVE^{evi} (triangles) and LJT^{evi} (circles) with evidence coverage ranging from 0% to 100% on the x-axis. The filled symbols show runtimes for certain evidence. The hollow symbols show runtimes for uncertain evidence. As expected, LJT runtimes are lower than LVE runtimes since LJT is able to use a smaller submodel compared to the original input model. For LJT^{evi}, certain evidence leads to lower runtimes than uncertain evidence due to the dimension reduction as well as its preprocessing. Evidence is already handled when LJT^{evi} starts answering the query. And as the submodel for query answering is rather small, the dimension reduction has a comparatively large impact. For LVE, certain

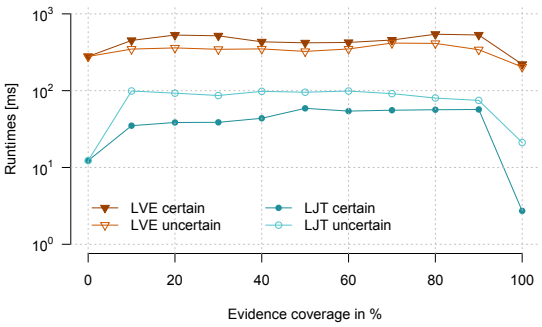


Figure 10.1: Runtimes for query answering

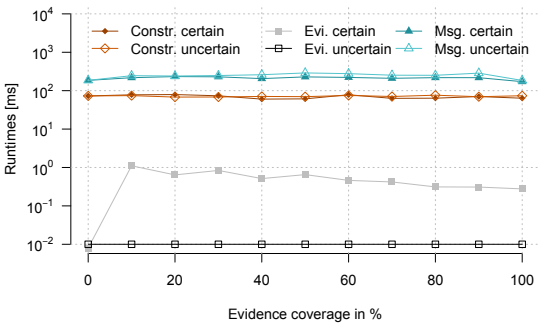


Figure 10.2: Runtimes for LJT steps

evidence leads to larger runtimes as the overall impact of the dimension reduction is not as large and absorption in itself is a rather expensive operation, even though is leads to faster runtimes afterwards. The increase in runtimes from 0% to 10% evidence as well as the decrease in runtimes from 90% to 100%, which occurs for both certain and uncertain evidence, comes from shattering on evidence. With 0% and 100% evidence, no splits are necessary, which means smaller models in terms of the number of parfactors to handle.

Figure 10.2 shows runtimes in milliseconds [ms] of the steps construction (diamond), evidence entering (squares), and message passing (triangles) of LVE^{evi} with evidence coverage ranging from 0% to 100% on the x-axis (filled = certain, hollow = uncertain). Evidence has no influence on construction. Therefore, runtimes are nearly the same for certain and uncertain evidence. Certain evidence leads to larger runtimes as LJT^{evi} absorbs the evidence during this step. Uncertain evidence is simply added to a local model and thus, entering uncertain evidence does not depend on evidence coverage. Message passing with uncertain evidence takes slightly longer than with certain evidence as the dimension reduction also helps during message calculation.

Overall, the case study shows that uncertain evidence leads to similar runtimes for LVE^{evi} and LJT^{evi} compared to certain evidence with a limited scope. Comparing runtimes for domain sizes of 10 to domain sizes of 1000 shows that even though the domain sizes rise by a factor of 100, runtimes only rise by a factor of 2.7 to 8.6 for uncertain evidence and LJT^{evi}. As uncertain evidence basically leads to an additional parfactor and a limited number of splits, we expect empirical results from Chapter 5 as well as from (Braun and Möller, 2018c; Braun, 2020) to also hold for LVE^{evi} and LJT^{evi}, with both algorithms outperforming the ground case.

10.4 Interim Conclusion

We present LVE^{evi} (Contribution **7a**) and LJT^{evi} (Contribution **7b**), i.e., versions of LVE and LJT, which incorporate uncertain evidence and allow for similar runtimes as before. We specify how to construct and handle uncertain evidence. LVE^{evi} and LJT^{evi} close the gap to temporal probabilistic databases (TPDBs) to also allow for uncertain evidence in probabilistic relational models. Further, uncertain evidence does not influence completeness results of LVE, LJT, and LDJT (Contribution **7c**).

Next, we have a look into implications of evidence in lifted solutions for temporal probabilistic relational models. Evidence can slowly ground a model over time and thereby, strip inference off the benefits of lifting. To benefit from lifted computation, we investigate how we can use approximate symmetries to restore a lifted representation over time and therefore, use lifted operations .

Chapter 11

Taming Reasoning in Temporal Probabilistic Relational Models

A key challenge for performing efficient inference in temporal probabilistic relational models is that evidence can slowly ground a model over time. Thus, in this chapter, we investigate a method to use approximate symmetries to restore a lifted, i.e., non-grounded, representation while proceeding in time. In general, reasoning in lifted representations has a complexity polynomial in domain sizes. But, models dissolve into ground instances by evidence, which no longer permits reasoning in polynomial time, making query answering infeasible for any reasoning algorithm, exact or approximate. Thus, a key challenge during inference in temporal models is to restore a lifted representation. Therefore, we formulate and study the problem of keeping reasoning polynomial (KRP) in temporal models to tame the effect of evidence for efficient query answering.

We begin by giving an intuition of how evidence can slowly ground a model. In a temporal probabilistic relational model, we would need completely symmetric evidence for a group to calculate a lifted solution for that group. Symmetric evidence in this case means that we observe the very same event for all instances of that group within one time step, i.e., with a boolean PRV, we either observe true, false, or nothing for all instances of this group within one time step. However, in case we observe non-symmetric evidence for a group of instance that group will slowly ground. With a boolean PRV, we can have 3 groups after one time step, one group with the event being true, one group with the event being false, and one group without evidence. After two time steps, we can already have 9 as the 3 groups from before can each be split into 3 groups again and so on.

Example 11.0.1 (Symmetric and non-symmetric evidence)**.** *Assume that* $|\mathcal{D}(X)| = 100$ *and that we observe evidence for* $DoR(X)$*. Symmetric evidence would be that we observe* $DoR(X') = true$ *for* $\mathcal{D}(X') = \{x_1, ..., x_{50}\}$ *as well as* $DoR(X'') = false$ *for* $\mathcal{D}(X'') = \{x_{51}, ..., x_{100}\}$ *for the first time step. For the next time step, we may observe* $DoR(X') = false$ *for* $\mathcal{D}(X') = \{x_1, ..., x_{50}\}$ *as well as* $DoR(X'') = false$ *for* $\mathcal{D}(X'') = \{x_{51}, ..., x_{100}\}$ *and, in the time step after that, we could possibly observe* $DoR(X') = false$ *for* $\mathcal{D}(X') = \{x_1, ..., x_{50}\}$ *as well as* $DoR(X'') = true$ *for* $\mathcal{D}(X'') = \{x_{51}, ..., x_{100}\}$*. Thus, within one time step we observe the same event for all instances within one symmetry group, but the events can change over time.*

For non-symmetric evidence we could observe $DoR(X') = true$ *for* $\mathcal{D}(X') = \{x_1, ..., x_{30}\}$, $DoR(X'') = false$ *for* $\mathcal{D}(X') = \{x_{31}, ..., x_{60}\}$, *and nothing for* $\mathcal{D}(X) = \{x_{61}, ..., x_{100}\}$. *Just focusing on the first group: In the next time step, we may observe* $DoR(X') = true$ *for* $\mathcal{D}(X') = \{x_1, ..., x_{10}\}$, $DoR(X'') = false$ *for* $\mathcal{D}(X'') = \{x_{11}, ..., x_{20}\}$, *and nothing for* $\mathcal{D}(X) = \{x_{21}, ..., x_{30}\}$ *(and the same for the other two initial groups). Thereby, after only two time steps, the* 100 *instance would already be split into* 9 *groups. Following the scheme, after* 3 *time steps, we could already have* 3^3 *groups. After* 4 *time steps, we could have* 3^4 *groups. Finally, after* 5 *time steps, everything could be grounded with each instance having its own group because every time one instance of a group behaves differently as we observe a different event for that instance, the instance is split from that group. Thus, after only five time steps, each of the* 100 *instances could already have a unique evidence sequence leading to a completely grounded model.*

To the best of our knowledge, none of the existing inference approaches for temporal probabilistic relational models tackle the KRP problem. Ahmadi *et al.* (2013) propose a colour passing scheme to obtain a lifted representation of a DMLN using exact symmetries. The colour passing scheme is highly efficient to find exact symmetries in a model. One could also apply the colour passing scheme to find exact symmetries while transitioning from one time step to the next. However, to solve the KRP problem, one needs to find approximate symmetries efficiently. Thus, the KRP problem is still open.

For static relational models, approaches exist to approximate symmetries as evidence may ground even a static model (Van den Broeck and Davis, 2012). Van den Broeck and Darwiche (2013) approximate lifted binary evidence. Singla *et al.* (2014) propose approximate lifting techniques, which group together distinguishable objects and treat them identically. Venugopal and Gogate (2014) form clusters of objects and project the marginal distribution of one object to all objects of a cluster. Both approaches introduce an unknown bias into the distributions of the groups. Van den Broeck and Niepert (2015) present an unbiased approach for approximating symmetries. However, these approaches do not account for temporal aspects.

Thus, we present temporal approximate merging (TAMe) as an approach to solve the KRP problem in temporal models. Specifically, TAMe incorporates (i) clustering to group submodels and (ii) statistical significance checks to test the groups to be merged. Model structure and behaviour are captured in a set of functions that define local distributions for the randvars in the model. Clustering forms groups of functions based on the similarity between local distributions. The significance checks allow for determining the fitness of the clustering. If the clustering is deemed fit, each group is merged, yielding an unbiased approximation. In exchange for faster runtime, TAMe introduces a bounded error, which becomes negligible over time.

Boyen and Koller (1998) show that for stationary processes, evidence can lead to conditional dependences in temporal probabilistic propositional models, making inference infeasible. They propose to introduce additional randvars to achieve conditional inde-

pendences between subprocesses even under evidence. Further, Boyen and Koller show that, for any approximation scheme of belief state representations, the error decreases exponentially as the process evolves, making the introduced error bounded indefinitely (Boyen and Koller, 1998). Their approach and TAMe are related as in both cases evidence can make inference infeasible. However, TAMe aims at *automatically* restoring a lifted representation. In summary, the cause, namely evidence, is the same for both problems but the means to make inference feasible again differ highly. Nonetheless, we show that we can build upon their theoretical result to prove that an approximation error of TAMe is bounded indefinitely.

