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

Rescued from a Sea of Queries:

Exact Inference in 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

Tanya Braun
from Hamburg

Lübeck 2019

First referee Prof. Dr. rer. nat. habil. Ralf Möller
Second referee Prof. Dr. rer. nat. Kristian Kersting
Date of oral examination February 21, 2020
Approved for printing. Lübeck

Abstract

At the heart of many machine learning (ML) algorithms lie large probabilistic models that use random variables to describe behaviour or structure hidden in data. Often, a model is shaped by a pool of known individuals (objects, constants) and relations between them, leading to probabilistic relational models. In probabilistic relational models, an ocean of queries is possible, asking for the probability of specific events, a most probable explanation, or a marginal distribution of a set of interchangeable randvars. Exact inference, in contrast to approximate inference, allows for attributing errors in a result to the model itself as the result is exact. Therefore, the problem studied in this dissertation is the problem of *exact repeated inference*, specifically, solving multiple instances of different query answering problems in probabilistic relational models.

To solve the problem, we combine the lifting idea with junction trees, also known as clique trees or join trees, a well studied data structure representing clusters of randvars in a model. Junction trees allow for efficient repeated inference. Lifting efficiently handles sets of known objects by working with representatives of objects behaving identically and only looking at specific objects if necessary. We categorise the contributions into three parts. First, we present the lifted junction tree algorithm, which includes lifting of junction trees to first-order junction trees and answering queries without inducing additional groundings. For calculations on first-order junction trees, we incorporate a version of lifted variable elimination to exploit lifting not only for the tree structure but also the calculations. Second, we extend the query language of this first version of the lifted junction tree algorithm. We introduce parameterised queries as a new construct of queries, which applies lifting to queries as well. We transform lifted variable elimination into an algorithm to compute most probable assignments. Third, we concentrate on the lifted junction tree algorithm itself: We provide a version for adaptive inference, which includes adapting a first-order junction tree to incremental changes in a model. We also dive into the lifted junction tree algorithm as a framework, investigating requirements for other inference algorithms to become a subroutine like lifted variable elimination. We end this dissertation by looking into unknown universes.

The algorithm versions presented in this dissertation each receive an empirical evaluation, testing how they react under growing numbers of objects and random variables. Additionally, we provide extensive theoretical analyses, investigating the connection of the lifted junction tree algorithm with lifted variable elimination and the propositional junction tree algorithm as well as investigating the connection between computing distributions versus most probable assignments in probabilistic relational models.

Kurzfassung

Im Herzen von vielen Algorithmen des maschinellen Lernens liegen große probabilistische Modelle, in denen mittels Zufallsvariablen ein bestimmtes Modellverhalten oder latente Strukturen in Daten beschrieben werden. Ein Modell wird dabei oft durch eine Menge von bekannten Individuen (Objekten, Konstanten) und Relationen zwischen Individuen charakterisiert, was zu probabilistischen relationalen Modellen führt. In probabilistischen relationalen Modellen ist ein Meer von Anfragen möglich, von Anfragen an Wahrscheinlichkeiten von bestimmten Ereignissen, über Anfragen an wahrscheinlichste Erklärungen, bis hin zu Anfragen an Marginalverteilungen von austauschbaren Zufallsvariablen. Exakte Inferenz, im Gegensatz zu approximativer Inferenz, ermöglicht es Abweichungen zwischen Inferenzergebnis und Realität dem Modell selbst zuzuschreiben. Deshalb befasst sich diese Dissertation mit dem Problem der *exakten mehrfachen Inferenz*. Genauer gesagt geht es darum, in probabilistischen relationalen Modellen wiederholt Instanzen von unterschiedlichen Problemen der Anfragebeantwortung zu lösen.

Um das Problem zu lösen, kombinieren wir das so genannte Lifting mit Baumzerlegungen von Graphen, die probabilistische relationale Modelle darstellen. Baumzerlegungen selbst bilden eine Baumstruktur (im englischen *junction tree*, *clique tree* oder *join tree*), in der Knoten eine Menge von Zufallsvariablen sind, die wiederum Cliquen im probabilistischen relationalen Modell bilden. Diese Baumstruktur ermöglicht effiziente mehrfache Inferenz. Lifting beschreibt die Idee, mit Repräsentanten für Objekte zu arbeiten, die sich gleich verhalten. Einzelne Objekte werden nur, wenn notwendig, angeguckt. Wir unterteilen die Beiträge dieser Dissertation in drei Teile. Als erstes präsentieren wir den Lifted Junction Tree Algorithmus, was das Lifting von Baumzerlegungen beinhaltet und die Anfragebeantwortung auf diesen Baumstrukturen ohne zusätzliche Instantiierungen. Für die Anfragebeantwortung setzen wir die geliftete Variablenelimination ein, so dass wir Lifting nicht nur bei der Baumzerlegung ausnutzen, sondern auch bei Rechnungen. Als zweites erweitern wir die Anfragesprache des Lifted Junction Tree Algorithmus.Wir stellen parametrisierte Anfragen als ein neues Konstrukt für Anfragen vor, welches Lifting ebenfalls auf Anfragen anwendet, und wir schreiben die geliftete Variablenelimination um, so dass wahrscheinlichste Zuweisungen berechnet werden. Als drittes erweitern wir den Lifted Junction Tree Algorithmus, einmal für adaptive Inferenz inklusive dem Anpassen einer Baumzerlegung an inkrementelle Änderungen in einem Modell und einmal als Rahmenwerk für Inferenzalgorithmen, die die geliftete Variablenelimination als Unterprozedur ersetzen können. We schließen diese Dissertation ab, in dem wir uns unbekannten Universen zuwenden, in denen die Objekte nicht vorher bekannt sind.

Die Algorithmen der Dissertation werden jeweils empirisch untersucht, um zu testen, wie sie auf wachsende Zahlen von Objekten und Zufallsvariablen reagieren. Zusätzlich analysieren wir die Algorithmen bezüglich deren Richtigkeit und Komplexität. Dabei untersuchen wir auch die Verbindung von dem propositionalen Junction Tree Algorithmus und der gelifteten Variablenelimination mit dem Lifted Junction Tree Algorithmus sowie die Verbindung zwischen der Berechnung von Marginalverteilungen und der Berechnung von wahrscheinlichsten Zuweisungen in probabilistischen relationalen Modellen.

Acknowledgements

Writing a PhD thesis is a strange task. It combines a handful of years of work into one large pile of pages, organised in chapters, and at the end, I sit here wringing my hands over the final "chapter" to write, the acknowledgements, which, as it turns out, is the hardest chapter of them all.

Because writing this thesis was less of a daunting task than I had come to believe. Compiling the first complete version took about a month. And that is a compliment to my supervisor, Ralf Möller. He told me to get out some papers, combine them into a journal article, and use that as the foundation for the thesis. And that is what I did. It made writing two journal articles a real pain but in turn allowed for the dissertation to be a piece of cake. At the very beginning, he told me "TEXis your friend". So, I tex'd everything, kept track of new ideas through `.tex` files, and developed algorithms by sketching them with `algpseudocode`. At the end, I often did need to "only 'tex' it" for an idea to become a fully formed paper. (It takes longer than "before breakfast", though.) He let me work out my own ideas, trusted me with technical details, and still guided me by never losing track of longterm goals and strategic steps towards an academic carrer. And sometimes, he would come around the corner with some wild idea or a hunch about something that I had deemed little interesting that lead to papers that I really like. So, thank you, Ralf, for answering my question about if I could cite you as a reference when I was looking for a PhD position with "I can offer you a job here." Thank you for taking on tasks like presenting tutorials together to get my name out there and allowing me to go off to Bamberg to give my first own lecture while I was still organising exercises for your lectures.

Of course, these past few years were not only shaped by a supervisor but also by colleagues and friends and family. To Angela and Nils, thank you for always helping and keeping an eye out. Felix, I am forever grateful for our "early morning" coffee breaks to discuss work, life, and the world. I really enjoy that after trying to work on papers together at the beginning and somehow giving up on it along the way, we have come back to it now and I look forward to writing more papers with you, preferably with the view of Sarasota at our feet. Thank you to the Mannemers (you know who you are) for getting me to walk 10,000 steps a day, for asking the hard questions about ethics and AI, and for always having an ear or two. Thank you to my parents and my brother for trying so valiantly to understand a single thing I am researching and for supporting me

in restarting my career plans. It seems like computer science is the better match for me. Thank you, Marcel, for sticking it out with me through Bachelor and Master's studies, all the jobs at the university, the little free time we had in between, and going along with crazy ideas like "We should totally move in together right when the most craziest times of our studies start." Thank you for being my sounding board during all these years and for not letting us drive ourselves crazy over working on the same research subject. Along the way, we generated some neat papers and saw some really awesome places. I am looking forward to whatever comes next for us.

In hindsight, I am also grateful for musicals. I am now fluent in Dance of the Vampires, Dirty Rotten Scoundrels, Catch Me If You Can, Phantom of the Opera, and Groundhog Day. They have kept me sane while some piece of theory was driving me nuts.

Thank you!

Tanya Braun
Lübeck, Juni 2019

Contents

List of Algorithms

List of Operators

List of Abbreviations

ADD	Algebraic Decision Diagram
aLJT	adaptive LJT
CRV	counting random variable
CNF	conjunctive normal form
d-DNNF	deterministic decomposable negation normal form
DPG	decomposition into partial groundings
dtree	decomposition tree
FG	factor graph
FO d-DNNF	first-order deterministic decomposable negation normal form
FO dtree	first-order decomposition tree
FO jtree	first-order junction tree
FOKC	first-order knowledge compilation
JT	junction tree algorithm
jtree	junction tree
KC	knowledge compilation
LBP	lifted belief propagation
LJT	lifted junction tree algorithm
LJTKC	LJT fused with KC
LVE	lifted variable elimination
logvar	logical variable
MAP	maximum a posteriori

MAP-LJT	LJT for MAP queries
MAP-LVE	LVE for MAP queries
MEU	maximum expected utility
ML	machine learning
MPE	most probable explanation
MPE-LJT	LJT for MPE queries
MPE-LVE	LVE for MPE queries
NNF	negation normal form
PDB	probabilistic database
parfactor	parametric factor
PCRV	parameterised counting random variable
PRV	parameterised random variable
PP	probability propagation
QA	query answering
randvar	random variable
VE	variable elimination
WMC	weighted model count
WFOMC	weighted first-order model count

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
$g, \phi(\mathcal{A})_{\mid C}, \phi(\mathcal{A})$	Parfactor
G	Model
$gr(P)$	Grounding
$rv(P)$	PRVs with constraints
$lv(P)$	Logvars
θ	Substitution, alignment
T	VE node, FO dtree
$T_{\mathbf{X}}$	DPG node
L	Leaf node
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
n_T	Number of nodes in an FO dtree
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
α	Runtime ratio of a compiled versus an uncompiled algorithm
β	Number of queries needed to tradeoff compilation overhead
$\mathbf{Q}$	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
$\mathcal{PCQ}, \mathcal{PCQ}^{lift}$	Class of (liftable) parameterised conjunctive queries
$\mathcal{B}$	Maxed-out parameterised random variables
p	Potential
ϕ^P	Potential in ϕ storing also assignments
ϕ^A	Assignment in ϕ storing also assignments
$\mathcal{MAP}, \mathcal{MAP}^{lift}$	Class of (liftable) maximum a posteriori assignment queries
Δ_G	Changes in G
$\Delta_\mathbf{E}$	Changes in $\mathbf{E}$
$\mathcal{C}$	Circuit
Δ	Theory of constrained clauses
w_T	Positive weight function
w_F	Negative weight function

Chapter 1

Introduction

At the heart of many machine learning (ML) algorithms lie vast amounts of data that are condensed into large probabilistic models that use random variables (randvars) to describe behaviour or structure hidden in data. After a surge in effective ML algorithms over the last decade, efficient algorithms for inference come into focus to make use of the models learned or to optimise ML algorithms further (LeCun, 2018). Often, a model is shaped by a pool of known individuals (objects, constants) and relations between them. Thus, handling sets of individuals efficiently enables algorithms to provide inference that no longer has a runtime complexity exponential in the number of individuals (Niepert and Van den Broeck, 2014). Lifting efficiently handles sets of individuals by working with representatives of individuals behaving identically and only looking at specific individuals if necessary. Lifting has emerged from the observation that symmetries in a probabilistic graphical model, i.e., a probabilistic model with a graphical structure, enable treating indistinguishable individuals as a group, while still answering probability queries exactly. Exact inference, in contrast to approximate inference, allows for attributing errors in an inference result to the model itself as the result is exact. Exact inference is paramount in, e.g., decision support for health care (Wemmenhove *et al.*, 2007). Additionally, one instance of an inference problem seldom appears by itself but rather in company with many other instances. Repeated inference aims at solving a set of problem instances efficiently. The problem that this dissertation focusses on is that of

exact repeated inference,
specifically, solving multiple instances of the query answering (QA) problem,
in *probabilistic relational models*.

In QA algorithms, queries typically concern a marginal (conditional) distribution, often with only a single randvar in the margin, or a most probable explanation (MPE), which is the most probable assignment to all model randvars. In contrast, the goal of this dissertation was to design an algorithm for exact repeated inference of *various types of queries*, ranging from a probability of a set of randvars (conjunctive query) to the most probable assignment of a subset of model randvars (maximum a posteriori, MAP).