The temporal behaviour of a model is the reason why approximation errors are indefinitely bounded in stationary processes and why distributions converge over time. While transitioning from one time step to the next, each time the inference engine multiplies the model behaviour on the distributions or potentials of (P)RVs, i.e., the parfactors of the copy pattern are multiplied on the current state of the system for each time step. Hence, in case we observe only slightly different evidence for instances, the model behaviour becomes such a big factor that the distributions of the (P)RVs converge.

Example 11.0.2 (Model behaviour)**.** *Assume that we observe for one instance a different event for each time step. So in the first time step, we observe for* x_1 *a different event than for the rest, leading to splitting* x_1 *from the rest. In the next time step, we could observe for* x_2 *a different event than for the rest, leading to splitting* x_2 *from the rest. Thus, after* 100 *time steps, we would have a completely grounded model as each instance has its own unique evidence sequence. However, in this example we observe in each time step the same event for* 99 *instances as only one instance behaves differently. Thus, the belief state of each instance will be most likely about the same after the* 100 *time steps as the temporal model behaviour becomes a rather large factor. The model behaviour gets multiplied on the potentials for each time step and therefore, will be a huge factor in comparison to one different event over* 100 *time steps.*

TAMe is applicable to different formalisms and algorithms. However, we discuss TAMe as part of LDJT for two reasons: First, when advancing in time, LDJT computes a minimal message that is the source of the most splits of the next time step. Applying TAMe on this message tackles the KRP problem at its root. Second, using TAMe with an exact algorithm allows for attributing errors to merging rather than imprecisions during reasoning. Additionally, TAMe is deterministic in its approximation, thereby, avoiding problems with sampling rates or ergodicity. Empirical results show that TAMe significantly improves the performance of LDJT in general, while keeping errors small and attributable to merging. Further, results show that the significance checks prevent unnecessary errors.

In the following, we slightly adapt the running example. Then, we present TAMe, which includes clustering, significance checks, and merging. Lastly, we evaluate TAMe theoretically and empirically.

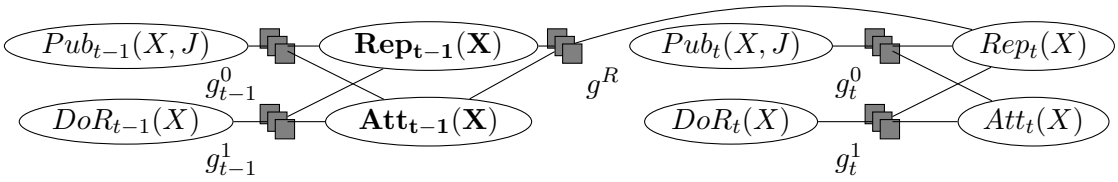


Figure 11.1: $G^{ex}_{\rightarrow}$ to illustrate TAMes

This chapter is based on the following paper:

> Marcel Gehrke, Ralf Möller, and Tanya Braun. Taming Reasoning in Temporal Probabilistic Relational Models. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, pages 2592–2599, 2020

11.1 Preliminaries

For this chapter, we slightly change our running example. In our example so far, the *inter*-slice parfactor contains Hot_t. Thus, a PRV from time slice t in the *inter*-slice parfactor has no logvars. Hence, no splits of logvars are carried over from one time slice to the next as all logvar are eliminated when Hot_{t-1} and $Att_{t-1}(X)$ are eliminated from g^H. However, in this chapter, we investigate how to undo splits that carry over from one time step to the next. Figure 11.1 shows $G^{ex}_{\rightarrow}$ for this chapter. The main difference is that we have replaced the PRV Hot with a PRV $Rep(X)$ as well as g^H with g^R. Thereby, the splits of X carry over from one time step to the next, as, X is not eliminated from one time step to the next in this model.

The intuition behind the changed example is that we are interested in the reputation of researches rather than whether a particular topic is hot. For example, we can observe AAAI conference attendance, which changes over time as, unfortunately, getting papers accepted at consecutive conferences is difficult. Nonetheless, people with high attendance usually have a good reputation.

Before we present TAMe, we recapitulate how LDJT proceeds in time. The FO jtrees in LDJT contain a minimal set of PRVs to m-separate time steps, which means that information about these PRVs renders FO jtrees independent from each other. Querying

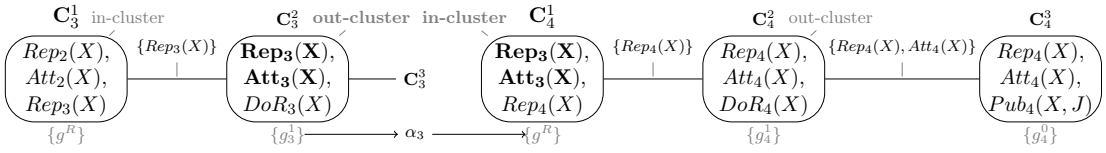


Figure 11.2: FO jtree J_3 without $\mathbf{C}^3_3$ and FO jtree J_4 to illustrate TAMe

a minimal set of PRVs with LVE in an FO jtree combines all information to m-separate time steps. To obtain the minimal set, LDJT uses interface PRVs $\mathbf{I}_t$ of $G_{\rightarrow}$.

To proceed in time, LDJT calculates a forward message α_t over $\mathbf{I}_t$ using the *out-cluster* of J_t. Hence, α_t contains exactly the necessary state descriptions , as a set of parfactors, to be able to answer queries in the next time step. Afterwards, LDJT adds α_t to the local model of the *in-cluster* of J_{t+1}.

Figure 11.2 depicts passing on the current state from time step 3 to 4. To capture the state at $t = 3$, LDJT sums out the non-interface PRV $DoR_3s(X)$ from the local model and received messages of $\mathbf{C}_3^2$ and saves the result in message α_3. Increasing t by one, LDJT adds m_3 to $\mathbf{C}_4^1$'s local model. Thus, α_3 contains all state descriptions to separate time step 4 from the past. However, α_3 also possibly carries over splits from previous time steps to time step 4.

11.2 Temporal Approximate Merging

Before we introduce TAMe, we start by introducing the problem that TAMe solves.

11.2.1 Keeping Reasoning Polynomial Problem

In a temporal probabilistic relational model, evidence can slowly ground the model over time by introducing splits. We propose to name the problem of *finding how to undo splits to retain a lifted solution over time, keeping any error unbiased and necessary*, as the KRP problem. Retaining a lifted solution over time means that lifted algorithms run in polynomial time w.r.t. the domain size if a lifted solution exists (Niepert and Van den Broeck, 2014). To solve the KRP problem, an approach is required to identify any number of clusters based on how similar ϕ's of parfactors are and combine them. Our understanding of an unbiased and necessary error, is that all data points are weighted the same and that the approximation error is as small as possible while providing a high gain. To keep the error unbiased and necessary, groundings need to be accounted for and the identified cluster means have to approximate all points of the clusters well. Unfortunately, to combine similar ϕ's, we cannot use the colouring algorithm by Ahmadi *et al.* (2013) as it uses exact symmetries.

Even though LDJT instantiates vanilla FO jtree structures from $G_{\rightarrow}$, α_t carries over splits caused by evidence. Formally, the problem is that in a model $G_t = \{g_t^i\}_{i=1}^n$ at time step t, many parfactors are split. Whenever evidence leads to a split of a parfactor, the split carries over to subsequent time steps. Thus, G_t has the following form:

$$\{g_t^{i,1}, \ldots, g_t^{i,m}\}_{i=1}^n, m \in \mathbb{N}^+. \tag{11.1}$$

For each i, the different $g_t^{i,j} = \phi_t^{i,j}(\mathcal{A}^i)_{|C^{i,j}}$, $1 \leq j \leq m$, have the same arguments $\mathcal{A}^i$ but different constraints $C^{i,j}$ and varying functions $\phi_t^{i,j}$ as a result of evidence. The

assumption is that some $g_t^{i,j}$ have similar ϕ's as differences introduced by evidence are minimal or otherwise are overcome by model behaviour over time, i.e., potentials of the parfactors align again. Then, one can combine similar ϕ's while introducing only a small and bounded error in exchange for faster reasoning. In the theoretical analysis, we show that the assumption holds, by showing that ϕ's converge, allowing them to be merged, and that the error TAMe introduces is bounded.

Example 11.2.1 (Splitting of parfactors)**.** *Assume that we observe alice doing research in time step* 3*. Then, LDJT enters an evidence parfactor encoding* $DoR(X') = true$ *for* $\mathcal{D}(X') = \{alice\}$ *in* J_3*. In* J_3*, the parcluster* $\mathbf{C}_3^2$ *contains the PRV* $DoR(X)$*. Entering the evidence parfactor in* $\mathbf{C}_3^2$ *leads to splits in the local model of* $\mathbf{C}_3^2$*. The parfactor* g_3^1 *is split into a parfactor for alice and into another parfactor with a constraint encoding that the parfactor holds for all instances but alice. During message passing, the splits carry over. Thus, the parfactors* g_3^0 *and* g^R *are also split into two parts. One part for alice and another part for all other instances. Therefore, all parfactors about the logvar* X *are split in the same way into the same amount of groups, i.e., in the local modes of* J_3 *are parfactors* $g_3^{0,1}$*,* $g_3^{1,1}$ $g^{R,1}$ *about alice and parfactors* $g_3^{0,2}$*,* $g_3^{1,2}$ $g^{R,2}$ *about all other instances.*

The idea for restoring a lifted representation is to merge those $g_t^{i,j}$ with similar ϕ's into one parfactor

$$g_t^{i,k} = \phi_t^{i,k}(\mathcal{A}^i)_{|C^{i,k}} \tag{11.2}$$

where $\phi_t^{i,k}$ represents a merged version of the combined $\phi_t^{i,j}$ and $C^{i,k}$ is a union of the combined $C^{i,j}$. Merging all parfactors that behave similarly for each i leads to a G_t' of the following form with parfactors as in Eq. (11.2) and $l < m$:

$$\{g_t^{i,1}, \ldots, g_t^{i,l}\}_{i=1}^n \tag{11.3}$$

With TAMe, we present a merging scheme that takes a model G as given in Eq. (11.1) and computes a model G' as given in Eq. (11.3). It is reasonable to apply TAMe to G when transitioning from time step t to $t+1$ as the transition transfers any splits as well. In general, G may be any parfactor model and one may also transfer the idea to a DMLN model (Ahmadi *et al.*, 2013). But, models may be very large, e.g., consist of the union of all local models of an FO jtree J_t, such that finding groups for each i is too costly. Therefore, we propose to make TAMe a subroutine of LDJT. Transitioning from t to $t+1$ requires computing message α_t, which provides a state description of t that is relevant to $t+1$. Applying TAMe to α_t prepares a message with fewer groups, leading to fewer splits in J_{t+1}. Additionally, α_t normally has considerably fewer parfactors than G_t. Next, we explain in detail how to get from Eq. (11.1) to Eq. (11.3) with TAMe.

Algorithm 14 Temporal Approximate Merging

```
procedure TAMe(Model G, Radius ϵ, Significance τ)
    P ← partitioning of G based on logvars                    ▷ Eq. (11.4)
    for each partition P ∈ P do
        P ← multiply overlapping parfactors                   ▷ Eq. (11.5)
        K ← DBSCAN(P, ϵ, 2, rsim)
        if ANOVA(K, rsim′, τ) rejects H0 then
            G ← G \ P
            for each cluster K ∈ K do
                G ← G ∪ {K merged}                            ▷ Eq. (11.7)
```

11.2.2 Keeping Reasoning Polynomial with TAMe

Algorithm 14 outlines TAMe to solve the KRP problem. Inputs are a model G, possibly α_t, as well as two additional parameters, radius ϵ and significance level τ, which become important later on. The first step is to preprocess G for easier handling in subsequent steps. The main loop describes how a clustering algorithm identifies groups for merging and how groups are merged if TAMe deems the clusters to fit. The upcoming paragraphs discuss the individual steps of Alg. 14.

Model Partitioning

The preprocessing of G is a consequence of the following considerations. A challenge that arises from a model as given in Eq. (11.1) is that merging parfactors for each i independently of each other may lead to different groups that cause splits again, undoing any merging efforts. Basically, i stands for one parfactor that is split into multiple parfactors with different constraints and ϕ's by evidence. Using an i at random and transferring the grouping of the i parfactors to all other parfactors may lead to unreasonable groups for the other i's. A safe option is to multiply parfactors with overlapping constraints into one parfactor which in a worst case leads to a single cluster and very large parfactors that no longer explicitly represent independencies and may complicate calculations for messages and queries. Within LDJT, one could trace back if a set of parfactors in α_t originates from the message that has come from the direction of the *in-cluster* to the *out-cluster* as this message contains information about the past and is the origin of the most splits in α_t. Therefore, it may be possible to identify a unique i in α_t as a reasonable source for merging. However, there are no guarantees to find such an i. Instead, we opt to partition the parfactors in G based on the logvars appearing in G into a set $\mathbf{P}$ of sets of parfactors. Each partition $P \in \mathbf{P}$ has a set of logvars $\mathbf{X}_p$ that has been affected in the same way by splitting due to evidence. Formally, P has the form

$$P = \{g_t^{i,1}, \ldots, g_t^{i,m}\}_{i=1}^{n_p} \tag{11.4}$$

with $lv(g_t^{i,j}) \subseteq \mathbf{X}_p$.

Example 11.2.2 (Partitioning)**.** *In our example, α_t contains the state descriptions of the PRVs $Rep_t(X)$ and $Att_t(X)$. Both PRVs are parameterised with logvar X. Observing evidence, e.g., for $DoR_t(X)$ influences X and thereby also $Rep_t(X)$ and $Att_t(X)$ in the same way, i.e., the same splits occur. Hence, for α_t we only have one partition. Even by looking at the complete model instead of only the α messages, we only have one partition. Observing evidence for X causes splits in all parfactors in the same way. Further, observing evidence for $Pub_t(X, J)$ causes splits in X and J. Thus, here the logvars X and J would be effected in the same way by splitting due to evidence, leading again to one partition as all parfactors are effected by same way by splits due to evidence.*

In case we would have another logvar Y, which does not occur with X or J in a parfactor, then we would have a second partition. However, in case Y would occur with either X or J in a parfactor, we again would only have one partition.

The next step is to identify groups of parfactors in each partition that behave similarly.

Parfactor Clustering

After partitioning G, each partition $P \in \mathbf{P}$ of the form in Eq. (11.4) has parfactors whose constraints overlap between all i for each j. Therefore, TAMe multiplies all parfactors with overlapping constraints into one parfactor before starting with identifying groups. If there exist $g_t^{i,j}$ s.t. $lv(g_t^{i,j}) = \mathbf{X}_p$, each i refers to m parfactors with the same constraint over all i's for each j, i.e, the constraints are the same at position j for all i's. Then, multiplication in P to combine PRVs with the same constraints boils down to

$$P = \left\{ \prod_{i=1}^{n_p} g_t^{i,1}, \ldots, \prod_{i=1}^{n_p} g_t^{i,m} \right\} = \{ g_t^{p,1}, \ldots, g_t^{p,m} \} \tag{11.5}$$

where multiplying parfactors corresponds to the LVE operation of *multiply*, c.f. (Taghipour *et al.*, 2013c).

To identify groups of parfactors with similar behaviour, one needs to specify (i) what "similar behaviour" means and (ii) how to find such groups automatically. We first consider the second item, which influences specifying the first item.

TAMe needs to identify an unknown number of groups based on how similar ϕ's are. Density-based clustering groups similar points into an unknown number of groups. Therefore, TAMe uses density-based clustering. For the evaluation, we instantiate TAMe with density-based spatial clustering of applications with noise (DBSCAN) (Ester *et al.*, 1996; Schubert *et al.*, 2017) as the clustering approach. In the following, we illustrate how density-based clustering fits into the overall scheme of TAMe using DBSCAN. DBSCAN identifies data points as core points if in their neighbourhoods, determined by a radius ϵ around a point, lie a certain number $minPts$ of other data points. A core data point

makes up a cluster along with all the data points in its neighbourhood, which recursively proceeds with the next core data point in the neighbourhood. To determine data points in a neighbourhood, DBSCAN requires a distance function as an input. DBSCAN is able to detect outliers, i.e., points which do not occur in any neighbourhood. For the purpose of clustering parfactors, we set $minPts$ to 2 to be able to cluster even two parfactors. The distance measure should assess how similarly parfactors behave, with 0 meaning identical behaviour and larger values meaning less similar behaviour.

To determine the similarity of the behaviour of two parfactors, one could calculate marginal distributions for a PRV that occurs with split constraints and compare if the marginals are in a certain Δ area. However, marginal distributions could result from completely different potentials and be similar by chance. The potentials of a parfactor on the other hand specify the current weight for each possible assignment. Thus, in case the ratio of the potentials of two parfactors are similar, they also have similar marginal distributions and behave similarly.

Example 11.2.3 (Similar parfactors)**.** *A parfactor mapping to* 4 *and* 2 *and another parfactor mapping to* 8.1 *and* 3.9 *behave similarly. Both parfactors weight the first assignment about twice as much as the second. Assuming both parfactors are independent from the rest and only have one grounding each, the marginals for true would be* 0.667 *and* 0.675 *respectively, i.e., less than* 0.01 *apart from each other. The same case arises for two parfactors mapping to* $\langle 4, 2\rangle$ *and* $\langle 4.1, 1.9\rangle$ *respectively.*

Such potentials, when thought of as vectors, have a small angle between them, i.e., a high cosine similarity, which we use to specify "similar behaviour". For the setup of the similarity of two parfactors $g_t^{i,j_1} = \phi_t^{i,j_1}(\mathcal{A}^i)_{|C^{i,j_1}}$ and $g_t^{i,j_2} = \phi_t^{i,j_2}(\mathcal{A}^i)_{|C^{i,j_2}}$, we use a function $rsim : (\times_{i=1}^n range(A^i) \mapsto \mathbb{R}^+, \times_{i=1}^n range(A^i) \mapsto \mathbb{R}^+) \mapsto \mathbb{R}^+$ that is defined as follows:

$$rsim(\phi_t^{i,j_1}, \phi_t^{i,j_2}) = 1 - \frac{\sum_{\mathbf{a} \in range(\mathcal{A}^i)} \phi_t^{i,j_1}(\mathbf{a}) \cdot \phi_t^{i,j_2}(\mathbf{a})}{\sqrt{\sum_{\mathbf{a} \in range(\mathcal{A}^i)} \phi_t^{i,j_1}(\mathbf{a})^2} \cdot \sqrt{\sum_{\mathbf{a} \in range(\mathcal{A}^i)} \phi_t^{i,j_2}(\mathbf{a})^2}} \tag{11.6}$$

The result of Eq. (11.6) lies in the interval $[0, 1]$. The fraction is the definition of the cosine similarity. We calculate 1 minus the fraction to get a "distance" measure, in which a lower value means a closer distance.

As a consequence of $rsim$ with its codomain $[0, 1]$ as the distance function for DBSCAN, ϵ needs to be ≤ 1. Overall, the inputs of DBSCAN for clustering parfactors are a partition P of parfactors, ϵ, $minPts = 2$, and $rsim$. The radius ϵ trades off cluster sizes with accuracy. The output is a clustering of P, i.e., a set $\mathbf{K}$ of sets in which each $K \in \mathbf{K}$ is a set of parfactors that are assumed to behave similarly.

Fitness of Clustering

The question that remains after clustering is: How good is the clustering? The clustering is highly influenced by the choice of the radius ϵ, which leads to large clusters if set to a high value but may also blur the potentials in the merged parfactor to a higher degree.

One could calculate the error introduced by the clustering w.r.t. a given PRV B by comparing marginal distributions of B before and after merging. However, if a model already is highly shattered, the computational effort can be very high to compute marginal distributions before merging.

DBSCAN clusters together parfactors with a small angle between them. So a clustering fits if the variance of angles within clusters is low and the variance of angles between clusters is high. Analysis of variance (ANOVA) (Fisher, 1925) is a statistical method to test for significance of a clustering. In our setup, ANOVA computes the variance of each parfactor in a cluster $K \in \mathbf{K}$ w.r.t. the mean parfactor of K as well as the variance of the mean parfactor of K w.r.t. the mean parfactor of all points in $\mathbf{K}$. Hence, it provides an indication of how good the clustering separates parfactors.

ANOVA is used to accept or reject hypotheses. The default hypothesis is that the means of all clusters are equal. For our problem, the default hypothesis H_0 is that the mean parfactors of the clusters are equal, i.e., are not statistically significant to discriminate clusters. The goal is to be able to reject H_0, that is to say there is more difference between than within clusters. In case TAMe can reject H_0, at least one cluster is significantly different from the others.