1.1 Related Work

In the last three decades and counting, researchers have sped up runtimes for inference significantly, first working on propositional inference and then incorporating lifting into various algorithms and modelling frameworks.

Propositional formalisms benefit from variable elimination (VE, Zhang and Poole, 1994). VE decomposes a model into subproblems to evaluate them in an efficient order. A decomposition tree (dtree) represents such a decomposition (Darwiche, 2001). For repeated inference in a propositional setting, Lauritzen and Spiegelhalter (1988) introduce junction trees (jtrees), a representation of clusters in a model, along with an algorithm for solving QA problems (junction tree algorithm, JT). A cluster is a subset sufficient for solving a QA problem without considering the original model for each QA problem. The remaining model contains factorial parts important for QA in the cluster. Therefore, the outside factors need to be projected into each cluster by recursively querying for outside randvars. JT implements answering these recursive queries efficiently using dynamic programming principles, which is called message passing, also known as probability propagation (PP), and then JT answers queries on clusters. Shafer and Shenoy (1990) and Jensen *et al.* (1990) provide well known PP schemes based on VE that trade off runtime and storage differently. Darwiche (2001) demonstrates a connection between jtrees and VE, namely, a dtree representing the decomposition during VE allows for building a jtree.

Lifted inference as a fundamental inference technique has sparked further progress. Lifted VE (LVE), first introduced by Poole (2003) and expanded by de Salvo Braz *et al.* (2005, 2006), saves computations by reusing intermediate results for isomorphic subproblems (lifted summing out). Milch *et al.* (2008) introduce counting to lift certain computations where lifted summing out does not apply. Taghipour *et al.* (2013b) as well as Apsel and Brafman (2011) refine counting to lift even more cases. Taghipour *et al.* (2013c) extend the formalism to its current standard by decoupling the algorithm from the formalism for specifying individuals. LVE focusses on solving one query, leading to inefficiencies given a set of queries. Additionally, LVE is specified for single query terms.

For repeated inference in probabilistic relational models, Van den Broeck *et al.* (2011) apply lifting to knowledge compilation (KC) and weighted model counting (WMC), introducing first-order KC (FOKC). FOKC compiles a model into a so called circuit and uses a lifted version of WMC to answer queries for probabilities of events and marginal distributions of single randvars. While FOKC computes probabilities in runtime linear w.r.t. its circuit size, its compilation process may lead to circuit sizes exponential w.r.t. an input model. Van den Broeck and Niepert (2015) approximate symmetries in asymmetrical models. Friedman and Van den Broeck (2018) go in the direction of approximate inference using approximate KC. Lifted belief propagation (LBP) combines PP and lifting for an approximate solution (Jaimovich *et al.*, 2007; Singla and Domingos, 2008; Kersting *et al.*, 2009; Gogate and Domingos, 2010), which may become arbitrar-

ily bad. Ahmadi *et al.* (2013) present counting LBP that also runs a so called colour passing algorithm to build a lifted representation out of a grounded model. Kersting *et al.* (2017) transfer lifting to linear programs, viewing a linear program as a coloured graph and applying colour passing to compress the graph. In fact, there are a plethora of lifted approximate QA algorithms, from LBP over lifted variational inference (Choi and Amir, 2012) to lifted sampling using importance sampling (Gogate and Domingos, 2011) or Markov Chain Monte Carlo (Niepert, 2012, 2013).

To scale lifting, Das *et al.* (2016) use graph databases storing compiled models for faster counting. Other methods incorporate lifting for gaining efficiency, e.g, in continuous models (Choi *et al.*, 2010) or in dynamic models (Ahmadi *et al.*, 2013; Gehrke *et al.*, 2019c), methods for the latter are based on work by Murphy (2002). Other formalism incorporate lifting as well such as logic programming (Bellodi *et al.*, 2014) and probabilistic theorem proving (Gogate and Domingos, 2011), which is also based on WMC. Lifting also applies for solving a maximum expected utility (MEU) problem in a probabilistic relational model, with Nath and Domingos (2010a) presenting an approximate approach. Apsel and Brafman (2012b) solve the MEU problem based on an early LVE version. More recently, Gehrke *et al.* (2019e) present an MEU version of LVE and transfer the result to dynamic models (Gehrke *et al.*, 2019a).

For an analysis of lifting, Taghipour *et al.* (2013a) lift dtrees to the first-order setting, incorporating that lifted summing out handles isomorphic instances. They introduce first-order dtrees (FO dtrees), which form the basis for analysing the complexity of LVE and permit determining in advance if lifted inference is possible. Van den Broeck (2011) provides completeness results for FOKC, introducing the notion of domain liftability. Completeness refers to an algorithm being able to do lifted inference for any possible model of some model class. Jaeger and Van den Broeck (2012) present upper and lower complexity bounds, while Taghipour *et al.* (2013d) present completeness results for LVE.

Only a handful of approaches focus on repeated inference in probabilistic relational models and they have their own drawbacks, e.g., by being approximate or by an exponential size of circuits. Therefore, we set out to design an algorithm for solving the problem of exact repeated inference differently, introducing the lifted junction tree algorithm (LJT) (Braun and Möller, 2016). LJT has its foundations in JT and lifting, compiling a model into a first-order jtree (FO jtree) in contrast to FOKC compiling a model into a circuit. We have adapted LJT and LVE to support lifted QA for conjunctive queries, parameterised queries, and MPE queries (Braun and Möller, 2018b,e,f). Additionally, we have worked on adaptive inference (Braun and Möller, 2018a) to efficiently handle incremental changes in a model or in observations. We have analysed LJT as a framework, allowing for other QA algorithms to take the role of LVE, and fused LJT with LVE and FOKC (Braun and Möller, 2018c). Together, these works provide a comprehensive investigation of an algorithm for solving the QA problem repeatedly in probabilistic relational models faced with different types of queries. Now, this research culminates in the dissertation at hand.

1.2 Contributions

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

(1) Lifting of jtrees Repeated patterns in a propositional probabilistic model, or a grounded version of a relational model, lead to duplicate nodes in a jtree. Lifting a jtree allows for compactly representing "duplicate" nodes, allowing QA algorithms to use relations explicitly represented in an FO jtree for efficient QA. This contribution comprises **(a)** a definition of FO jtrees along with parameterised clusters (parclusters) as its nodes, **(b)** an algorithm for building an FO jtree from an FO dtree, including a proof that, also in the lifted case, the clusters of an FO dtree form an FO jtree, and **(c)** a merging procedure to minimise an FO jtree.

(2) Lifted QA on FO jtrees incorporating LVE Given an FO jtree, we transfer JT to the lifted setting, exploiting relational structures for efficient exact, repeated inference. Like JT, LJT still consists of the steps construction, evidence entering, message passing, and query answering. But given the FO nature of the underlying model, messages may cause groundings and thus, require additional work, called fusion, after construction to ensure an FO jtree does not cause groundings during message calculation.

(3) Completeness and complexity results for LJT We show that the completeness results of LVE also hold for LJT. We also analyse the complexity of LJT, demonstrating that the connection existing between VE and JT also exists between LVE and LJT and that lifting JT to LJT mirrors the effect of lifting VE to LVE.

(4a) Lifted QA for conjunctive queries on FO jtrees On our way to supporting more complex queries, we extend the query answering step of LJT to handle conjunctive queries, in which a single query may contain a set of query terms. The challenge for LJT arises when the query terms do not appear in a single parcluster.

(4b) Lifted QA by lifting of queries A query with interchangeable query terms leads to a blowup in the query, the computational effort, and the result representation. Presenting lifting for queries, we allow for parameters in query terms to compactly encode interchangeable query terms. Using parameters in queries enables LVE to eliminate non-query terms, avoiding grounding on query terms. Benefitting from work on counting, the result can also be encoded in a more compact way.

(4c) Combined completeness and complexity results for complex queries We provide results for LVE and LJT given more complex queries, which influence whether groundings

occur. Thus, given more complex queries, completeness of LVE and LJT also depends on the class of queries.

(5a) Redefined LVE operators for solving MPE queries Previous work on solving MPE queries in a lifted way rely on very early LVE versions. With this contribution, we redefine the LVE operator suite by Taghipour *et al.* (2013c) to solve MPE problems.

(5b) LVE and LJT versions for MAP queries MAP queries are even harder to answer than MPE queries as MAP queries involve a set of randvars that have to be summed out and a set of randvars that have to be maxed out afterwards with max-out and sum-out operations not being commutative. Thus, MAP queries impose an elimination order that may result in intermediate results much larger than during answering MPE queries. We specify lifted algorithms based on LVE and LJT for MAP queries.

(5c) Completeness and complexity results for LVE and LJT versions answering MPE and MAP queries We analyse LVE and LJT for MPE and MAP w.r.t. completeness, drawing parallels to LVE and LJT for answering queries for probability distributions. Given an FO jtree, we are able to characterise MAP queries that do not lead to larger intermediate results than MPE queries.

(6) Adaptive inference on FO jtrees Adaptive inference aims at answering queries more efficiently than starting from scratch after incremental changes in either model or evidence. Incremental changes in evidence allow LJT to save parts of its message passing. Incremental changes in a model are less obvious to handle by LJT as changes may influence an underlying FO jtree. This contribution consists of **(a)** an algorithm for adjusting an FO jtree to incremental changes in a model and **(b)** adaptive LJT (aLJT) to fast reach the point of answering queries again after changes.

(7) LJT as a framework for QA algorithms LJT uses LVE as a subroutine for calculations regarding messages and queries. Therefore, it is possible to replace LVE with another QA algorithm as long as the QA algorithm supports the queries needed for message and query calculations. This contribution contains **(a)** conditions for QA algorithms to function as a subroutine within LJT, **(b)** a discussion of LVE as a subroutine as well as FOKC as a candidate, and **(c)** a fused version of LJT with LVE and FOKC, using LVE for messages and FOKC for queries.

1.3 Structure

After this introduction, we start with an introductory example, highlighting the potential of lifting and jtrees as well as the different query types we cover. We follow the example

with a chapter on preliminaries, defining parameterised models as the representation formalism based on Poole (2003); Taghipour *et al.* (2013c) and recapping LVE as defined by Taghipour *et al.* (2013c) as a QA algorithm for singular inference. We also introduce FO dtrees (Taghipour *et al.*, 2013a) as a means to build FO jtrees. We end the preliminaries with a brief excursion into FOKC.

After the preliminaries, the main body of this dissertation begins, which is divided into three parts, presenting the contributions of this dissertation.

- Part I presents LJT for repeated inference in probabilistic relational models.
  - Chapter 4 presents LJT itself along with FO jtrees (Contributions **1**, **2**).
  - Chapter 5 presents a theoretical analysis of LJT, looking at soundness, completeness, and complexity (Contribution **3**).
  - Chapter 6 presents an empirical evaluation in five parts.

  This first part was mainly published in

  > 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 - Extended. In *Postproceedings of the 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning at the 26th International Joint Conference on Artificial Intelligence*, pages 54–72. Springer, 2018
  >
  > Tanya Braun and Ralf Möller. Avoiding Repetition in Repeated Inference on Probabilistic Relational Models: The Lifted Junction Tree Algorithm. Submitted

  LJT in this form answers queries with single query terms and a set of observations.

- Part II contains extensions to the query language.
  - Chapter 8 presents LJT for conjunctive queries, allowing for a set of query terms in a single query (Contribution **4a**).
  - Chapter 9 presents parameterised queries as a new construct for queries with interchangeable query terms (Contribution **4b**).
  - Chapter 10 presents LVE and LJT versions for MPE or MAP queries, redefining LVE operators for maxing out randvars (Contributions **5a**, **5b**). The chapter also contains an overview of related work on MPE and MAP queries.

Each chapter contains a theoretical analysis (Contributions **4c** and **5c**) as well as an empirical evaluation. The second part is based on the following conference papers

> Tanya Braun and Ralf Möller. Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm - Extended. In *Postproceedings of the 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning at the 26th International Joint Conference on Artificial Intelligence*, pages 54–72. 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. IJCAI Organization, 2018
>
> Tanya Braun and Ralf Möller. Lifted Most Probable Explanation. In *Proceedings of the International Conference on Conceptual Structures*, pages 39–54. Springer, 2018

- Part III presents further extensions of LJT.
  - Chapter 11 presents LJT for adaptive inference, including an overview of related work on adaptive inference (Contribution **6**). This chapter draws upon the following publication, but is extended with proofs regarding adapting an FO jtree.

    > Tanya Braun and Ralf Möller. Adaptive Inference on Probabilistic Relational Models. In *Proceedings of AI 2018: Advances in Artificial Intelligence*, pages 487–500. Springer, 2018

  - Chapter 12 presents LJT as a framework and a fused version of LJT, LVE, and FOKC, based on the following works (Contribution **7**). The second paper, published at a conference, is a slightly shorter version of first one, published at a workshop.