To compute a mean parfactor of a cluster K, TAMe calculates the average of all potentials while accounting for groundings. Formally, given a set of parfactors $\{\phi_t^{i,j}(\mathcal{A}^i)_{|C^{i,j}}\}_{j=1}^m$, a mean parfactor $g_t^{i,k} = \phi_t^{i,k}(\mathcal{A}^i)_{|C^{i,k}}$ is determined by

$$\phi_t^{i,k}(\mathbf{a}) = \frac{\sum_{j=1}^m |gr(\phi_t^{i,j}(\mathbf{a})_{|C^{i,j}})| \phi_t^{i,j}(\mathbf{a})_{|C^{i,j}}}{|gr(\phi_t^{i,k}(\mathbf{a})_{|C^{i,k}})|} \tag{11.7}$$

for each $\mathbf{a} \in range(\mathcal{A}^i)$ and $C^{i,k}$ is a union of the different $C^{i,j}$. Thus, TAMe goes through all potentials and for each assignment, adds the current potential, which is multiplied by the number of groundings of the current parfactor. After all potentials for one assignment are added up and the groundings are accounted for, TAMe divides the potential by the number of overall groundings to obtain a mean potential.

Example 11.2.4 (Mean parfactor)**.** *To illustrate Eq.* (11.7)*, consider a cluster with* 3 *parfactors. The first parfactor maps to the potentials* 2 *and* 1 *with* 2 *groundings, the second maps to* 3.9 *and* 1.9 *with* 5 *groundings, and the third maps to* 8.1 *and* 4 *with* 1 *grounding. To calculate the mean potential, TAMe calculates for the first mapping* $(2 \cdot 2 + 5 \cdot 3.9 + 1 \cdot 8.1)/8 = 3.95$ *and for the second mapping* $(2 \cdot 1 + 5 \cdot 1.9 + 1 \cdot 4)/8 = 1.9375$*. Thus, the mean parfactor maps to* 3.95 *and* 1.9375 *with* 8 *groundings.*

To calculate variances of parfactors, TAMe uses $rsim$ as the clusters have been built based on $rsim$. The intuition behind the choice is that if two parfactors have a very small angle between their potentials, then the variance of the potentials would be close to 0. The variance increases with the angle between potentials. As the number of groundings influences the new potentials, we also include the number of groundings while calculating variances. A parfactor that represents more groundings has a greater weight than one parfactor with one grounding. As we have a grounding semantics, grounding a parfactor with more instances leads to more factors contributing to the full joint distribution.

After computing a mean parfactor $g_t^{i,k}$ for each cluster $K \in \mathbf{K}$ and an overall mean parfactor $g_t^{i,m}$ based on all parfactors in $\mathbf{K}$, ANOVA proceeds to compute the variation between groups, i.e., MSG, and the variation within groups, i.e., MSE, using Eq. (11.6) and the groundings of parfactors:

$$MSG = \frac{1}{l-1} \sum_{K \in \mathbf{K}} |gr(g_t^{i,k})| \cdot (rsim(g_t^{i,k}, g_t^{i,m}))^2$$

$$MSE = \frac{1}{m-l} \sum_{K \in \mathbf{K}} \sum_{g_t^{i,j} \in K} |gr(g_t^{i,j})| \cdot (rsim(g_t^{i,j}, g_t^{i,k}))^2$$

where $l = |\mathbf{K}|$, i.e., number of clusters, and $m = |gr(\mathbf{K})|$, i.e., number of overall groundings. Computing $F = \frac{MSG}{MSE}$, ANOVA compares F against a critical value F_{crit}, which depends on τ, $l-1$, and $m-l$ and can be looked up in a pre-computed table. If $F \leq F_{crit}$, TAMe accepts H_0 and discards the clustering. In case TAMe rejects H_0, i.e., $F > F_{crit}$, there is more difference between clusters than within clusters and TAMe proceeds to merging parfactors in each cluster.

Merging Parfactors

The new parfactor for each cluster $K \in \mathbf{K}$ is the mean parfactor $g_t^{i,k}$, which is already computed by ANOVA. TAMe replaces P in G with the merged parfactors. Then, TAMe proceeds with the next partition, identifying and checking a clustering for the new partition, until all partitions are processed. The result is a model whose parfactors are merged versions of the input model, partially restoring a lifted representation. Given a forward message α_t, the output is a message that possibly contains fewer groups within logvars and thus, prevents ongoing splitting over time.

Application Cycle

As ANOVA may determine that the clustering is not fit enough, TAMe may incur overhead if TAMe cannot merge groups. Therefore, TAMe does not need to be applied at every time step. Normally, the model is slowly grounded over time with evidence, but if

the groups behave similarly, which is the case due to the impact of the model, the reoccurring application of the model behaviour results in the potentials being similar enough for TAMe to merge them. Thus, based on how much evidence splits up the model, the interval of how often TAMe should be used as a subroutine needs to be determined.

Next, we look at theoretical implications of TAMe.

11.3 Theoretical Analysis

We show that TAMe introduces a necessary, unbiased, and bounded error and that TAMe keeps reasoning polynomial.

Proposition 11.3.1. *TAMe errors are necessary and unbiased.*

Due to a density-based clustering, TAMe clusters parfactors with similar ϕ's. ANOVA determines the fitness of clusterings to prevent unnecessary errors. By accounting for groundings during merging, the error is unbiased.

Knowing that TAMe produces necessary and unbiased errors, let us have a look at theoretical bounds of the approximation error TAMe introduces as well as whether groups with only slightly different evidence actually do converge, allowing TAMe to keep reasoning polynomial. Boyen and Koller (1998) show that for a Markov process with a stochastic transition model $\mathcal{Q}$, an error introduced by an approximation of a belief state is at least reduced by the factor $(1-\gamma_{\mathcal{Q}})$ when transitioning from t to $t+1$, with $\gamma_{\mathcal{Q}}$ being a so-called minimal mixing rate. $\gamma_{\mathcal{Q}}$ is the minimal extent to which the model behaviour causes an approximation to converge to the true belief state while transitioning from one time step to the next. From that insight, they induce that an error δ introduced by approximating a belief state of a temporal process is bounded indefinitely by $\delta/\gamma_{\mathcal{Q}}$. We use $\gamma_{\mathcal{Q}}$ for two contributions, (i) TAMe introduces a bounded error and (ii) TAMe is able to merge parfactors to keep reasoning polynomial.

The proof that TAMe introduces a bounded error is based on Boyen and Koller (1998). We begin by showing that a PDM G is a Markov process with a stochastic transition model $\mathcal{Q}$. Afterwards, we define the so-called minimal mixing rate $\gamma_{\mathcal{Q}}$ for G and use $\gamma_{\mathcal{Q}}$ to show that the error TAMe introduces is bounded indefinitely.

Lemma 11.3.1. *A PDM is a Markov process with a stochastic transition model $\mathcal{Q}$.*

Proof. A PDM $(G_0, G_{\rightarrow})$ follows the first-order Markov assumption and therefore, is a Markov process. Further, in a PDM, the PM $G_{\rightarrow}$ describes how a model transitions from one belief state to the next. Given the semantics of a PM, $G_{\rightarrow}$ forms a stochastic transition model $\mathcal{Q}$. Therefore, it holds that a PDM is a Markov process with a stochastic transition model $\mathcal{Q}$. □

To show that the error TAMe introduces is bounded indefinitely, we begin by proposing some notations. Let $\Omega = \{\omega_1, .., \omega_n\}$ refer to the state space for time slice $t-1$, i.e.,

the previous time slice, and $\Omega' = \{\omega'_1, .., \omega'_n\}$ refer to the state space for time slice t, i.e., the current time slice. For a PDM G, Ω consists of $\times_{A \in gr(\mathbf{A}_{t-1})} \mathcal{R}(A)$ and Ω' consists of $\times_{A \in gr(\mathbf{A}_t)} \mathcal{R}(A)$ from $G_{\rightarrow}$. Thus, Ω contains all possible worlds from time slice $t-1$ of $G_{\rightarrow}$ and Ω' contains all possible worlds from time slice t of $G_{\rightarrow}$. A world is one particular assignment of all grounded PRVs. After renaming, the state spaces Ω and Ω' are the same, i.e., the only difference is that in Ω all randvars are from time slice $t-1$ and in Ω' the same randvars are from time slice t. Further, $G_{\rightarrow}$ represents a process from Ω to Ω'. Further, let $\omega_{i_1} \in \Omega$ be one possible world and let $\omega_{i_2} \in \Omega$ be another possible world. Then, for some posterior world $\omega'_j \in \Omega'$, the probability of transferring from ω_{i_1} to ω'_j is $P(\omega'_j \mid \omega_{i_1})$ and the probability of transferring from ω_{i_2} to ω'_j is $P(\omega'_j \mid \omega_{i_2})$ w.r.t. $P_{G_{\rightarrow}}$. The queries ask for the probability of a certain world, here ω'_j, in time slice t given the assignments of a world from time slice $t-1$, ω_{i_1} or ω_{i_2}, as evidence. Using these notations, we define the so-called minimal mixing rate for a PDM.

Definition 11.3.1 (Minimal mixing rate)**.** For a PDM with a stochastic transition model $\mathcal{Q}$, the minimal mixing rate of $\mathcal{Q}$ is

$$\gamma_{\mathcal{Q}} = \min_{i_1, i_2} \sum_{j=1}^{n} \min[P(\omega'_j \mid \omega_{i_1}), P(\omega'_j \mid \omega_{i_2})]$$

Basically, $\gamma_{\mathcal{Q}}$ describes the extent to which all pairs of possible worlds from time slice $t-1$ at least agree on all possible worlds in time slice t based on the temporal model behaviour. The value of $\gamma_{\mathcal{Q}}$ is always between 0 and 1. The intuition behind $\gamma_{\mathcal{Q}} \in [0, 1]$ is that summing over all possible worlds from time slice t yields 1, i.e., $\sum_{j=1}^{n} P(\omega'_j) = 1$. However, for $\gamma_{\mathcal{Q}}$ we do not only sum over all possible worlds from time slice t, but have possible worlds from time slice $t-1$ as evidence. Thus, roughly speaking we multiply the probabilities of the possible world from time slice t with a value between 0 and 1. Therefore, $\gamma_{\mathcal{Q}}$ is between 0 and 1.

To be able to measure the distance between the true belief state, let us say φ, and an approximation of that true belief state, let us say ψ, we introduce the Kullback–Leibler divergence (Kullback and Leibler, 1951), also called KL-divergence. The KL-divergence can be used to measure the discrepancy of a distribution and its approximation.

Definition 11.3.2 (KL-divergence)**.** If φ and ψ are two distributions over the same space Ω, the KL-divergence of φ to ψ is given by:

$$\mathbf{D}(\varphi || \psi) = \sum_{\omega \in \Omega} \varphi(\omega) \log \frac{\varphi(\omega)}{\psi(\omega)}$$

For each summand, we multiply the probability of the true belief state $\varphi(\omega)$ with the logarithm of the division of the probabilities of $\varphi(\omega)$ and its approximation $\psi(\omega)$. Thus, in case $\varphi(\omega) = \psi(\omega)$ the distance is 0 and otherwise the distance is greater than 0.

Further, let φ' and ψ' be the distributions corresponding to φ and ψ induced over Ω' by $G_{\rightarrow}$. With the KL-divergence, we can define how an approximation error gets reduced by the model behaviour while transitioning from time step $t-1$ to time step t. Then, with the minimal mixing rate $\gamma_{\mathcal{Q}} \in [0,1]$ the following holds:

$$\mathbf{D}(\varphi' || \psi') \leq (1-\gamma_{\mathcal{Q}})\mathbf{D}(\varphi||\psi). \tag{11.8}$$

Based on Definition 11.3.2, two distributions, in this case the true belief state and an approximation of the true belief state, converge with at least $\gamma_{\mathcal{Q}}$. That is to say the belief state of the approximation converges to the true belief state with the rate $\gamma_{\mathcal{Q}}$. In terms of KL-divergence, converging means that the distance between the true belief state and the approximation of the true belief state is reduced at least by the factor $(1-\gamma_{\mathcal{Q}})$ while transitioning from one time step to the next.

Now, we can show that the error that TAMe introduces is indefinitely bounded. Let σ be the true belief state, let $\hat{\sigma}$ be TAMe applied to σ, i.e., an approximation of σ, and let $\tilde{\sigma}$ be TAMe applied to $\hat{\sigma}$, i.e., an approximation of the approximated belief state. TAMe is constantly applied, e.g., every time step. Thus, TAMe approximates an already approximated belief state. Hence, to measure the error that TAMe introduces, we need to compare the KL-divergence of σ to $\hat{\sigma}$ against the KL-divergence of σ to $\tilde{\sigma}$. Further, there exists a δ such that the following holds:

$$\mathbf{D}(\sigma||\tilde{\sigma}) - \mathbf{D}(\sigma||\hat{\sigma}) \leq \delta. \tag{11.9}$$

Using Eq. (11.9), we can deduce that applying TAMe introduces an error of at most δ. The underlying assumption of Eq. (11.9) is that by applying an approximation, the distance of the approximated belief state to the true belief state increases. By applying TAMe, the distance between the true belief state and its approximation is at most increased by δ.

Before TAMe merges parfactors and thereby, approximates a belief state, TAMe uses a significance check to determine the fitness of a proposed clustering. Therefore, one can use the significance check to obtain a small δ. Additionally, from Eq. (11.8), we know the distance of an approximation to the true belief state is reduced at least by the factor $(1-\gamma_{\mathcal{Q}})$ while transitioning from t to $t+1$. Using these insights, we can provide error bounds for TAMe.

Theorem 11.3.1. *TAMe error is indefinitely bounded by $\delta/\gamma_{\mathcal{Q}}$.*

Proof. We have shown that a PDM is a Markov decision process with a stochastic transition matrix $\mathcal{Q}$ that has a minimal mixing rate $\gamma_{\mathcal{Q}}$. TAMe approximates the belief state of the interface $\mathbf{I}_t$ and LDJT computes the transition from $t-1$ to t. From Eq. (11.9), we know that applying TAMe increases the distance of the approximated belief state to the true belief state by at most δ. Further, from Eq. (11.8) we know that LDJT

reduces the distance between the approximated belief state and the true belief state at least by the factor $(1 - \gamma_{\mathcal{Q}})$ with each transition. To show that the error is indefinitely bounded, we need to show that the accumulated error over all time steps is bounded. Thereby, the distance between the approximated belief state and the true belief state is also bounded. For the current time step, TAMe introduces an error of δ, the error from the previous time step is at most $(1 - \gamma_{\mathcal{Q}}) \cdot \delta$, and the error from the first approximation is $(1 - \gamma_{\mathcal{Q}})^{t-1} \cdot \delta$ at time step t. By approximating each time step, the expected error up to time step t accumulates to $\delta + (1 - \gamma_{\mathcal{Q}}) \cdot \delta + ... + (1 - \gamma_{\mathcal{Q}})^{t-1} \cdot \delta = \sum_{i=0}^{t} \delta \cdot (1 - \gamma_{\mathcal{Q}})^i \leq \sum_{i=0}^{\infty} \delta \cdot (1 - \gamma_{\mathcal{Q}})^i = \delta / \gamma_{\mathcal{Q}}$. For the last step, we apply the geometric series, i.e., $\sum_{i=0}^{\infty} \delta \cdot (1 - \gamma_{\mathcal{Q}})^i = \delta / 1 - (1 - \gamma_{\mathcal{Q}}) = \delta / \gamma_{\mathcal{Q}}$. Thus, the error is indefinitely bounded by $\delta / \gamma_{\mathcal{Q}}$. □

Let us now prove that TAMe keeps reasoning polynomial by showing that TAMe can merge distributions and thereby, LDJT can calculate a lifted solution.

Theorem 11.3.2. *TAMe keeps reasoning polynomial.*

Proof. Without loss of generality, assume we observe evidence for one unary and boolean PRV $B(Y)$. Observing events for multiple instances of logvar Y can split Y into at most three parts, i.e., the true, the false, and the unknown part. Hence, for each time step, LDJT can only assign true, false, or unknown to each grounded PRV of $B(Y)$. For a huge n, where $|\mathcal{D}(Y)| = n$, there always is a large number of ground PRVs with the same subsequence of events. In case these ground PRVs have been split by evidence, the minimal mixing rate $\gamma_{\mathcal{Q}}$ ensures that the distributions of the ground PRVs converge again. With a subsequence of length o say, the distance between the split distributions of these ground PRVs is reduced by $(1 - \gamma_{\mathcal{Q}})^o$. Therefore, the split distributions converge again and TAMe will merge the split distributions at some point in time. Merging parfactors ensures that LDJT calculates a solution in polynomial time w.r.t. domains. □

Based on Thm. 11.3.2 and the minimal mixing rate $\gamma_{\mathcal{Q}}$, we show that without new evidence, TAMe obtains a fully lifted representation again and that the representation represents the true belief state.

Corollary 11.3.1. *Without new evidence, TAMe obtains a fully lifted representation with the true belief state.*

Proof. During each transition from t to $t+1$, $\gamma_{\mathcal{Q}}$ ensures that approximated distributions converge to the true distribution as the distributions converge at least by the factor $(1 - \gamma_{\mathcal{Q}})$. Thus, the approximated distributions converge to the true belief state without new evidence. Further, all groups have the same origin. Therefore, all groups converge to the same true belief state. Hence, TAMe can merge all groups and thereby, obtain a fully lifted representation at some point in time again without new evidence. □

Thus, TAMe solves the KRP problem. Since the underlying distributions of ϕ's converge, TAMe is able to merge ϕ's allowing TAMe to keep reasoning polynomial. Further, TAMe introduces a bounded, unbiased, and necessary error.

11.4 Evaluation

For the evaluation, we compare runtimes of LDJT with and without TAMe and have a look at the introduced error. We use the model G^{ex} from this chapter with $|\mathcal{D}(X)| = 100$ and divide these 100 persons equally into symmetry groups, where members of each group behave identically over time. For one time step, each symmetry group has the same evidence, but the evidence can change from one time step to the next. To break symmetries within a group, evidence may be missing with a probability of 0.1 for each person. We split $D(X)$ into 2 to 10 symmetry groups and generate evidence for 20 time steps. For each symmetry group i, LDJT answers $A_{t+\pi}(x_i)$ for $\pi = \{0, 5, 10\}$, i.e., a filtering and two prediction queries, in each time step t for all 20 time steps.

We vary ϵ and the interval I of how often LDJT applies TAMe. The parameterτ is fixed to 0.005. Based on the problem at hand, an appropriate τ needs to be determined in advance (Benjamin *et al.*, 2018). The three options we evaluate are, from conservative to aggressive: 1) $I = 5$, $\epsilon = 5 \cdot 10^{-14}$, 2) $I = 5$, $\epsilon = 5 \cdot 10^{-2}$, and 3) $I = 2$, $\epsilon = 5 \cdot 10^{-2}$. TAMe with Option 1 mostly merges parfactors that only differ in a scaling factor. TAMe with Options 2 and 3 also merges parfactors that slightly differ in their ratio. With $I = 2$, LDJT calls TAMe every other time step, and with $I = 5$ every fifth time step.

Figure 11.3 shows runtimes of LDJT without TAMe and with TAMe for the three options. The number of symmetry groups is plotted on the x-axis. With more symmetry groups, evidence can ground the model faster over time. Thus, the runtimes correlate to the number of groups. For 5 symmetry groups, LDJT without TAMe takes about twice as long as LDJT with TAMe using the conservative option (1), answering 300 queries for the 20 time steps. However, for 8 symmetry groups, LDJT without TAMe is slightly faster. As merging depends on evidence, which here is randomly generated, TAMe may not always be able to trade off its overhead. TAMe with Option 2 merges more parfactors. Hence, every fifth time step, LDJT answers queries on fewer groups, which are then again split up by evidence. With the most aggressive option (3), LDJT applies TAMe every other time step and thus answers queries on highly lifted models.

In summary, even by only merging parfactors that hardly differ, TAMe merges enough parfactors to improve runtimes of LDJT. TAMe with Options 2 and 3 improves runtimes of LDJT significantly. Overall, TAMe is able to save runtime of LDJT of up to 2 orders of magnitude. Knowing that TAMe can significantly improve the performance of LDJT, we look at the costs of the speed up, namely the introduced error.