    > Tanya Braun and Ralf Möller. Fusing First-order Knowledge Compilation and the Lifted Junction Tree Algorithm. In *8th International Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence*, 2018
    >
    > Tanya Braun and Ralf Möller. Fusing First-order Knowledge Compilation and the Lifted Junction Tree Algorithm. In *Proceedings of KI 2018: Advances in Artificial Intelligence*, pages 24–37. Springer, 2018

The first part and all subsequent chapters each end with a brief interim conclusion including future work concentrating on the topic at hand. Chapter 13 takes a look into what the future might hold for lifted inference, swarming into unknown universes. Chapter 14 presents conclusions and ends with some more broad future work.

Chapter 2

Repeated Inference by Example

Before we head into the midst of lifted inference, we take a step back and look at a propositional model. With the help of this model, we illustrate repeated inference with JT and show how lifting has potential for enhancing JT.

2.1 An Epidemic Example

Based on an example by de Salvo Braz *et al.* (2005) about sick people and an epidemic, we model the interplay of natural disasters, an epidemic, and people being sick. If a natural disaster occurs, an epidemic outbreak is more likely, which makes it more likely to get sick. A randvar *Epid* indicates whether an epidemic occurs. We consider M natural disasters (e.g., earthquake, flood) and N people. For each disaster j and each person i, we introduce a randvar Nat_j indicating whether j occurs and a randvar $Sick_i$ indicating whether i is sick. Each randvar has a boolean range, i.e., $\mathcal{R} : Vars \to \{true, false\}$, where $Vars$ refers to the randvars.

Given a set of randvars, a world describes a particular state where each randvar has some range value assigned. Following the idea of distribution semantics (Sato, 1995), a joint probability distribution emerges by assigning all possible worlds a value, i.e., a positive, real number, and normalising the values s.t. the sum is 1. A factorisation breaks a joint distribution down into factors for a sparse encoding. Factors explicitly represent influences between randvars. Hammersley and Clifford (1971) show in an unpublished paper the equivalence between a positive full joint distribution and its factorisation, with Besag (1974) providing more general proofs, superseding the unpublished work as noted by Clifford in the discussion of Besag's article (1974).

For the epidemic example, the set of factors contains for each j a factor $\phi_1(Epid, Nat_j)$ modelling how a natural disaster impacts an epidemic outbreak. For each i, the set contains a factor $\phi_2(Epid, Sick_i)$ expressing how an epidemic influences a person being sick. ϕ_2 is identical for all N persons, which implies that an epidemic affects all persons in exactly the same way. The same holds for all M disasters causing an epidemic. The factors specify a model which is shown in Fig. 2.1 as a factor graph (FG). Variable nodes (ellipses) correspond to randvars, factor nodes (boxes) to factors. An edge goes from a variable node to a factor node if the depicted factor contains the randvar. A model defines

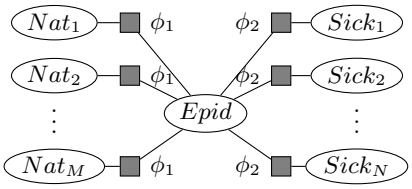


Figure 2.1: An example FG

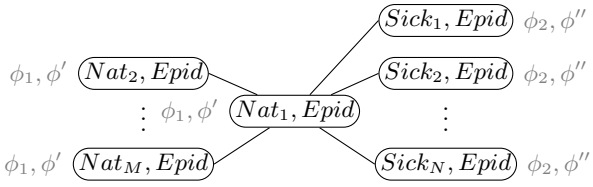


Figure 2.2: An example jtree

a joint distribution over all randvars. For the epidemic example, the joint distribution is given by

$$P(Epid, Nat_1, \ldots, Nat_M, Sick_1, \ldots, Sick_N) = \frac{1}{Z} \prod_{j=1}^{M} \phi_1(Epid, Nat_j) \prod_{i=1}^{N} \phi_2(Epid, Sick_i)$$

$$Z = \sum_{e \in \mathcal{R}(Epid)} \prod_{j=1}^{M} \sum_{n_j \in \mathcal{R}(Nat_j)} \phi_1(Epid = e, Nat_j = n_j) \prod_{i=1}^{N} \sum_{s_i \in \mathcal{R}(Sick_i)} \phi_2(Epid = e, Sick_i = s_i)$$

with Z as the normalisation constant. The product $\prod$ is a join over shared randvars (here *Epid*), leading to possibly large intermediate results. The sum $\sum$ goes over the range of a randvar, adding the values that the randvar maps to while other randvars have fixed range values. When computing probability distributions, the goal is to exploit a factorisation as much as possible.

2.2 Repeated Inference versus Singular Inference

Suppose we want to compute the marginal distributions $P(Epid)$ and $P(Sick_1)$. For $P(Epid)$, VE sums out each randvar not occurring in the query, eliminating the randvars:

$$\begin{aligned} P(Epid) = & \sum_{n_1 \in \mathcal{R}(Nat_1)} \cdot \ldots \cdot \sum_{n_M \in \mathcal{R}(Nat_M)} \cdot \sum_{s_1 \in \mathcal{R}(Sick_1)} \cdot \ldots \cdot \sum_{s_N \in \mathcal{R}(Sick_N)} \\ & P(Epid, Nat_1 = n_1, \ldots, Nat_M = n_M, Sick_1 = s_1, \ldots, Sick_N = s_N) \\ = & \frac{1}{Z} \sum_{n_1 \in \mathcal{R}(Nat_1)} \phi_1(Epid, Nat_1 = n_1) \cdot \ldots \cdot \sum_{n_M \in \mathcal{R}(Nat_M)} \phi_1(Epid, Nat_M = n_M) \\ & \cdot \sum_{s_1 \in \mathcal{R}(Sick_1)} \phi_2(Epid, Sick_1 = s_1) \cdot \ldots \cdot \sum_{s_N \in \mathcal{R}(Sick_N)} \phi_2(Epid, Sick_N = s_N) \end{aligned} \tag{2.1}$$

In each factor, VE sums out the randvar that is not *Epid*, multiplying the resulting factors in the end, and normalising the result. The computation of Z is not necessary, a simple

α normalisation suffices. For $P(Sick_1)$, VE does not sum-out $Sick_1$. Instead, VE sums out *Epid*. After summing out the $Sick_i$ randvars (except $Sick_1$) and Nat_j randvars, VE multiplies the resulting factors into one s.t. *Epid* only appears in one factor (prerequisite for VE) and sums out *Epid*. Comparing both computations, we can observe that both queries require eliminating the Nat_j randvars and all $Sick_j$ randvars but $Sick_1$.

In a jtree, the model is clustered into submodels (discussed in detail in Chapter 4). Each node in a jtree is a cluster, i.e., a set of randvars. Each cluster has a local model containing those factors of the model whose arguments appear in the cluster, as well as information required from other clusters. For the epidemic example, each combination of *Epid* with a Nat_j and a $Sick_i$ randvar forms a cluster as shown in Fig. 2.2 with local models in grey. JT answers a query on a cluster that contains the query term. For $P(Epid)$, JT can choose any cluster, e.g., $\{Nat_2, Epid\}$, and eliminates all non-query randvars of the chosen cluster, i.e., Nat_2, using VE:

$$P(Epid) = \frac{1}{Z}\phi'(Epid) \cdot \sum_{n_2 \in \mathcal{R}(Nat_2)} \phi_1(Epid, Nat_2 = n_2)$$

where $\phi'(Epid)$ refers to information from other nodes. For query $P(Sick_1)$, JT eliminates *Epid* from the factors at cluster $\{Sick_1, Epid\}$. A jtree introduces some overhead, but in the above example, immensely decreases the number of computations to perform for answering a single query.

2.3 Ground Inference versus Lifted Inference

The example also shows the great potential of lifting. In Eq. (2.1), all Nat_j sums are identical due to ϕ_1 being identical. The same holds for the $Sick_i$ sums. Lifted summing out computes the sum once and exponentiates the result to the number of sums:

$$P(Epid) = \frac{1}{Z}\left(\sum_{n \in \mathcal{R}(Nat_1)} \phi_1(Epid, Nat_1 = n)\right)^M \cdot \left(\sum_{s \in \mathcal{R}(Sick_1)} \phi_2(Epid, Sick_1 = s)\right)^N$$

Symmetries in a model affect its jtree. In the example, the clusters containing Nat_j randvars are all duplicates of each other. The same holds for the $Sick_i$ clusters. Information in each ϕ' (or ϕ'') is identical. Two instead of $M + N$ clusters suffice. For query $P(Epid)$, LJT chooses one of the two clusters and eliminates all non-query randvars in the chosen cluster using LVE:

$$P(Epid) = \frac{1}{Z} \cdot \phi'(Epid) \cdot \left(\sum_{n \in \mathcal{R}(Nat_1)} \phi_1(Epid, Nat_1 = n)\right)^M$$

2.4 More Query Types

The queries so far, $P(Epid)$ and $P(Sick_1)$, concern a single query term, $Epid$ and $Sick_1$ respectively. But queries may be more complex than concerning a single query term. With more complex queries, challenges arise to answer queries efficiently by still leveraging lifting and selecting clusters. Another query type we look at are conjunctive queries such as $P(Epid, Sick_1)$ containing more than one query terms, which are not straightforward to answer if the query terms do not occur in one cluster. We also look at a special form of conjunctive query, namely, conjunctions of interchangeable query terms, e.g., $P(Sick_1, Sick_2, \ldots, Sick_N)$. These queries, answered naively, can lead LVE to revert to propositional VE. Another important query type are assignment queries in the form of MPE and MAP queries. In contrast to eliminating randvars by summing out as before, an MPE requires eliminating randvars by maxing out. For the epidemic example, an MPE query is answered by solving

$$\underset{e\in\mathcal{R}(Epid)}{\arg\max}\ \underset{n_1\in\mathcal{R}(Nat_1)}{\arg\max}\ \ldots\ \underset{n_M\in\mathcal{R}(Nat_M)}{\arg\max}\ \underset{s_1\in\mathcal{R}(Sick_1)}{\arg\max}\ \ldots\ \underset{s_N\in\mathcal{R}(Sick_N)}{\arg\max} P(Epid = e, Nat_1 = n_1, \ldots, Nat_M = n_M, Sick_1 = s_1, \ldots, Sick_N = s_N)$$

which can be optimised by exploiting the factorisation of the full joint, which, under lifting, reduces to solving $\arg\max$ once for the different Nat_j and $Sick_i$ randvars. An MAP query asks for the most likely assignment to a subset of all randvars, which involves summing out the remaining randvars, e.g.,

$$\underset{e\in\mathcal{R}(Epid)}{\arg\max}\ \underset{s_1\in\mathcal{R}(Sick_1)}{\arg\max}\ \ldots\ \underset{s_N\in\mathcal{R}(Sick_N)}{\arg\max} \sum_{n_1\in\mathcal{R}(Nat_1)} \cdot \ldots \cdot \sum_{n_M\in\mathcal{R}(Nat_M)} P(Epid = e, Nat_1 = n_1, \ldots, Nat_M = n_M, Sick_1 = s_1, \ldots, Sick_N = s_N)$$

Computing an MAP query is harder than an MPE query as $\arg\max$ and $\sum$ are not commutative, thereby restricting the order of elimination, which may lead to prohibitively large intermediate results. The given MAP query does not lead to larger intermediate results than answering an MPE query. But, an MAP query without $Epid$ requires $Epid$ to be summed out before maxing out the $Sick_i$ randvars. To sum out a randvar, it may only occur in one factor, which means that all factors, including the N $Sick_i$ factors, need to be multiplied into one factor, which leads to a factor size of 2^{N+1} instead of a maximum factor size of 2^2. The problem of larger intermediate factors than during MPE queries or simple probability distribution queries also apply for lifted algorithms, but under lifting, identical $\sum$ and $\arg\max$ operations are only calculated once

In the forthcoming chapters, we present lifted algorithms for answering multiple queries of various types. Before starting with the main contribution, LJT for repeated inference, we introduce notations and recap the original LVE.

Chapter 3

Preliminaries

This section presents definitions for models and the QA problem. We recap LVE for single queries and introduce FO dtrees, which LJT uses for the construction of FO jtrees.

3.1 Parameterised Models

In the epidemic example, all N persons and M natural disasters behave in the same way w.r.t. *Epid*. To facilitate modelling such a scenario, logical variables (logvars) parameterise randvars (parameterised randvars, PRVs) to represent a set of randvars. Logvars have a domain that contains constants. Replacing logvars with constants, i.e., grounding a logvar in a PRV, leads to classic propositional randvars. For certain scenarios, one might wish to restrict logvars to certain constants, which is why PRVs are often accompanied by a constraint indicating admissible constants. Given a set of PRVs, a world still describes a particular state where each grounding of a PRV has some value. This form of notation and its semantics have its roots in the work by Poole (2003). The definitions here are mostly based on the definitions given by Taghipour *et al.* (2013c), though we have developed the definitions further. We first define PRVs with all its components.

Definition 3.1.1 (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 $(\mathcal{X}, C_{\mathcal{X}})$ 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 PRV 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, substituting 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 specify the randvars from the epidemic example as PRVs.

Example 3.1.1. The set of logvar names $\mathbf{L} = \{X, D\}$ contains the logvars for the epidemic example. Logvar X represents people with domain $\mathcal{D}(X) = \{x_1, \ldots, x_N\}$, e.g., $\mathcal{D}(X) = \{alice, bob, eve\}$ for $N = 3$. Logvar D represents disasters with domain $\mathcal{D}(D) = \{d_1, \ldots, d_M\}$, e.g., $\mathcal{D}(D) = \{earthquake, flood\}$ for $M = 2$. The set of randvar names is $\mathbf{R} = \{Epid, Sick, Nat\}$. Combining X with $Sick$ and D with Nat leads to the PRVs $Sick(X)$ and $Nat(D)$. Additionally, $Epid$ forms a parameterless PRV. The range of each PRV is boolean. A substitution $\theta = \{X \to eve\}$ applied to $Sick(X)$ leads to $Sick(eve)$. A constraint $C = ((X), C_{(X)})$ allows for restricting X to a subset of its domain $\mathcal{D}(X)$, e.g., $C_{(X)} = \{(eve), (bob)\}$. Given C, $gr(Sick(X)_{|C})$ refers to set of all instances of $Sick(X)$ under C, i.e., $\{Sick(eve), Sick(bob)\}$. Given a $\top$ constraint, the set also includes $Sick(alice)$.

Another syntactical construct is a counting randvar (CRV), which possibly allows for a compact encoding of a factor. The idea behind a CRV is that there is a set of n interchangeable randvars where it does not matter which randvars have a certain range value, 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 of those n randvars have this value v. Let us look at an example.

Example 3.1.2 (CRV as a compact encoding)**.** Assume a factor ϕ with $n = 3$ boolean arguments R_1, R_2, and R_3 (i.e., $m = 2$ range values), mapping to potentials in $\{5, 6, 7\}$:

$$(\neg r_1, \neg r_2, \neg r_3) \mapsto 5,\ (\neg r_1, \neg r_2, r_3) \mapsto 6,\ (\neg r_1, r_2, \neg r_3) \mapsto 6,\ (\neg r_1, r_2, r_3) \mapsto 7,$$
$$(r_1, \neg r_2, \neg r_3) \mapsto 6,\ (r_1, \neg r_2, r_3) \mapsto 7,\ (r_1, r_2, \neg r_3) \mapsto 7,\ (r_1, r_2, r_3) \mapsto 6$$

Two $false$ values and one $true$ value map to 6. One $false$ value and two $true$ values map to 7. 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 $n = 3$ interchangeable arguments, with the first position referring to $true$ and the second to $false$. The factor is defined by

$$[0, 3] \mapsto 5,\ [1, 2] \mapsto 6,\ [2, 1] \mapsto 7,\ [3, 0] \mapsto 6.$$

Both ϕ and ψ encode the same information. But whereas ϕ has $m^n = 2^3 = 8$ mappings, ψ has $\binom{n+m-1}{m-1} = \binom{3+2-1}{2-1} = 4$ mappings, which is no longer exponential in n.

A CRV as part of the model allows for explicitly modelling different probabilities for n interchangeable randvars R_i being all true, all false, or something in between. Instead of n arguments, a factor has one argument A with a range of histograms from $[n, 0]$ to $[0, n]$. The range of histograms contains all possibilities of distributing n objects into m buckets, m being the number of possible range values of the R_i randvars.

Example 3.1.3 (Modelling with a CRV)**.** Consider not only one but five epidemics. It is most likely that no epidemic occurs at all, less likely that one occurs, but least likely that all occur. With a logvar E of five domain values and a CRV $\#_E[Epid(E)]$, a factor $\phi(\#_E[Epid(E)])$ models such a scenario:

$$[0,5] \mapsto 10,\ [1,4] \mapsto 5,\ [2,3] \mapsto 3,\ [3,2] \mapsto 3,\ [4,1] \mapsto 2,\ [5,0] \mapsto 1$$

CRVs allow for a compact encoding as well as a more straight-forward modelling of some scenarios, and will become important during LVE as a major device for enabling lifted computations. We formally define a CRV next.

Definition 3.1.2 (CRV)**.** Let $R(\mathbf{X})_{|C}$ denote a PRV under constraint C with $lv(R(\mathbf{X})) = \{X\}$, meaning either $\mathbf{X}$ is a singleton set or other parameters of 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 its count n_i from the tuple (v_i, n_i). Summing over j histograms $\{(v_i, n_{i,j})\}_{i=1}^m$ of a PRV means adding for each v_i the n_i counts, i.e., $\{(v_i, \sum_j n_{i,j})\}_{i=1}^m$. Multiplying a histogram $\{(v_i, n_i)\}_{i=1}^m$ with a value c yields a histogram $\{(v_i, c \cdot n_i)\}_{i=1}^m$. The multinomial coefficient $Mul(h)$ denotes the number of assignments h encodes (Taghipour, 2013), given by

$$Mul(h) = \frac{n!}{\prod_{i=1}^m n_i!} \tag{3.1}$$

If $\{X\} \subset lv(R(\mathbf{X}))$, the CRV is a *parameterised CRV (PCRV)* representing a set of CRVs. Since counting binds logvar X, $lv(\#_X[R(\mathbf{X})]) = lv(R(\mathbf{X})) \setminus \{X\}$.

With PRVs and PCRVs, we need to define how to determine set relations and operations as well as how to perform element tests.

Definition 3.1.3. Let two sets $\mathbf{A}'_{|C'}$ and $\mathbf{A}''_{|C''}$ of constrained P(C)RVs be given. Then,

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

where $op \in \{=, \subset, \subseteq, \supset, \supseteq, \cup, \cap\}$. An element test of a P(C)RV $A_{|C} \in \mathbf{A}'_{|C'}$ is determined by $A_{|C} \in \mathbf{A}'_{|C'}$ iff $gr(A_{|C}) \subseteq gr(\mathbf{A}'_{|C'})$. An element test of an instance of a P(C)RV $R(\mathbf{x})$ works analogously, i.e., $R(\mathbf{x}) \in \mathbf{A}'_{|C'}$ iff $R(\mathbf{x}) \in gr(\mathbf{A}'_{|C'})$

In most cases, it suffices to work on the P(C)RVs and their constraints. E.g., for an equality of two PRVs, the randvar names and the constraints have to coincide. If one were to assume that logvar names refer to distinct domains, a situation which occurs during FO jtree construction (see Chapter 4), one does not even need to compare constraints but only the randvar names and the logvars appearing in the PRVs.

At this point, we have constructs to represent relations in a world. We still need means to form a model with logvars, PRVs, and PCRVs. Here, parametric factors (parfactors) come into play. Parfactors allow for PRVs and PCRVs as arguments. A parfactor describes a factor, mapping argument values to real values (potentials), of which at least one is non-zero. This factor applies to each instance of the arguments.

Definition 3.1.4 (Parfactor, model)**.** 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 P(C)RVs, 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 *model*.

Fully specifying ϕ requires all input-output pairs. The definitions still permit models with only propositional randvars. For a parfactor or model P, $lv(P)$ refers to the logvars in P, $gr(P)$ to the set of instances of P, leading to a set of grounded parfactors. The term $rv(P)$ refers to the set of PRVs with their constraints in P. Next, we model the epidemic example as a parameterised model and specify an extended model G_{ex} as a running example for the remainder of this thesis.

Example 3.1.4. Using the PRVs $Epid$, $Sick(X)$, and $Nat(D)$, $\mathcal{D}(X) = \{x_i\}_{i=1}^{N}$ and $\mathcal{D}(D) = \{d_j\}_{j=1}^{M}$, and factor names ϕ_1, ϕ_2, the epidemic example becomes $G = \{g_i\}_{i=1}^{2}$,

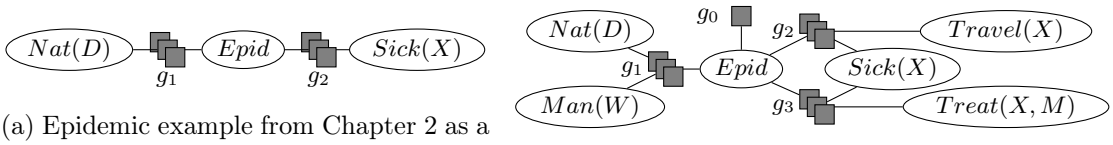
$$g_1 = \forall d \in \{d_j\}_{j=1}^{M} : \phi_1(Epid, Nat(d))_{|\top} = \phi_1(Epid, Nat(D)),$$
$$g_2 = \forall x \in \{x_i\}_{i=1}^{N} : \phi_2(Epid, Sick(x))_{|\top} = \phi_2(Epid, Sick(X)).$$

Figure 3.1a shows G as a parfactor graph, i.e., an FG with PRVs as variable nodes. Factor nodes are layered if any input contains a logvar. $gr(G)$ yields a set of factors equivalent to the original example, where $Sick(x_i)$ corresponds to $Sick_i$ and $Nat(d_j)$ to Nat_j.

For the running example, we extend G to include man-made disasters, e.g., a man-made virus or a war to cause an epidemic. A person may also travel, contributing to a faster spread of a disease. Helping against an epidemic are some form of medicine for treating a person. Formally, $\mathbf{L} = \{D, W, M, X\}$, $\Phi = \{\phi_0, \phi_1, \phi_2, \phi_3\}$, and $\mathbf{R} = \{Epid, Nat, Man, Sick, Travel, Treat\}$. The domains are $\mathcal{D}(D) = \{earthquake, flood\}$, $\mathcal{D}(W) = \{virus, war\}$, $\mathcal{D}(M) = \{injection, tablet\}$, and $\mathcal{D}(X) = \{alice, bob, eve\}$. Next to $Epid$, $Sick(X)$, and $Nat(D)$, the boolean PRVs $Man(W)$, $Travel(X)$, and $Treat(X, M)$ exist. The model is given by $G_{ex} = \{g_0, g_1, g_2, g_3\}$ where

$$\begin{aligned}
g_0 &= && \phi_0(E), \\
g_1 &= \forall (d, w) \in \mathcal{D}(D) \times \mathcal{D}(W) && :\phi_1(E, Nat(d), Man(w))_{|\top} = \phi_1(E, Nat(D), Man(W)), \\
g_2 &= \forall (x) \in \mathcal{D}(X) && :\phi_2(E, S(x), Travel(x))_{|\top} = \phi_2(E, S(X), Travel(X)), \\
g_3 &= \forall (x, m) \in \mathcal{D}(X) \times \mathcal{D}(M) && :\phi_3(E, S(x), Treat(x, m))_{|\top} = \phi_3(E, S(X), Treat(X, M)),
\end{aligned}$$

with $Epid$ abbreviated by E and $Sick$ by S. We omit concrete mappings for ϕ_0 to ϕ_3, of which ϕ_0 has 2^1 and the others 2^3 input-output pairs.



(a) Epidemic example from Chapter 2 as a parameterised model

(b) Extended epidemic example model G_{ex}

Figure 3.1: Parfactor graphs

Figure 3.1b depicts G_{ex} as a parfactor graph with four factor nodes for g_0 to g_3 and six variable nodes for the six PRVs. The factor node of g_0 is not layered as *Epid* is parameterless. The other factor nodes have layers since g_1, g_2, and g_3 contain PRVs.

The semantics of a parameterised model G is given by grounding w.r.t. constraints and building a full joint distribution, formally defined as follows.

Definition 3.1.5 (Semantics)**.** 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 $\mathcal{A}$.

Basic query types on G are (i) a probability of a particular event, i.e., $P(Q = q)$, (ii) a marginal distribution of a randvar, i.e., $P(Q)$, or (iii) a conditional 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. P_G. We define a query as follows.

Definition 3.1.6 (Query, QA problem)**.** A *query* $P(Q|\{E_j = e_j\}_{j=1}^m)$ consists of a *query term* Q and a set of events $\{E_j = e_j\}_{j=1}^m$, with $Q \in rv(G)$ and $E_j \in rv(G)$ being grounded or parameterless PRVs, i.e., of the form $R(\mathbf{x})$ or R. Given a non-empty set of events, the query is for a conditional distribution. Otherwise, the query is for a marginal distribution. To query a probability, the query term is an event $Q = q$. The *QA problem* refers to the problem of computing a probability (distribution) for a query.

Example 3.1.5 (Queries)**.** For G_{ex}, $P(Treat(eve, injection))$ is a query without a set of events asking for the marginal distribution of $Treat(eve, injection)$. Given a single event $Sick(eve) = true$, $P(Treat(eve, injection)|sick(eve))$ asks for a conditional distribution.

Over the course of this dissertation, the query definition becomes more general. Before looking at QA algorithms, we take a closer look at the set of events, also referred to as evidence. If a model contains PRVs, events may concern groundings of one PRV with the same observation. Thus, a parfactor compactly encodes these events.

Definition 3.1.7 (Evidence)**.** A parfactor $\phi_e(R(\mathbf{X}))_{|C_e}$ encodes *evidence* for a set of events $\{R(\mathbf{x}_i) = o\}_{i=1}^n$ of a PRV $R(\mathbf{X})$ or PCRV $\#_X[R(\mathbf{X})]$. ϕ_e maps the value o to 1 and the other values in $\mathcal{R}(R(\mathbf{X}))$ to 0. $C_e = (\mathcal{X}, \{(\mathbf{x}_i)\}_{i=1}^n)$ holds observed instances.