Table 11.1 shows the error in the marginals for 10 symmetry groups for the most aggressive option, when performing filtering, 2 time step prediction, and 4 time step prediction

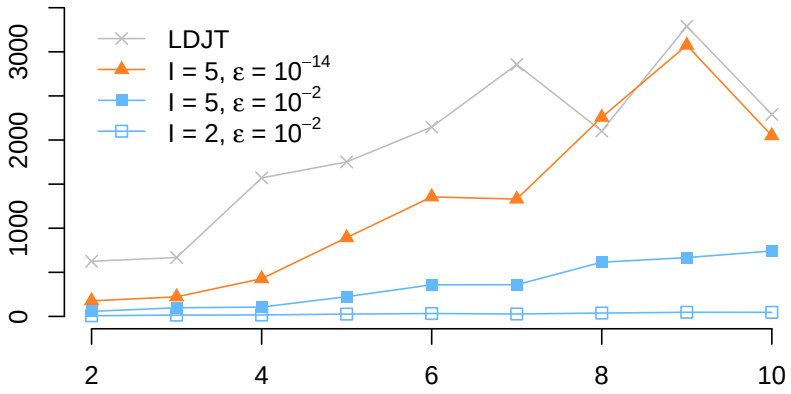


Figure 11.3: Runtimes [seconds], x-axis: #symmetry groups

π	Max	Min	Average
0	0.0001537746121	0.0000000001720	0.0000191206488
2	0.0000000851654	0.0000000000001	0.0000000111949
4	0.0000000000478	0	0.0000000000068

Table 11.1: Introduced error; $I = 2$, $\epsilon = 5 \cdot 10^{-2}$, 10 symmetry groups

for each instance and each time step. For filtering queries, the error is already negligible and decreases for prediction queries. Thus, the empirical evaluation underscores that TAMe can keep reasoning polynomial, introducing only a negligible error. Further, the error converges to the true belief state without new evidence as the prediction queries show. Next, we take a look at the significance check.

To empirically evaluate the significance check, we run LDJT with TAMe on a model once with and once without the significance check. Table 11.2 shows the introduced errors for these runs. The maximum error hardly differs between the two runs, which is is due to the error being bounded. Further, the minimum error is lower with the significance check as the significance check does not accept all proposed clusters. Discarding a clustering and thus, not following through with another approximation, the current approximation and the true belief state continue to converge based on the mixing rate. Lastly, the

	Max	Min	Average
w	0.0002259927071	0.0000000000000	0.0000104567643
w/o	0.0002260554389	0.0000000000168	0.0000137870835

Table 11.2: Introduced error; with and without significance test

average error without the significance check is around 32% higher. Even though in this case both average errors are negligible on an absolute scale, the average error on a relative scale without the significance check does increase significantly.

Overall, we show empirically that TAMe does not introduce any unnecessary error due to the significance check and that TAMe keeps reasoning polynomial for LDJT.

11.5 Interim Conclusion

Evidence often grounds a temporal model over time. Consequently, inference runtimes suffer. Hence, the idea is to use approximate symmetries to restore a lifted representation and thus, keep reasoning polynomial by taming evidence. To the best of our knowledge, we present the first approach solving the KRP problem for temporal relational probabilistic models. The algorithm can be used within any (exact or approximate) temporal inference algorithm (Contribution **8a**). The main idea is that instances of parfactors with similar ratios between potentials behave similarly. To merge parfactors, TAMe uses a the forward message of LDJT as this message is smaller than the model and causes splits in the next time step. To identify similar instances, TAMe uses density-based clustering with the cosine similarity as a distance measure, which captures similarity of potentials. TAMe applies ANOVA to the clustering result to check if the cluster means significantly discriminate the clusters. We show that TAMe can merge parfactors as their distributions converge and that TAMe introduces a bounded error (Contribution **8b**). Additionally, the approximated distributions converge to the true distributions and without new evidence TAMe obtains a fully lifted representation again. Empirical results show that LDJT with TAMe significantly outperforms LDJT without TAMe. The results support our analysis that TAMe retains a lifted solution, while keeping the introduced error negligible. Hence, LDJT with TAMe produces fast and precise results.

Future work includes how to approximate evidence (Van den Broeck and Darwiche, 2013) to cause fewer splits in temporal models. Additionally, we plan to look into using TAMe for forgetting. One could imagine that we have a number of objects, but sometimes new objects are introduced and old object disappear, Here, we could use the ideas of TAMe to handle the forgetting and insertion of new objects.

Chapter 12

Outlook

Before we conclude, we present our ideas for changing domains over time as an interesting direction of inference in temporal probabilistic relational models. Changing domains means that the number of instances to reason over changes over time, i.e., some instances get added and others get removed. Interesting questions concern, for example, the semantics of changing domains, e.g., are we allowed to assume that an instance got added only on this time step or has the change of the domain size also an influence on previous time steps. Having a proper semantics for changing domains, we can also think about changing domains to identify a world which best explains our observations. Based on these questions, in the following, we provide a semantics for changing domain sizes. Afterwards, we investigate the possibility that there is uncertainty about when exactly a change in a domain size happens. Finally, we look into forgetting and reusing instances.

Semantics Having a grounding semantics, changes in domain sizes are straightforward if we can say with certainty that the change occurs from one time step to the next. In the ground model, the change in domain sizes would result in adding or removing corresponding randvars. Thus, from one time step to the next, the full joint distribution would change. Further, observing evidence for newly added instances might have an effect on other PRVs and their instances. The change could influence hindsight queries. Such a case in a ground model means that newly added randvars could be connected over other randvars to randvars of other instances in previous time steps. Then, observing evidence for the newly added randvars also influences randvars of other instances in previous time steps and could change the distributions in hindsight queries.

However, a forward message when the domain size increases between time steps is not as straightforward. Here, we could imagine that we have an inter-slice parfactor mapping $S_{t-1}(X)$ to $S_t(X)$ and $|\mathcal{D}(X)| = 10$ for time step $t-1$ and $|\mathcal{D}(X)| = 11$ for time step t. In such a case, LDJT would have to split X in time slice t into the x's from time step $t-1$ and the x's from time step t. Thus, LDJT would introduce a new group to reason over by adding new instances. Further, for all PRVs LDJT may have a form of priors for the very first time step. The following question arrises: Should the newly added randvar also have priors and if yes, which priors. We could use the very same prior as for the very first time step. But is it reasonable to do so as the model has already been proceeding

in time? Additionally, by using the priors from the first time step, LDJT would have to hope that it could merge the newly added group with other groups at some point in time with TAMe to not get lost in reasoning over too many groups. Another approach for the prior would be to use the posterior of $S_{t-1}(X)$ as a prior for $S_t(x_{11})$. Such an approach goes along with the idea that the instances behave identically. Further, in such a fashion, LDJT would not have to split x_{11} from the rest and thereby, would not introduce a new group. Therefore, using the posterior of the previous time step would be valid based on the assumptions we make about the model. The approach becomes more involved, in case the X in $S_{t-1}(X)$ is split into multiple parts. Here, one could use the posterior of the group that encodes the most instances.

Uncertain Changes Another interesting part in changing domain sizes would be uncertainties. Assume that there is an oracle telling you that within the next ten time steps the domain size will increase by one, then there is an uncertainty about when exactly the domain change happens. Such an oracle could describe, for example, how a population changes over time. With uncertainties, one has to reason over distributions of full joint distributions. The uncertainty could be modelled as a distribution over time steps stating how likely different domain sizes are. In such a case with discrete time steps, one might be able to combine full joint distributions w.r.t. how probable their corresponding domain sizes are. But the combination of the full joint distributions is not a trivial case as their sizes do not match. To be able to reason with such an uncertainty, LDJT could answer queries on the applicable full joint distributions and combine the marginal distributions with how probable the corresponding full joint distributions are.

Formally, for each full joint distribution we would have a tuple (p, P_G), where p is the probability that the full joint distribution P_G of a particular domain size change is applicable. Then, LDJT can answer queries w.r.t. the different P_G, multiply the result with p and sum over all applicable full joint distributions. To determine how probable a full joint distribution is, LDJT has to account for the distribution over time encoding how the domain size changes. For the first time step within this temporal domain size uncertainty, LDJT can calculate the probability of the domain size staying the same as well as the probability of the domain size changing based on the defined uncertainty distribution. In our earlier example, that corresponds to the probability that the domain size remains at 10 or changes to 11. After the first time step, LDJT has two full joint distributions with probabilities of how likely they are. Thus, LDJT also needs to compute a forward message for both full joint distributions. In the next time step, LDJT again can calculate how likely it is that the domain size is still 10 as well as how likely it is that there are already 11 instances. The case that there are still 10 instances is the more trivial case. Here, LDJT has to use the forward message from the previous time step with also 10 instances. The case where there are 11 instances in time step $t + 1$ can originate from both full joint distributions from time step t. Either, the domain has

already changed in the last time step to 11 or the domain size changes in the current time step from 10 to 11. Therefore, how probable the full joint distributions in $t+1$ are needs to be combined with how probable the full joint distributions are in time step t. Hence, the number of full joint distributions increases over time. If necessary, to reduce the number of full joint distributions, LDJT could either approximate full joint distributions by pruning highly unlikely full joint distributions or could apply TAMe to combine some full joint distributions that are already over 11 instances. After the last time step of the domain size uncertainty, LDJT has only full joint distributions over 11 instances, which then can be again combined based on how probable the different full joint distributions are to consolidate the full joint distributions into a single full joint distribution again. The end is also reached in case LDJT receives an event about the newly added instance. Then, LDJT is certain that there are 11 instances for the current time step, but still needs to consolidate the full joint distributions.

With continuous time steps, uncertain changes could also be of interest to Gaussian processes (Rasmussen, 2003). To the best of our knowledge, the time when an event is observed is certain in Gaussian processes. Temporal uncertainty when an event is observed could also be an interesting extension for Gaussian processes. However, in Gaussian processes one would not restrict the uncertainty to domain size changes but have in general vagueness about when an event is observed.

Forgetting and Reusing Instances Having a fully defined semantics for changing domains, one could also think about using temporal probabilistic relational models to perform discourse analysis. Assume we want to analyse a novel. In a novel, instances or persons may vary in different chapters. By varying, we mean that sometimes instances do not occur in a chapter, are newly introduced, or appear again after several chapters. To solve such a scenario, one needs to be able to determine which possible world best describes the evidence. Thus, one needs to solve an optimisation problem over possible worlds with different domain sizes and different instance assignments. To solve the optimisation problem, one needs to be able to determine whether introducing a new instance or reactivating an already existing instance, including the observed events for that instance, makes a world more probable. To forget an instance between chapters, one could imagine to define an approach in the spirit of TAMe to not drown in a sea of superfluous knowledge. Therefore, in such a case to solve the optimisation problem one also needs to account for the value of information. Storing too many observations and forgetting too few instances will cause runtimes to drastically suffer, while forgetting too much knowledge might make it infeasible to discriminate possible worlds leading to an over-approximated and thereby, not useable result. To be able to solve the optimisation problem, we have already taken a crucial step with TAMe.