Example 3.1.6 (Evidence)**.** Assume we observe $sick(eve)$, which is only one event. The parfactor $\phi_e(Sick(X))_{|C_e}$ represents $sick(eve)$ as follows: The factor ϕ_e has the mappings $\phi_e(true) = 1$ and $\phi_e(false) = 0$. $C_e = ((X), \{(eve)\})$ restricts the domain of X to eve. As C_e contains one grounding, we can simplify $\phi_e(Sick(X))_{|C_e}$ to $\phi_e(Sick(eve))$ because C_e is a singleton constraint, meaning (eve) is the only sequence in C_e.

Assume there are 100 people in G_{ex} and we observe $sick(x_i)$ for 10 of them, i.e., the set of events is $\{Sick(x_i) = true\}_{i=1}^{10}$. Then, $C'_e = ((X), \{(x_1), \ldots, (x_{10})\})$ restricts the domain of X to $x_1, \ldots, x_{10}$ in $\phi_e(Sick(X))_{|C'_e}$. Assuming that the set of events also includes $\{Travel(x_i) = true\}_{i=1}^{10}$, another parfactor $\phi_e(Travel(X))_{|C'_e}$ models the $\{Travel(x_i) = true\}_{i=1}^{10}$ subset of events using the same ϕ_e and C'_e as before.

The next section recaps LVE including how evidence encoded in parfactors enables LVE to handle evidence as a whole, i.e., without grounding.

3.2 Lifted Variable Elimination - Singular Inference

LVE is an algorithm for solving the QA problem. Many researchers have refined LVE over the years. LVE seeks to avoid grounding as well as explicitly building a joint distribution using lifting, leveraging relational structures in a model, and the factorisation of a model. LVE accomplishes efficient QA for single queries. But, given another query for the same model, LVE starts with the original model.

Even though the basic idea of lifting is rather intuitive as seen in Chapter 2, a complete specification of LVE as an algorithm is highly involved to ensure correct results. This section highlights some of the LVE operators, recaps LVE as an inference algorithm, and introduces FO dtrees as a helper structure for constructing FO jtrees.

LVE Operators (Taghipour *et al.*, 2013c) LVE works on parameterised models. Its main operator *lifted summing out* basically sums out a PRV for one representative instance and exponentiates the result for the isomorphic instances as seen during the epidemic example in Chapter 2. Lifted summing out of a PRV A has three preconditions to ensure correctness, i.e., summing out isomorphic instances (Taghipour *et al.*, 2013c):

(i) A may only appear in one parfactor (similarly to randvars during VE).

(ii) A must contain all logvars of the parfactor.

(iii) The logvars that are only in A, $\mathbf{X}^{excl}$, need to be count-normalised w.r.t. the remaining logvars in the parfactor. Count normalisation means that for different groundings of the remaining logvars the same number of groundings of $\mathbf{X}^{excl}$ exists.

The importance of these conditions will become apparent over the course of the upcoming examples. Let us look at how LVE sums out PRV $Treat(X, M)$ in G_{ex}.

Example 3.2.1 (Lifted summing out)**.** Only $g_3 = \phi_3(Epid, Sick(X), Treat(X, M))$ contains $Treat(X, M)$. $Treat(X, M)$ also contains all logvars in g_3, with $\mathbf{X}^{excl} = \{M\}$ and logvar X remaining. The constraint in g_3 is $\top$, i.e., each combination of constants from $\mathcal{D}(X)$ and $\mathcal{D}(M)$ exist. Thus, for each constant of X, there appear $|\mathcal{D}(M)| = 2$ constants of M, meaning M is count-normalised w.r.t. X with a count of $c = 2$. Summing out follows VE: Fixing the range values of $Epid$ and $Sick(X)$, LVE adds up the potentials for the range values of $Treat(X, M)$. It then raises the resulting potentials to the power of c, yielding an intermediate parfactor $\phi'_3(Epid, Sick(X))$ with four input-output pairs.

Further LVE operators transform a model to enable lifted summing out. The most basic transforming operator *grounds* a logvar. The operator *multiply* ensures that a PRV A appears in only one parfactor. Multiplying parfactors is even more elaborate than multiplying factors. The multiplication is still a join over shared PRVs. But, multiplying parfactors also needs to consider how many instances each parfactor represents to ensure correct potentials in the result.

Another important operator is a *count conversion* to build a CRV (Milch *et al.*, 2008). Counting binds a logvar, and thus, may lead to a PRV containing all logvars of a parfactor, enabling lifted summing out. The alternative to counting would be grounding, which is exponential in the number of constants, while counting is polynomial as indicated in Example 3.1.2. To illustrate counting, let us sum out $Nat(D)$ and $Man(W)$ in G_{ex}, which LVE also needs to eliminate given $Treat(eve, injection)$ as a query term.

Example 3.2.2 (Counting)**.** $Nat(D)$ and $Man(W)$ occur in $\phi_1(Epid, Nat(D), Man(W))$ but do not contain both logvars. LVE could ground a logvar, e.g., logvar D:

$$\phi_1(Epid, Nat(d_1), Man(W)), \phi_1(Epid, Nat(d_2), Man(W))$$

where $d_1 = earthquake$ and $d_2 = flood$. For summing out $Man(W)$, which also eliminates W, i.e., all potentials are raised to the power of $|\mathcal{D}(W)|$ after summing out, i.e.,

$$\left(\sum_{m \in \mathcal{R}(Man(W))} \phi_1(Epid, Nat(d_1), m) \cdot \phi_1(Epid, Nat(d_2), m) \right)^{|\mathcal{D}(W)|} \tag{3.2}$$

LVE would need to multiply the parfactors s.t. $Man(W)$ appears in one parfactor:

$$\left(\sum_{m \in \mathcal{R}(Man(W))} \phi'_1(Epid, Nat(d_1), Nat(d_2), m) \right)^{|\mathcal{D}(W)|} = \phi''_1(Epid, Nat(d_1), Nat(d_2))$$

Since the product in Expression (3.2) only regards ϕ_1, the resulting parfactor with function ϕ'_1 exhibits the same symmetry as the factor in Example 3.1.2. Instead of grounding D, LVE replaces $Nat(D)$ with a CRV $\#_D[Nat(D)]$ in g_1:

$$\phi_1^{\#}(Epid, \#_D[Nat(D)], Man(W))$$

$\#_D[Nat(D)]$ has a range of $[0, 2]$, $[1, 1]$, and $[2, 0]$, the first position referring to *nat* and

the second position to $\neg nat$. $\phi_1^{\#}$ has 12 input-output pairs instead of the 16 pairs of ϕ_1'. With $e \in \mathcal{R}(Epid)$, $[n_1, n_2] \in \mathcal{R}(\#_D[Nat(D)])$, and $m \in \mathcal{R}(Man(W))$, $\phi_1^{\#}$ is given by:

$$\phi_1^{\#}(e, [n_1, n_2], m) = \phi_1(e, true, m)^{n_1} \cdot \phi_1(e, false, m)^{n_2} \tag{3.3}$$

Counting excludes D as a logvar, enabling summing out $Man(W)$ without grounding:

$$\left(\sum_{m \in \mathcal{R}(Man(W))} \phi_1^{\#}(Epid, \#_D[Nat(D)], m) \right)^{|\mathcal{D}(W)|} = \phi_1^{\#\prime}(Epid, \#_D[Nat(D)])$$

Next, LVE sums out $\#_D[Nat(D)]$ from $\phi_1^{\#\prime}$, which is also more efficient than summing out $Nat(d_1)$ and $Nat(d_2)$ individually in ϕ'' after grounding.

A logvar X has to fulfil preconditions for count conversion to apply in a parfactor $g = \mathbf{X} : \phi(\mathcal{A})_{|C}$ (Taghipour *et al.*, 2013c):

(i) Only one input $A_i \in \mathcal{A}$ contains X.

(ii) X is count-normalised w.r.t. $\mathbf{X} \setminus \{X\}$.

(iii) No inequality constraint exists between X and any other counted logvar in g.

If the above preconditions hold, LVE counts X in g by converting A_i into a CRV A_i'. In the new function ϕ', the input for the position of A_i' is a histogram $h = \{(a_i, n_i)\}_{i=1}^{n}$ and the output is defined accordingly:

$$\phi'(\dots, a_{i-1}, h, a_{i+1}, \dots) \mapsto \prod_{a_i \in \mathcal{R}(A_i)} \phi(\dots, a_{i-1}, a_i, a_{i+1}, \dots)^{h(a_i)}$$

See Expression (3.3) as a showcase. More refined count conversions exist (Apsel and Brafman, 2011; Taghipour *et al.*, 2013b), which we use in later chapters. To sum out a CRV, LVE has to consider that a histogram h may represent several assignments. E.g., histogram $[1, 1]$ for $\#_D[Nat(D)]$ represents two assignments, namely, $nat(d_1), \neg nat(d_2)$ and $\neg nat(d_1), nat(d_2)$. Summing out $\#_D[Nat(D)]$, LVE adds the potential at $[1, 1]$ twice. The number of assignments h encodes is given by $Mul(h)$ defined in Expression (3.1).

Now that we have seen the two most important LVE operators (lifted summing out, count conversion), we look at how LVE prepares a model for eliminating non-query terms, which includes shattering, a term coined by de Salvo Braz *et al.* (2005), and evidence handling, given a query $P(Q|\{E_j = e_j\}_{j=1}^{m})$.

First, LVE *shatters a model on a query term* Q if Q is an instance of a PRV. Shattering splits each parfactor that contains the query term. More specifically, splitting a parfactor g means that LVE replaces g with two copies g^q and g^r, and alters their constraints respectively. The constraint in g^q contains those sequences that include the Q grounding for $\mathbf{X}$. The constraint in g^r holds the remaining sequences. Let us look at G_{ex} and the query term $Treat(eve, injection)$ from the query $P(Treat(eve, injection)|sick(eve))$.

Example 3.2.3 (Shattering)**.** LVE shatters G_{ex} on $Treat(eve, injection)$, which appears in parfactor $g_3 = \phi_3(Epid, Sick(X), Treat(X, M))_{|\top}$ with eve and $injection$ allowed by the constraint $\top$. Shattering duplicates g_3, leading to g_3^q and g_3^r. In $g_3^q = \phi_3(Epid, Sick(X), Treat(X, M))_{|C_3^q}$, the second component of C_3^q is $\{(eve, injection)\}$, which is singleton, i.e., g_3^q can be simplified: $\phi_3(Epid, Sick(eve), Treat(eve, injection))$. The constraint in g_3^r contains the tuples $\{(eve, tablet), (alice, injection), (alice, tablet), (bob, injection), (bob, tablet)\}$. At this point, M is not count-normalised w.r.t. X as eve has one M constant in C_3^q, while $alice$ and bob have two.

Second, LVE *absorbs evidence*, which shatters a model on evidence as well. For each evidence parfactor $g_e = \phi_e(R(\mathbf{X}))_{|C_e}$, LVE tests each parfactor $g \in G$ if $R(\mathbf{X}) \in rv(g)$. If this is the case, LVE splits g on C_e by replacing g with g^e and g^r. The constraint in g^e contains those sequences that include the groundings of $\mathbf{X}$ in C_e. The constraint in g^r contains the remaining sequences. Then, g^e absorbs g_e, which can be thought of as multiplying g_e into g^e, which sets all potentials in g^e to 0 where $R(\mathbf{X}) \neq o$. Then, LVE drops the mappings with 0 and removes $R(\mathbf{X})$ from g^e as $R(\mathbf{X}) = o$ in all mappings. LVE exponentiates the remaining potentials if a logvar disappears with absorption. Though we look at evidence in general, computing conditional probabilities is not guaranteed to be liftable unless evidence consists of PRVs with at most one logvar (Van den Broeck, 2013). Let us look at G_{ex} and evidence $sick(eve)$ from $P(Treat(eve, injection)|sick(eve))$.

Example 3.2.4 (Absorption)**.** After shattering G_{ex} on $Treat(eve, injection)$, LVE shatters G_{ex} also on $sick(eve)$. Parfactors g_0 and g_1 do not contain $Sick(eve)$. Parfactor $g_2 = \phi_2(Epid, Sick(X), Travel(X))$ covers $Sick(eve)$. Shattering splits g_2 into $g_2^e = \phi_2(Epid, Sick(eve), Travel(eve))$ and $g_2^r = \phi_2(Epid, Sick(X), Travel(X))_{|C_2^r}$, C_2^r containing $\{(alice), (bob)\}$. g_3^q contains only $Sick(eve)$. The constraint in g_3^r has the tuples $\{(eve, tablet), (alice, injection), (alice, tablet), (bob, injection), (bob, tablet)\}$. LVE splits g_3^r into g_3^{re} for $(eve, tablet)$ and g_3^{rr} for the remaining tuples. In g_3^{rr}, M is count-normalised w.r.t. X. LVE absorbs $sick(eve)$ in g_2^e, g_3^q, and g_3^{re}, yielding $\phi_2'(Epid, Travel(eve))$, $\phi_3^{q\prime}(Epid, Treat(eve, injection))$, and $\phi_3^{re\prime}(Epid, Treat(eve, tablet))$.

With lifted operations, shattering, and absorption in place, we look at LVE as an inference algorithm for answering a given query.

Algorithm Description The inference algorithm of LVE as specified by Taghipour *et al.* (2013c) answers queries with a single query term. Algorithm 1 shows LVE with input model G, query term Q, and evidence $\mathbf{E}$. LVE first splits G on Q. It absorbs $\mathbf{E}$ (shattering if needed) and starts to eliminate non-query terms. LVE selects an operation from a list of possible sum-out operations based on the expected size of the intermediate result (number of mappings in the resulting parfactor) as a simple heuristics. For lifted summing out, the result is smaller than before because LVE eliminates a PRV. If lifted summing out is not possible, LVE applies one of its transforming operators to enable a lifted sum-out operation, choosing the operator using the same heuristics as before. After

Algorithm 1 Lifted Variable Elimination Algorithm

```
procedure LVE(Model G, Query term Q, Evidence E)
    G ← Shatter G on Q
    G ← Absorb E in G                          ▷ Shatters G on E if necessary
    while G contains non-query PRV do
        if there exists a PRV A eliminable then
            G ← Sum out A in G
        else
            G ← Apply transforming operator applicable in G
    G ← Multiply remaining parfactors in G and normalise the result
    return G                                   ▷ Contains one normalised parfactor
```

a transformation, LVE tests again for lifted summing out. After LVE has eliminated all non-query terms, a set of parfactors with the query term as argument remains. The set may be singleton. LVE multiplies the parfactors in the set into one and normalises the result to get the queried probability distribution. For a new query or new evidence, LVE restarts from the original model G. To illustrate LVE, let us look at how LVE answers $P(Treat(eve, injection)|sick(eve))$.

Example 3.2.5 (LVE)**.** Figure 3.2 shows G_{ex} as a parfactor graph after query shattering and absorbing evidence according to Examples 3.2.3 and 3.2.4. LVE starts with summing out operations. Proceeding with increasing size of intermediate results, LVE sums out

- $Travel(eve)$ from g_2^e, resulting in $g_2^{e\prime} = \phi_2'(Epid)$,
- $Treat(eve, tablet)$ from g_3^{re}, resulting in $g_3^{re\prime} = \phi_3'(Epid)$,
- $Travel(X)$ from g_2^r (g_2^r contains all occurrences of $Travel(X)$, $Travel(X)$ contains all logvars, X is count-normalised; exponent is 1 as no logvar is eliminated), resulting in $g_2^{r\prime} = \phi_2''(Epid, Sick(X))$, and
- $Treat(X, M)$ from g_3^{rr} (g_3^{rr} contains all occurrences of $Treat(X, M)$, $Treat(X, M)$ contains all logvars, M is count-normalised; exponent is 2 as two M values exist for each X value), resulting in $g_3^{rr\prime} = \phi_3''(Epid, Sick(X))$.

Fig. 3.3 shows the remaining model after performing the four sum-out operations. At this point, no further summing out is possible. Next to grounding each logvar (8 to 16 mappings), LVE can count either D or W (each 12 mappings), and multiply $g_2^{r\prime}$ and $g_3^{rr\prime}$ (4 mappings). Thus, LVE multiplies $g_2^{r\prime}$ and $g_3^{rr\prime}$ into $g_{23} = \phi_{23}(Epid, Sick(X))$ and can now eliminate $Sick(X)$, resulting into $g_{23}' = \phi_{23}'(Epid)$. Next, LVE randomly chooses to count D, resulting in $g_1^{\#} = \phi_1^{\#}(Epid, \#_D[Nat(D)], Man(W))$. Now, LVE eliminates $Man(W)$, followed by eliminating $\#_D[Nat(D)]$, resulting in $g_1' = \phi_1'(Epid)$. The remaining non-query randvar is $Epid$. LVE multiplies all remaining parfactors

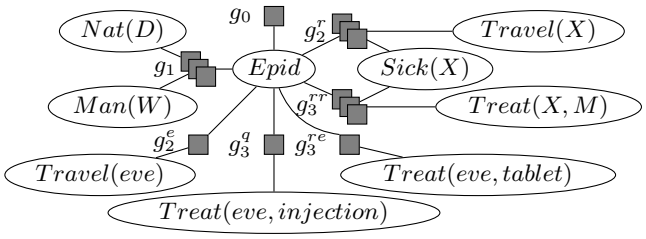


Figure 3.2: Parfactor graph of G_{ex} after shattering and absorbing evidence

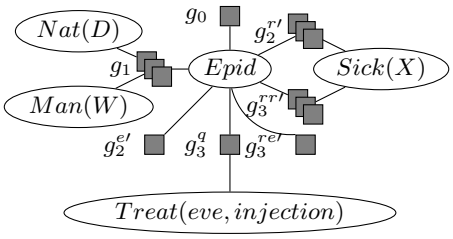


Figure 3.3: Parfactor graph of G_{ex} in Fig. 3.2 after four summing-out

$(g_2^{e\prime},\ g_3^{re\prime},\ g'_{23},\ g'_1,\ g_0,\ g_3^q)$ into one parfactor $g = \phi(Epid, Treat(eve, injection))$ and sums out $Epid$. Normalising $g' = \phi'(Treat(eve, injection))$ provides the result for $P(Treat(eve, injection)|sick(eve))$.

At the end of this example, LVE has successfully produced an answer to the query $P(Treat(eve, injection)|sick(eve))$ on G_{ex}. As demonstrated, LVE provides an efficient means for QA, incorporating relational aspects in its computations. However, it does not handle multiple queries efficiently, redoing computations. Here, LJT comes in, using an FO jtree to cluster model factors to avoid redoing computations. But before presenting LJT, we look at FO dtrees, as means to build FO jtrees.

First-order Dtrees The changes in a model over the course of an LVE run can be represented as a tree based on partitions of parfactors. LVE starts with all parfactors in separate partitions. Multiplying parfactors combines the partitions of the parfactors. When summing out a P(C)RV from a parfactor afterwards, the P(C)RV disappears from the partition. At the end of an LVE run, all partitions have been combined and only the query term is left. One can represent such a partitioning as a tree, specifically an FO dtree, building one bottom-up as follows: The parfactors appear at the leaves. Inner nodes represent multiplication of parfactors and subsequent summing out of a P(C)RV along with the manipulation of partitions. When summing out a P(C)RV, LVE multiplies parfactors by way of representatives, sums out a representative instance of the P(C)RV and exponentiates the result for all isomorphic instances. Thus, one inner node might represent the multiplication and summing out with representatives, while another represents the exponentiation for the isomorphic instances. The root represents the last operation performed in an LVE run. Since the elimination order during LVE is not necessarily unique, there does not exist a unique FO dtree. The same holds for VE and its representation as a propositional dtree.

One can interpret an FO dtree top-down as a recursive decomposition of a model into partitions of parfactors where one partition includes P(C)RVs or logvars not part of any other partition. This top-down interpretation gives rise to a simple routine for construct-

ing an FO dtree without a query and a specific elimination order (Taghipour, 2013).[1] At the root, a model forms one partition. Within a partition, one can partition the current model based on P(C)RVs/logvars or test for logvars $\mathbf{X}$ that, if grounded, lead to isomorphic models that are partially grounded w.r.t. $\mathbf{X}$. If such logvars exist, an inner node for isomorphic instances, called DPG node for decomposition into partial groundings, is introduced and the model is partially grounded with representative constants for $\mathbf{X}$. Partitioning and partially grounding a model continues until parfactors only contain representative constants and each appears in an own partition. Before we look at an FO dtree for G_{ex}, we define FO dtrees based on the definition by Taghipour *et al.* (2013a).

Definition 3.2.1 (FO dtree nodes)**.** An FO dtree can have three types of nodes:

- A *DPG node* $T_{\mathbf{X}}$ represents an exponentiation (bottom up) or a decomposition into partial groundings (top down). $T_{\mathbf{X}}$ is given by a tuple $(\mathbf{X}, \mathbf{x}, C)$. $\mathbf{X} = \{X_1, \dots X_k\}$ is a set of logvars of the same domain $\mathcal{D}_{\mathbf{X}}$. $\mathbf{x} = \{x_1, \dots x_k\}$ is a set of *representative constants* from $\mathcal{D}_{\mathbf{X}}$. C is a constraint on $\mathbf{x}$ such that $\forall i, j : x_i \neq x_j$.
- A VE node T represents a partitioning (multiplication, summing out bottom up).
- A leaf node L contains a parfactor, grounded with representative constants.

Let the sets DPG, VE, and $Leaf$ contain all DPG, VE, and leaf nodes respectively. Then, an *FO dtree* for a model G is a tree (V, E) where $V = DPG \cup VE \cup Leaf$ and

$$E = (DPG \times VE) \cup (VE \times DPG) \cup (VE \times VE) \cup (VE \times Leaf)\,.$$

A VE node can follow a DPG node as well as a VE node. DPG nodes and leaf nodes can only follow VE nodes. Further, the following holds:

- Each DPG node $T_{\mathbf{X}}$ has a child VE node $T_{\mathbf{x}}$, whose model is a representative of the model at $T_{\mathbf{X}}$ using a bijective substitution $\theta = \{X_i \to x_i\}_{i=1}^{k}$ mapping $\mathbf{X}$ to $\mathbf{x}$.
- Child $T_{\mathbf{x}}$ of a DPG node $T_{\mathbf{X}}$ has $k!$ children isomorphic up to permutation of $\mathbf{x}$.
- Each leaf with representative constant x in its parfactor descends from exactly one DPG node $T_{\mathbf{X}}$ s.t. $x \in \mathbf{x}$.
- Each leaf descending from DPG node $T_{\mathbf{X}}$ has all constants $\mathbf{x}$ in its parfactor.

Given a DPG or VE node T in an FO dtree, $mod(T)$ refers to all parfactors and $rv(T)$ to all PRVs in leaf parfactors of the subtree starting in T after applying inverse substitutions for the DPG nodes in the subtree. Given a leaf node L, $mod(L)$ refers to the parfactor in L and $rv(L)$ to the arguments of its parfactor. Depicting an FO dtree as a graph, we denote a DPG node $T_{\mathbf{X}}$ by $(\forall \mathbf{x} : C)$, a VE node as a dot, and a leaf by its parfactor.

[1] An FO dtree requires a normal form for the model, in which each pair of logvars has either identical or distinct domains, each constraint encodes $X \neq X'$ for each pair of co-domain logvars appearing in a parfactor, and for each pair of co-domain logvars in a parfactor, its constraint encodes $X \neq X'$.

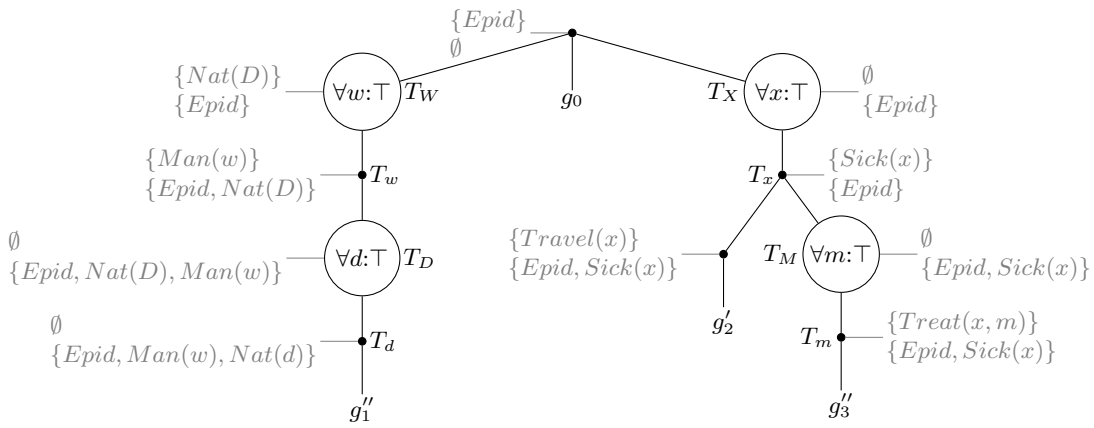


Figure 3.4: FO dtree for G_{ex} (grey labels: upper line – cutset, lower line – context)

Example 3.2.6 (FO dtree)**.** Figure 3.4 depicts an FO dtree without set braces for the DPG node labels and their children. Ignore the grey labels for now. Let us look at the FO dtree in a top-down fashion. The root node is a VE node with three children. One is a leaf node for g_0. The remaining model is partitioned w.r.t. logvars (one could use X, W, or D for partitioning). The two partitions are $\{g_1 = \phi_1'(Epid, Nat(D), Man(W))\}$ and $\{g_2 = \phi_2(Epid, Sick(X), Travel(X)),\ g_3 = \phi_3(Epid, Sick(X), Treat(X, M))\}$. In the first partition, logvars D and W allow for a partial grounding into isomorphic instances. If choosing W, as done for the FO dtree in the figure, the root has a child DPG node $T_W = (W, w, \top)$. T_W has a child T_w replacing logvar W with representative object w, i.e., $\theta = \{W \to w\}$. T_w has the model $\{g_1' = \phi_1'(Epid, Nat(D), Man(w))\}$. Again, D permits a partial grounding into isomorphic instances. Thus, the child of T_w is a DPG node $T_D = (D, d, \top)$ with child T_d where $\theta = \{D \to d\}$. The model at T_d is $\{g_1'' = \phi_1''(Epid, Nat(d), Man(w))\}$. g_1'' contains only representative constants so we have a leaf node. Going back to the root, we look at the second partition, where logvar X allows for a decomposition into partial groundings. So, the root has a third child, a DPG node $T_X = (X, x, \top)$, leading to a substitution $\theta = \{X \to x\}$. T_X has a child T_x with the model $\{g_2' = \phi_2'(Epid, Sick(x), Travel(x)),\ g_3' = \phi_3'(Epid, Sick(x), Treat(x, M))\}$. The model is partitioned on logvar M. Thus, the children are a VE node with model $\{g_2'\}$ and a DPG node $T_M = (M, m, \top)$, as grounding M leads to isomorphic instances. The VE node with model $\{g_2'\}$ has a leaf child containing g_2'. T_M has a child T_m with model $\{g_3'' = \phi_3''(Epid, Sick(x), Treat(x, m))\}$. T_m has a leaf child containing g_3''.