Chapter 13

Conclusion

In this thesis, we present LDJT with multiple extension. LDJT efficiently answers multiple *hindsight*, *filtering*, and *prediction* queries for temporal probabilistic relational models (Contribution **1**). Further, we show that simply lifting a propositional algorithm might result in grounding a model unnecessarily. To ensure preconditions of lifting, LDJT prevents unnecessary groundings (Contribution **2**). In our theoretical analysis of LDJT (Contribution **3**), we show that there is a trade off between handling temporal aspects efficiently and lifting, resulting in restricting completeness results for lifted inference algorithms for LDJT. Further, our complexity results indicate that lifting especially matters in the temporal case as in the ground case the interface and thereby the ground width increases with the domain size. The empirical evaluation of LDJT also shows that lifting matters as well as that, for temporal probabilistic relational models, an inference algorithm has to handle temporal aspects efficiently.

In its base form, LDJT can answer marginal queries with a single query term efficiently. To allow more query types, we extend the query language of LDJT in Part II. The first extension to the query language that we present is conjunctive queries (Contribution **4a** and **4b**). To answer conjunctive queries, LDJT^{con} delays the elimination of PRVs of query terms until all query terms are in a single parcluster. In a temporal model, the parclusters holding the query terms can be far apart leading to a huge subtree in between them. Therefore, LDJT^{con} does not merge subtrees (as LJT does), but delays the elimination. In the empirical evaluation, we also show that LDJT^{con} is beneficial for multiple conjunctive queries, e.g., referring to different representatives. For assignment queries (Contribution **5a**, **5b**, and **5c**), our second extension we show how LDJT can handle temporal aspects efficiently. Further, we identify safe MAP queries, namely queries over complete time steps, e.g., the last 20 time steps, which are of great practical interest. The last extension that we present is maximum expected utility queries (Contribution **6a**, **6b**, and **6c**) to support decision making.

In Part III, we introduce uncertain evidence (Contribution **7a**, **7b**, and **7c**) and tame the effects of evidence over time (Contribution **8a** and **8b**). Evidence grounding a model over time is a key challenge for inference in temporal probabilistic relational models. TAMe solves the KRP problem by restoring a lifted representation. We theoretically show that TAMe introduces a bounded error and that TAMe solves the KRP problem.

To the best of our knowledge LDJT is the first *exact* inference algorithm for *temporal* probabilistic relational models calculating a lifted solution if the model permits. In addition to *hindsight*, *filtering*, and *prediction* queries, LDJT also allows for other types of queries such a conjunctive, assignment, or maximum expected utility queries. Lastly, with TAMe, we solve a key challenge for inference in temporal probabilistic relational models allowing for approximating symmetries to keep reasoning polynomial.

Besides the changing domain sizes in Chapter 12, another interesting direction for future work is temporal and spatial uncertainties. So far, LDJT deals with (certain) temporal data. A first step would be to include spatial data, in a similar fashion as we added temporal aspects with PDMs to PMs, as another dimension of PRVs and extend LDJT to also reason over spatial data. Thereby, LDJT could handle certain temporal spatial data. However, there often are uncertainties also in temporal and spatial data. For example an ancient object might be pin pointed to originate from the early first century somewhere in the south of Greece. Hence, there are uncertainties about the exact year and the exact location.

A first approach could be to model the uncertainties with uncertain evidence. Thus, providing the locations as uncertain evidence and observing that the object was created in a some years with uncertain evidence. However, using uncertain evidence might be an over-simplification. Instead of uncertain evidence, distributions over temporal and spatial uncertainty might be the more precise representation. Thus, one would have uncertainties in two dimensions. Such a scenario can be compared to a partially observable Markov decision process (POMDP), where one cannot observe the state location exactly. However, in our scenario, we do not only have uncertainties about the location, but also the time. In general, there already exists work for spatio-temporal event detection (Yin *et al.*, 2009). However, the idea of distributions for temporal and spatial uncertainty induces exciting new research opportunities.

Bibliography

Babak Ahmadi, Kristian Kersting, Martin Mladenov, and Sriraam Natarajan. Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training. *Machine learning*, 92(1):91–132, 2013.

Udi Apsel and Ronen I. Brafman. Extended Lifted Inference with Joint Formulas. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*, pages 11–18. AUAI Press, 2011.

Leonard E Baum, Ted Petrie, George Soules, and Norman Weiss. A Maximization Technique Occurring in the Statistical Analysis of Probabilistic Functions of Markov Chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

Daniel Benjamin, James Berger, Magnus Johannesson, Brian Nosek, E.-J Wagenmakers, Richard Berk, Kenneth Bollen, Björn Brembs, Lawrence Brown, Colin Camerer, David Cesarini, Christopher Chambers, Merlise Clyde, Thomas Cook, Paul De Boeck, Zoltan Dienes, Anna Dreber, Kenny Easwaran, Charles Efferson, and Valen Johnson. Redefine Statistical Significance. *Nature Human Behaviour*, 2(1):6, 2018.

Xavier Boyen and Daphne Koller. Tractable Inference for Complex Stochastic Processes. In *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 33–42. Morgan Kaufmann Publishers Inc., 1998.

Tanya Braun and Marcel Gehrke. Inference in Statistical Relational AI. In *Proceedings of the 24th International Conference on Conceptual Structures*, pages xvii–xix. Springer, 2019.

Tanya Braun and Ralf Möller. Lifted Junction Tree Algorithm. In *Proceedings of KI 2016: Advances in Artificial Intelligence*, pages 30–42. Springer, 2016.

Tanya Braun and Ralf Möller. Preventing Groundings and Handling Evidence in the Lifted Junction Tree Algorithm. In *Proceedings of KI 2017: Advances in Artificial Intelligence*, pages 85–98. Springer, 2017.

Tanya Braun and Ralf Möller. Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm. In *Postproceedings of the 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning*. Springer, 2018.

Tanya Braun and Ralf Möller. Lifted Most Probable Explanation. In *Proceedings of the 23rd International Conference on Conceptual Structures*, pages 39–54. Springer, 2018.

Tanya Braun and Ralf Möller. Parameterised Queries and Lifted Query Answering. In *IJCAI-18 Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 4980–4986. International Joint Conferences on Artificial Intelligence Organization, 2018.

Tanya Braun. *Rescued from a Sea of Queries: Exact Inference in Probabilistic Relational Models*. PhD thesis, 2020.

Rodrigo De Salvo Braz, Eyal Amir, and Dan Roth. Lifted First-order Probabilistic Inference. In *IJCAI05 Proceedings of the 19th International Joint Conference on Artificial intelligence*, pages 1319–1325. Morgan Kaufmann Publishers Inc., 2005.

Hei Chan and Adnan Darwiche. On the Revision of Probabilistic Beliefs using Uncertain Evidence. *Artificial Intelligence*, 163(1):67 – 90, 2005.

Jan Chomicki and Tomasz Imieliński. Temporal Deductive Databases and Infinite Objects. In *Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 61–73. ACM, 1988.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

Rodrigo de Salvo Braz. *Lifted First-Order Probabilistic Inference*. PhD thesis, Ph. D. Dissertation, University of Illinois at Urbana Champaign, 2007.

Anton Dignös, Michael H Böhlen, and Johann Gamper. Temporal Alignment. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 433–444. ACM, 2012.

Maximilian Dylla, Iris Miliaraki, and Martin Theobald. A Temporal-Probabilistic Database Model for Information Extraction. *Proceedings of the VLDB Endowment*, 6(14):1810–1821, 2013.

Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD'96 Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.

R.A. Fisher. *Statistical Methods for Research Workers*. Edinburgh Oliver & Boyd, 1925.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Answering Hindsight Queries with Lifted Dynamic Junction Trees. In *8th International Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence*, 2018.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Answering Multiple Conjunctive Queries with the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the AI 2018: Advances in Artificial Intelligence*, pages 543–555. Springer, 2018.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the 23rd International Conference on Conceptual Structures*, pages 55–69. Springer, 2018.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *8th International Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence*, 2018.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the AI 2018: Advances in Artificial Intelligence*, pages 556–562. Springer, 2018.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Towards Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of KI 2018: Advances in Artificial Intelligence*, pages 38–45. Springer, 2018.

Marcel Gehrke, Tanya Braun, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhäuser. Towards Lifted Maximum Expected Utility. In *Proceedings of the Joint Workshop on Artificial Intelligence in Health in Conjunction with the 27th IJCAI, the 23rd ECAI, the 17th AAMAS, and the 35th ICML*, pages 93–96. CEUR-WS.org, 2018.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Temporal Maximum Expected Utility. In *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*, pages 380–386. Springer, 2019.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Temporal Most Probable Explanation. In *Proceedings of the 24th International Conference on Conceptual Structures*, pages 72–85. Springer, 2019.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Relational Forward Backward Algorithm for Multiple Queries. In *Proceedings of the 32nd International Florida Artificial Intelligence Research Society Conference (FLAIRS-32)*, pages 464–469. AAAI Press, 2019.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Uncertain Evidence for Probabilistic Relational Models. In *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*, pages 80–93. Springer, 2019.

Marcel Gehrke, Tanya Braun, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhäuser. Lifted Maximum Expected Utility. In *Proceedings of Artificial Intelligence in Health*, pages 131–141. Springer International Publishing, 2019.

Marcel Gehrke, Ralf Möller, and Tanya Braun. Taming Reasoning in Temporal Probabilistic Relational Models. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, pages 2592–2599, 2020.

Thomas Geier and Susanne Biundo. Approximate Online Inference for Dynamic Markov Logic Networks. In *Proceedings of the 23rd IEEE International Conference on Tools with Artificial Intelligence*, pages 764–768. IEEE, 2011.

Richard C Jeffrey. *The Logic of Decision*. University of Chicago Press, 1990.

Saket Joshi, Kristian Kersting, and Roni Khardon. Generalized First Order Decision Diagrams for First Order Markov Decision Processes. In *IJCAI09 Proceedings of the 21st International Joint Conference on Artifical Intelligence*, pages 1916–1921. Morgan Kaufmann Publishers Inc., 2009.