Starting bottom up, the VE node T_m represents summing out $Treat(X, M)$ using representatives x and m. Its parent node represents the exponentiation for isomorphic instances of M, i.e., *injection* and *tablet*. The parent node of the g_2' leaf node represents summing out $Travel(X)$ using representative x. No immediate DPG node appears as a parent since summing out $Travel(X)$ does not eliminate a logvar. VE node T_x represents

multiplying the two resulting parfactors of the just mentioned sum-out operations and summing out $Sick(X)$ using x. T_X represents the exponentiation for the isomorphic instances of X, with a parfactor with argument *Epid* remaining. Starting at the leaf node of g_1'', a similar interpretation about a sequence of LVE operations is possible.

Darwiche (2001) defines properties, namely cutset, context, and cluster, for propositional dtrees. The properties, which denote sets of randvars, characterise VE operations at a node, where cutset randvars are eliminated ("cut"), context randvars appear fixed, and a cluster is the union of both. Taghipour *et al.* (2013a) define cutsets, contexts, and clusters for FO dtrees. LJT will use the clusters for constructing an FO jtree. We define the sets for FO dtrees based on the definitions by Taghipour *et al.* (2013a).

Definition 3.2.2 (Cutset, context, cluster)**.** Let $ancestors(T)$ refer to all nodes between node T and the root, $children(T)$ to all direct children of T. If $T \in DPG$, $children(T) = \{T_{\mathbf{x}\theta} | T_{\mathbf{x}} \in children(T) \wedge \theta \in \Theta_{\mathbf{x}}\}$, $\Theta_{\mathbf{x}} = \{\theta_i\}_{i=1}^n$ is the set of grounding substitutions for representatives $\mathbf{x}$. *Cutset*, *context*, and *cluster*[2] of a DPG or VE node T are given by

$$
\begin{aligned}
cutset(T) &= \left(\bigcup_{T_i, T_j \in children(T)} rv(T_i) \cap rv(T_j) \right) \setminus acutset(T) \\
acutset(T) &= \bigcup_{T' \in ancestors(T)} cutset(T') \qquad \text{(ancestor cutset)} \\
context(T) &= rv(T) \cap acutset(T) \\
cluster(T) &= cutset(T) \cup context(T)
\end{aligned}
$$

For a leaf node L with parfactor g, $cutset(L) = \emptyset$ and $context(L) = cluster(L) = rv(g)$.

An FO dtree shows if a count conversion is necessary: Counting logvars $\mathbf{X}$ is possible and necessary if, at a DPG node $T_{\mathbf{X}}$, $\mathbf{X}$ appears in the cluster at $T_{\mathbf{X}}$. Next, we illustrate cutsets, contexts, and clusters in the FO dtree of G_{ex}.

Example 3.2.7 (Properties)**.** The grey labels in Fig. 3.4 are cutsets and contexts. The upper line is the cutset, the lower line the context at each inner node. Leaf contexts and clusters consist of the parfactor arguments. At the root, the cutset consists of *Epid* since it occurs in all children. The context is empty with an empty acutset. The left child T_W is a DPG node with child T_w where the possible grounding substitutions for w are *virus* and *war*. Thus, $children(T_W) = \{T_{virus}, T_{war}\}$, relevant for computing the cutset of T_W. The intersection of the randvars of T_{virus} and T_{war} contains *Epid* and $Nat(D)$, which means $Nat(D)$ remains in the cutset of T_W, since *Epid* is in the acutset of T_W. Intersecting the randvars of T_W with its acutset leaves *Epid* in the context of T_W. At T_w, the only child is a DPG node with randvars *Epid*, $Nat(D)$, and $Man(w)$, meaning no other child exists for intersection. Removing the acutset leaves $Man(w)$ in the cutset.

[2]There exists an algorithm to calculate cutsets in polynomial time for each inner node based on a set of conditions derived from the definitions (Taghipour, 2013).

The context consists of the other two randvars. For T_D, the grounding substitutions are *earthquake* and *flood*. Intersecting the randvars of $T_{earthquake}$ and T_{flood} results in $\{Epid, Man(w)\}$, which both appear in the acutset of T_D. Intersecting $Epid$, $Man(w)$, and $Nat(D)$ with its acutset for the context leads to all three variables in the context.

The DPG child T_X of the root has an empty cutset as the only randvar in two $T \in children(T_X)$, $Epid$, appears in its acutset, i.e., $Epid$ appears in the context of T_X. All other PRVs contain an x that is replaced with different values, i.e., *alice*, *bob*, *eve*, and therefore lead to different randvars. At T_x, the VE child has the randvars $Epid$, $Sick(x)$, and $Travel(x)$, while the DPG child has the randvars $Epid$, $Sick(x)$, and $Treat(x, M)$. The intersection leaves $Epid$ and $Sick(x)$, which results in $Sick(x)$ in the cutset and $Epid$ in the context. At the VE child, only one child exists with randvars $Epid$, $Sick(x)$, and $Travel(x)$, meaning a cutset of $Travel(x)$ and a context of $Epid$ and $Sick(x)$. At the DPG child T_M, the grounding substitutions are *injection* and *tablet*. Intersecting the randvars of the children $T_{injection}$ and T_{tablet} results in $\{Epid, Sick(x)\}$, which is the acutset of T_M. Thus, the cutset is empty, with the context containing $\{Epid, Sick(x)\}$. At T_m, the only child with randvars $Epid$, $Sick(x)$, and $Treat(x, m)$ leads to a cutset of $\{Treat(x, m)\}$ and a context of $\{Epid, Sick(x)\}$.

To illustrate the connection between the properties and the LVE operations behind the nodes, let us look at the parent of leaf g_3''. The parent is a VE node, which stands for the representative elimination of $Treat(x, m)$, which appears in the cutset of the VE node. The PRVs in the context, $Epid$ and $Sick(x)$, also appear during summing out as a fixed part, which remains after the elimination. The parent DPG node representing the exponentiation does not have $Treat(x, m)$ in its cutset, as it is already eliminated. At the parent VE node of g_2', LVE eliminates $Travel(x)$, which appears in the cutset, while $Epid$ and $Sick(x)$ remain in the result after summing out. At T_x, the two paths unite. T_x represents a multiplication for and summing out of $Sick(x)$, which appears in the cutset, while T_X represents the exponentiation, after which $Epid$ remains.

The left child of the root does not represent the LVE operations so explicitly as LVE performs a count conversion, which the FO dtree shows as necessary and possible: At DPG node T_D, the PRV $Nat(D)$ appears in the cluster of T_D, specifically in the context of T_D. The cutsets below T_w are empty, which follows LVE since it cannot eliminate any PRV without a count conversion. After counting D in $Nat(D)$, LVE eliminates $Man(w)$ represented at T_w and exponentiates the result at T_W. After eliminating $Man(W)$, LVE can eliminate $\#_D[Nat(D)]$, which appears in the cutset of T_W to eliminate next. One could reformulate the FO dtree with a CRV $\#_D[Nat(D)]$: The DPG node T_D and its child T_d disappear and the leaf follows after T_w. An extra VE node between the root and T_W explicitly represents the elimination of $\#_D[Nat(D)]$.

LVE is an algorithm for single queries that leverages relational structures for efficient inference. LVE uses the factorisation of a full joint expressed in a probabilistic graphical model. The factors are then used for computations, with computations organised in

such a way that small intermediate results arise. LVE falls into the category of bottom-up dynamic programming approaches (Dechter, 1999) where the individual factors are combined and processed to produce the final result. Another algorithm falling into this category is the propositional JT, which uses VE in its computations.

In contrast to the bottom-up method just presented, WMC methods for example perform probabilistic inference by way of top-down dynamic programming (Darwiche, 2001). These methods condition a model on each possible world for a subset of randvars, possibly dividing a model into smaller subproblems. The solutions to the subproblems are then combined into one overall solution. Van den Broeck *et al.* (2011) provide WFOMC as a relational variant of WMC and present FOKC to solve WFOMC problems.

Before presenting LJT as an efficient algorithm for multiple queries based on LVE, we take a brief excursion into FOKC as another lifted algorithm for repeated inference, which we use during the empirical evaluation of the next part for comparison.

3.3 Exkursion: First-order Knowledge Compilation

The inputs for KC-based algorithms are logical formulas associated with a weight, i.e., weighted models. Propositional KC compiles a weighted model into a so-called circuit for probabilistic inference (Darwiche and Marquis, 2002). A circuit is a tree-like structure where leaves contain simple events of randvars, i.e., $A = a$ or $A = \neg a$ for boolean randvars, and inner nodes represent either a conjunction or a disjunction. Events can be set to 0 or 1 based on evidence. Further leaves contain the weights (or probabilities) from the model. During inference, the weights are propagated through the tree with conjunctions being replaced by a multiplication and a disjunction being replaced by an addition. The result at the root is a model count for the underlying model.

To ensure that weights are combined appropriately, the events of the branches going into a conjunction need to be disjoint (independent) s.t. the weights can be multiplied, and only one of the branches going into a disjunction may be true at a time s.t. the weights can simply be added. These restrictions require a model to fulfil a normal form, specifically, the deterministic decomposable negation normal form (d-DNNF). The NNF ensures that only simple events occur with negations only directly in front of terms. Decomposable means that all pairs of conjuncts are independent. Deterministic refers to only one disjunct being true at a time. In summary, d-DNNF allows for efficient reasoning, as computing the weight of a conjunction decomposes into a product of the weights of its conjuncts and the weight of a disjunction is equal to the sum of weights of its disjuncts. Other normal forms exist, which allow for QA of certain query types to depend linearly on the size of the circuit (Darwiche and Marquis, 2002). The disadvantage of the normal forms is that the circuits grow very large. They grow the larger, the more strict the normal form is. The upside is that the more strict the normal form is, the more query types are supported in linear time w.r.t. circuit size.

Given repeated patterns in a model, identical branches exist in a circuit which can be exploited for more efficient inference in two ways: (i) Identical branches can be combined into one representative to form a more compact circuit. (ii) In identical branches, the calculations are identical, which means they can be carried out once for a representative and applied to all instances. Van den Broeck *et al.* apply lifting to KC and WMC, introducing weighted first-order model counting (WFOMC) and a first-order d-DNNF (FO d-DNNF, Van den Broeck *et al.*, 2011; Van den Broeck and Davis, 2012). FOKC aims at solving a WFOMC problem by building FO d-DNNF circuits given a query and evidence and computing model counts on the circuits. Of course, different compilation flavours exist, e.g., compiling into C++ program code (Kazemi and Poole, 2016b), which does not change the problem in itself. So, we focus on FOKC. We briefly take a look at WFOMC problems, FO d-DNNF circuits, and QA with FOKC. Please refer to the work by Van den Broeck (2013) for details.

WFOMC Problem (Van den Broeck *et al.*, 2011) Notations are based on (function-free) first-order logic, where an *atom* $p(t_1, \ldots, t_n)$ consists of a predicate p of arity n and n terms t_i. Terms are either constants or variables, the latter comparable to logvars in parameterised models. A *literal* l is an atom a or its negation $\neg a$. From these basic units, one forms a *clause* c as a disjunction of literals $l_1 \vee \cdots \vee l_k$, assumed to be universally quantified. Clauses may be constrained by an (in)equality between two terms. A *theory* Δ is a finite set of clauses, denoting the conjunction of its clauses, i.e., the theory is in conjunctive normal form (CNF). An expression is an atom, a literal, a clause, or a theory. An expression is *ground* if its does not contain any variables. The Herbrand base $HB(\Delta)$ is the set of all possible ground atoms of the atoms in a theory Δ. A Herbrand *interpretation* I is a set of ground atoms, in which all atoms are assumed to be true, while all other atoms from a Herbrand base are assumed to be false. An interpretation satisfies a theory Δ, written as $I \models \Delta$, if is satisfies all clauses $c \in \Delta$. Such an interpretation is called a model for the theory, with $\mathcal{M}(\Delta) = \{I | I \models \Delta\}$ denoting the set of models.