Solomon Kullback and Richard A Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, 1951.

Steffen L. Lauritzen and David J Spiegelhalter. Local Computations with Probabilities on Graphical Structures and their Application to Expert Systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(2):157–224, 1988.

Cristina E Manfredotti. *Modeling and Inference with Relational Dynamic Bayesian Networks*. PhD thesis, Ph. D. Dissertation, University of Milano-Bicocca, 2009.

Brian Milch, Luke S. Zettlemoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted Probabilistic Inference with Counting Formulas. In *AAAI08 Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, pages 1062–1068. AAAI Press, 2008.

Kevin Murphy and Yair Weiss. The Factored Frontier Algorithm for Approximate Inference in DBNs. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pages 378–385. Morgan Kaufmann Publishers Inc., 2001.

Kevin Patrick Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, University of California, Berkeley, 2002.

Aniruddh Nath and Pedro Domingos. A language for relational decision theory. In *International Workshop on Statistical Relational Learning*, 2009.

Aniruddh Nath and Pedro Domingos. Efficient Lifting for Online Probabilistic Inference. In *Proceedings of the 6th AAAI Conference on Statistical Relational Artificial Intelligence, AAAIWS'10-06*, pages 1193–1198. AAAI Press, 2010.

Aniruddh Nath and Pedro M Domingos. Efficient Belief Propagation for Utility Maximization and Repeated Inference. In *AAAI10 Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 1187–1192. AAAI Press, 2010.

Mathias Niepert and Guy Van den Broeck. Tractability through Exchangeability: A New Perspective on Efficient Probabilistic Inference. In *AAAI14 Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2467–2475. AAAI Press, 2014.

Davide Nitti, Tinne De Laet, and Luc De Raedt. A Particle Filter for Hybrid Relational Domains. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2764–2771. IEEE, 2013.

Özgür Özcep, Ralf Möller, and Christian Neuenstadt. Stream-Query Compilation with Ontologies. In *Proceedings of the AI 2015: Advances in Artificial Intelligence*. Springer, 2015.

Tivadar Papai, Henry Kautz, and Daniel Stefankovic. Slice Normalized Dynamic Markov Logic Networks. In *NIPS12 Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, pages 1907–1915. Curran Associates Inc., 2012.

Judea Pearl. On Two Pseudo-Paradoxes in Bayesian Analysis. *Annals of Mathematics and Artificial Intelligence*, 32(1-4):171–177, 2001.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Reasoning*. Elsevier, 2014.

Yun Peng, Shenyong Zhang, and Rong Pan. Bayesian Network Reasoning with Uncertain Evidences. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 18(05):539–564, 2010.

David Poole. First-order probabilistic inference. In *IJCAI03 Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 985–991. Morgan Kaufmann Publishers Inc., 2003.

Carl Edward Rasmussen. Gaussian Processes in Machine Learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

Matthew Richardson and Pedro Domingos. Markov Logic Networks. *Machine learning*, 62(1):107–136, 2006.

Stuart J Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 1995.

Scott Sanner and Craig Boutilier. Approximate Solution Techniques for Factored First-order MDPs. In *17th International Conference on Automated Planning and Scheduling*, page 288–295. AAAI Press, 2007.

Scott Sanner and Kristian Kersting. Symbolic Dynamic Programming for First-order POMDPs. In *AAAI10 Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 1140–1146. AAAI Press, 2010.

Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transactions on Database Systems (TODS)*, 42(3):19, 2017.

Vishal Sharma, Noman Ahmed Sheikh, Happy Mittal, Vibhav Gogate, and Parag Singla. Lifted Marginal MAP Inference. In *Proceedings of the 34th Conference on Uncertainty in Artificial Intelligence*, pages 917–926. AUAI Press, 2018.

Parag Singla, Aniruddh Nath, and Pedro M Domingos. Approximate Lifting Techniques for Belief Propagation. In *AAAI14 Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2497–2504. AAAI Press, 2014.

Jost Steinhäuser and Thomas Kühlein. Role of the General Practitioner. In *Patient Blood Management*, pages 61–65. Thieme, 2015.

Nima Taghipour, Jesse Davis, and Hendrik Blockeel. First-order Decomposition Trees. In *NIPS13 Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 1*, pages 1052–1060. Curran Associates Inc., 2013.

Nima Taghipour, Jesse Davis, and Hendrik Blockeel. Generalized Counting for Lifted Variable Elimination. In *International Conference on Inductive Logic Programming*, pages 107–122. Springer, 2013.

Nima Taghipour, Daan Fierens, Jesse Davis, and Hendrik Blockeel. Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. *Journal of Artificial Intelligence Research*, 47(1):393–439, 2013.

Nima Taghipour, Daan Fierens, Guy Van den Broeck, Jesse Davis, and Hendrik Blockeel. Completeness Results for Lifted Variable Elimination. In *Artificial Intelligence and Statistics*, pages 572–580, 2013.

Nima Taghipour. *Lifted Probabilistic Inference by Variable Elimination*. PhD thesis, Ph. D. Dissertation, KU Leuven, 2013.

Matthias Thimm and Gabriele Kern-Isberner. On Probabilistic Inference in Relational Conditional Logics. *Logic Journal of IGPL*, 20(5):872–908, 2012.

Ingo Thon, Niels Landwehr, and Luc De Raedt. Stochastic relational processes: Efficient inference and applications. *Machine Learning*, 82(2):239–272, 2011.

Guy Van den Broeck and Adnan Darwiche. On the Complexity and Approximation of Binary Evidence in Lifted Inference. In *NIPS13 Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, pages 2868–2876. Curran Associates Inc., 2013.

Guy Van den Broeck and Jesse Davis. Conditioning in First-order Knowledge Compilation and Lifted Probabilistic Inference. In *AAAI12 Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 1961–1967. AAAI Press, 2012.

Guy Van den Broeck and Mathias Niepert. Lifted Probabilistic Inference for Asymmetric Graphical Models. In *AAAI15 Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 3599–3605. AAAI Press, 2015.

Guy Van den Broeck, Ingo Thon, Martijn van Otterlo, and Luc De Raedt. DTPROBLOG: A Decision-theoretic Probabilistic Prolog. In *AAAI10 Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 1217–1222. AAAI Press, 2010.

Guy Van den Broeck. On the Completeness of First-Order Knowledge Compilation for Lifted Probabilistic Inference. In *NIPS11 Proceedings of the 24th International Conference on Neural Information Processing Systems*, pages 1386–1394. Curran Associates Inc., 2011.

Guy Van den Broeck. *Lifted Inference and Learning in Statistical Relational Models*. PhD thesis, KU Leuven, 2013.

Deepak Venugopal and Vibhav Gogate. Evidence-Based Clustering for Scalable Inference in Markov Logic. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 258–273. Springer, 2014.

Jonas Vlasselaer, Wannes Meert, Guy Van den Broeck, and Luc De Raedt. Efficient Probabilistic Inference for Dynamic Relational Models. In *Proceedings of the 13th AAAI Conference on Statistical Relational AI*, AAAIWS'14-13, pages 131–132. AAAI Press, 2014.

Jonas Vlasselaer, Guy Van den Broeck, Angelika Kimmig, Wannes Meert, and Luc De Raedt. TP-Compilation for Inference in Probabilistic Logic Programs. *International Journal of Approximate Reasoning*, 78:15–32, 2016.

YH Wang, Kening Cao, and XM Zhang. Complex Event Processing over Distributed Probabilistic Event Streams. *Computers & Mathematics with Applications*, 66(10):1808–1821, 2013.

Bastian Wemmenhove, Joris M Mooij, Wim Wiegerinck, Martijn Leisink, Hilbert J Kappen, and Jan P Neijt. Inference in the Promedas Medical Expert System. In *Conference on Artificial Intelligence in Medicine in Europe*, pages 456–460. Springer, 2007.

Jie Yin, Derek Hao Hu, and Qiang Yang. Spatio-Temporal Event Detection using Dynamic Conditional Random Fields. In *IJCAI09 Proceedings of the 21st International Jont Conference on Artifical Intelligence*, pages 1321–1326. Morgan Kaufmann Publishers Inc., 2009.

Publications

Conference Papers

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the 23rd International Conference on Conceptual Structures*, pages 55–69. Springer, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Towards Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of KI 2018: Advances in Artificial Intelligence*, pages 38–45. Springer, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the AI 2018: Advances in Artificial Intelligence*, pages 556–562. Springer, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Answering Multiple Conjunctive Queries with the Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the AI 2018: Advances in Artificial Intelligence*, pages 543–555. Springer, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Relational Forward Backward Algorithm for Multiple Queries. In *Proceedings of the 32nd International Florida Artificial Intelligence Research Society Conference (FLAIRS-32)*, pages 464–469. AAAI Press, 2019

Marcel Gehrke, Tanya Braun, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhäuser. Lifted Maximum Expected Utility. In *Proceedings of Artificial Intelligence in Health*, pages 131–141. Springer International Publishing, 2019

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Temporal Maximum Expected Utility. In *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*, pages 380–386. Springer, 2019

Marcel Gehrke, Tanya Braun, and Ralf Möller. Uncertain Evidence for Probabilistic Relational Models. In *Proceedings of the 32nd Canadian Conference on Artificial Intelligence, Canadian AI 2019*, pages 80–93. Springer, 2019

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Temporal Most Probable Explanation. In *Proceedings of the 24th International Conference on Conceptual Structures*, pages 72–85. Springer, 2019

Marcel Gehrke, Ralf Möller, and Tanya Braun. Taming Reasoning in Temporal Probabilistic Relational Models. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, pages 2592–2599, 2020

Workshop Papers

Marcel Gehrke, Tanya Braun, and Ralf Möller. Preventing Unnecessary Groundings in the Lifted Dynamic Junction Tree Algorithm. In *8th International Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence*, 2018

Marcel Gehrke, Tanya Braun, and Ralf Möller. Answering Hindsight Queries with Lifted Dynamic Junction Trees. In *8th International Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence*, 2018

Marcel Gehrke, Tanya Braun, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhäuser. Towards Lifted Maximum Expected Utility. In *Proceedings of the Joint Workshop on Artificial Intelligence in Health in Conjunction with the 27th IJCAI, the 23rd ECAI, the 17th AAMAS, and the 35th ICML*, pages 93–96. CEUR-WS.org, 2018

Extended Abstracts

Tanya Braun and Marcel Gehrke. Inference in Statistical Relational AI. In *Proceedings of the 24th International Conference on Conceptual Structures*, pages xvii–xix. Springer, 2019