For WFOMC, a theory Δ is augmented with a positive and a negative weight function w_T and w_F respectively, which assign weights to predicates in Δ. Then, a *WFOMC problem* for Δ, w_T, and w_F consists of computing

$$\sum_{I \in \mathcal{M}(\Delta)} \prod_{a \in I} w_T(pred(a)) \prod_{a \in HB(\Delta) \setminus I} w_F(pred(a))$$

where *pred* maps atoms to their predicate.

To transform parfactor models into WFOMC problems, one maps each input-output pair in a parfactor to a formula with corresponding weights. See (Van den Broeck, 2013) for a complete description of how to transform parfactor models into WFOMC problems. Consider parfactor $g_2 \in G_{ex}$ as an example. Assume that its potential function $\phi_2(Epid, Sick(X), Travel(X))$ maps the input $(true, true, true)$ to 7 and the remaining

inputs to 2. The PRVs in g_2 become atoms, with X becoming a variable. Then, ϕ_2 translates into two formulas,

$$\forall X : f_1(X) \Leftrightarrow epid \wedge sick(X) \wedge travel(X) \tag{3.4}$$
$$\forall X : f_2(X) \Leftrightarrow \neg epid \vee \neg sick(X) \vee \neg travel(X) \tag{3.5}$$

creating new predicates f_1 and f_2 for the weight functions that encode 7 and 2 (w_F maps to 1 by default):

$$w_T(f_1) = 7 \quad w_F(f_1) = 1$$
$$w_T(f_2) = 2 \quad w_F(f_2) = 1$$

Formula (3.4) describes the first input-output pair given above. Formula (3.5) catches the remaining cases that all have the same weight. If ϕ_2 maps to eight different potentials, eight formulas are necessary to encode the same model. The two formulas are not in CNF, but can be transformed by replacing $\Leftrightarrow$ with corresponding implications, which in turn can be represented using disjunctions. Then, the expression can be turned into CNF by expand the conjunctions accordingly.

Compilation The way Δ is defined with being in CNF and negations appearing directly in front of atoms, Δ may already be in FO d-DNNF. But, one may want to use other logical connectors than $\vee$ and $\wedge$ in clauses. Therefore, FOKC needs to convert Δ to be in FO d-DNNF such that the following holds: (i) Negations appear directly in front of the atoms. (ii) All conjunctions are decomposable (all pairs of conjuncts independent). (iii) All disjunctions are deterministic (only one disjunct true at a time). In the above example of turning g_2 into two formulas as given in (3.4) and (3.5), the set of clauses does not adhere to the definition of a theory given before. With these two formulas as input, FOKC converts them to fulfil the FO d-DNNF, which leads to a theory Δ_2 consisting of the following eight clauses:

$$\begin{array}{lll} epid \vee f_2(X) & sick(X) \vee f_2(X) & travel(X) \vee f_2(X) \\ epid \vee \neg f_1(X) & sick(X) \vee \neg f_1(X) & travel(X) \vee \neg f_1(X) \end{array}$$
$$\neg epid \vee \neg sick(X) \vee \neg travel(X) \vee \neg f_2(X)$$
$$\neg epid \vee \neg sick(X) \vee \neg travel(X) \vee f_1(X) \tag{3.6}$$

Given the constants of the parameterised model, an interpretation of Δ_2 is given by

$$\{epid, sick(alice), sick(bob), sick(eve),$$
$$travel(alice), travel(bob), travel(eve), f_1(alice), f_1(bob), f_1(eve)\} \tag{3.7}$$

with the remaining ground atoms $f_2(alice)$, $f_2(bob)$, and $f_2(eve)$ in the Herbrand base of Δ_2 being false. The smallest interpretation is $\{f_2(alice), f_2(bob), f_2(eve)\}$.

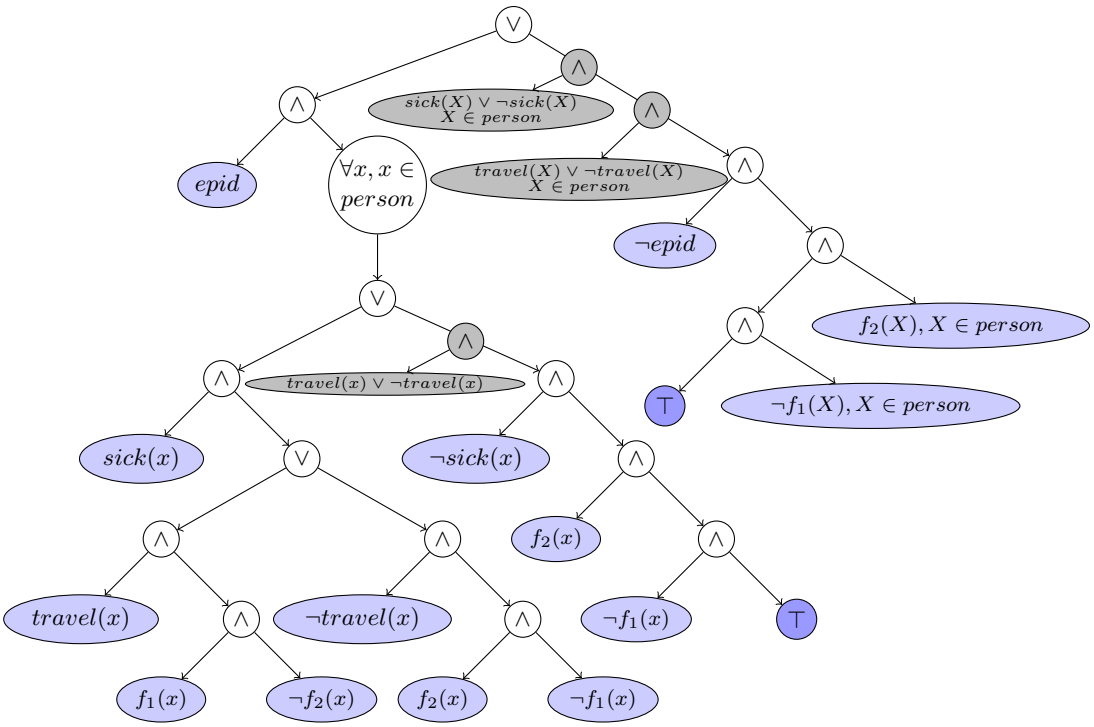


Figure 3.5: FO d-DNNF circuit (grey nodes appear after smoothing)

An *FO d-DNNF circuit* represents such a theory as a directed acyclic graph. As with a propositional circuit, inner nodes are labelled with $\vee$ and $\wedge$. Additionally, set-disjunction and set-conjunction represent isomorphic parts in Δ, explicitly representing repeated structures. Figure 3.5 shows an FO d-DNNF circuit for Δ_2 (ignore grey nodes for now). Leaf nodes contain atoms from the clauses, with atoms of newly created predicates (e.g., f_1 and $\neg f_1$) getting corresponding weights (7 and 1) associated. Looking at the root disjunction, the two subtrees are deterministic. E.g., the left subtree has *epid* as a node of the following conjunction, while the right subtree has $\neg$*epid* as a node, which are mutually exclusive. The other two disjunction have mutually exclusive assignments for $sick(x)$ and $travel(x)$ respectively. If $\neg$*epid* holds, then the last two clauses of Δ_2 are satisfied. To satisfy $epid \vee f_2(X)$ and $epid \vee \neg f_1(X)$, f_2 and $\neg f_1$ have to hold, which makes the remaining clauses satisfied as well. Together, $\neg epid$, f_2, and $\neg f_1$ make up the right subtree of the root. The left subtree becomes true under the interpretation in (3.7). The subtree also contains a set-conjunction for all groundings of X, denoting that the subtree starting in the set-conjunction is identical for each grounding of X. Before one can propagate weights, the circuit needs to be smoothed, i.e., in each branch need to occur the same set of predicates. For the circuit in Fig. 3.5, smoothing leads to the grey nodes to appear. The labels of the grey nodes anticipate weight calculations.

Algorithm 2 First-order Knowledge Compilation

procedure FOKC(Model G, Queries $\{Q_j\}_{j=1}^m$, Evidence $\mathbf{E}$)
  Reduce G to WFOMC problem with Δ, w_T, w_F
  Compile a circuit $\mathcal{C}_e$ for Δ, $\mathbf{E}$
  for each query Q_j **do**
    Compile a circuit $\mathcal{C}_{qe}$ for Δ, Q_j, $\mathbf{E}$
    Compute $P(Q_j|\mathbf{E})$ through model counts in $\mathcal{C}_{qe}, \mathcal{C}_e$ ▷ Expression (3.8)

Propagating weights in a FO d-DNNF circuit compares to propagating weights in a propositional d-DNNF circuit, with $\vee$ and $\wedge$ being replaced by $+$ and $\cdot$, respectively. The weight of leaf nodes containing a variable is exponentiated to the power of the groundings of its variables. At inner nodes of set-conjunctions, the model count of the child is exponentiated to the power of the number of interchangeable conjuncts being represented. At inner nodes of set-disjunctions, the children can be grouped into subsets of equal size, where each child has the same model count. Thus, one has to compute a model count for each subset. Propagating weights from the leaves to the root in Fig. 3.5 leads to a model count of 2709, using 1 as the neutral element for the model counts of predicate leaves. Next, we look at how FOKC answers queries.

Query Answering Algorithm 2 shows QA with FOKC for input model G, a set of query randvars $\{Q_i\}_{i=1}^m$, and evidence $\mathbf{E}$. FOKC starts with transforming G into a WFOMC problem Δ with weight functions w_T and w_F. It compiles a circuit $\mathcal{C}_e$ for Δ including $\mathbf{E}$. Handling evidence requires FOKC to set up three types of nodes instead of one for a set of instances, one for instances with observation $true$, one for the instances with observation $false$ and one for remaining instances, which blows up the circuit size further but is necessary to handle evidence. For details regarding how FOKC exactly handles evidence, refer to Van den Broeck and Davis (2012). For each query Q_i, FOKC compiles a circuit $\mathcal{C}_{qe}$ for Δ including $\mathbf{E}$ and Q_i. It then computes

$$P(Q_i|\mathbf{E}) = \frac{WFOMC(\mathcal{C}_{qe}, w_T, w_F)}{WFOMC(\mathcal{C}_e, w_T, w_F)} \tag{3.8}$$

by propagating model counts in $\mathcal{C}_{qe}$ and $\mathcal{C}_e$ based on w_T and w_F. FOKC can reuse the denominator model count for all Q_i.

FOKC reuses model circuits and counts for different queries. Calculations within a circuit reuse counts for interchangeable instances like LVE does during lifted summing out. The next part introduces LJT, applying lifting to jtrees instead of circuits and using LVE in its calculations, to provide efficient repeated inference based on LVE.

Part I

The Lifted Junction Tree Algorithm

Chapter 4

The Lifted Junction Tree Algorithm

LVE as an algorithm for solving the QA problem provides a means for lifted inference to efficiently answer single queries. But, LVE unnecessarily repeats computations when presented with another query. LJT focusses on efficient repeated inference, meaning solving multiple QA problems, avoiding repetition of calculations. An important basis for LJT is the well-known propositional junction tree algorithm, JT, which focusses on repeated inference in propositional models. With LJT, we combine both ideas, the lifting concept as well as LVE to efficiently incorporate relational aspects of a model during computations and the jtree concept to efficiently provide answers to multiple queries.

The following paper introduced LJT, providing first definitions,

> Tanya Braun and Ralf Möller. Lifted Junction Tree Algorithm. In *Proceedings of KI 2016: Advances in Artificial Intelligence*, pages 30–42. Springer, 2016

which was extended regarding counting, evidence, and fusion, a new step in LJT, and then received a full specification and theoretical and empirical analysis (under review):

> 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 - Extended. In *Postproceedings of the 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning at the 26th International Joint Conference on Artificial Intelligence*, pages 54–72. Springer, 2018

> Tanya Braun and Ralf Möller. Avoiding Repetition in Repeated Inference on Probabilistic Relational Models: The Lifted Junction Tree Algorithm. Submitted

This part contains the first three contributions of this dissertation regarding FO jtrees, LJT as a QA algorithm, and first completeness and complexity results. Chapter 4 starts with an introduction to jtrees before defining FO jtrees, followed by a presentation of

LJT. The chapter ends with a description of how to prevent algorithm-induced groundings. Chapters 5 and 6 contribute a theoretical analysis and an empirical evaluation respectively. A conclusion wraps up this first part before we move on to the next part on more complex queries.

4.1 An Introduction to Junction Trees

LJT focusses on efficient repeated inference, meaning solving multiple different QA problems. An important basis for LJT is the well-known propositional junction tree algorithm, JT, by Lauritzen and Spiegelhalter (1988) for repeated inference in propositional models.

Using a small submodel would be preferable to restarting with a model G for each query. Exploiting that G encodes (conditional) independencies between randvars through its factorisation, one needs to find a submodel $G_i \subset G$, as small as possible, whose randvars are independent of the randvars in the remaining model given a set or sets of randvars $\mathbf{S}$, called separators. Separators separate G_i from the remaining model, i.e., the randvars in G_i are conditionally independent from the remaining randvars given the separators.

In other words, hidden behind each $\mathbf{S}$ lie factors that are no longer relevant for answering a query on G_i if $\mathbf{S}$ is given, i.e., answering a query for a randvar Q on G_i is possible. But without all $\mathbf{S}$ given, the remaining factors in $G \setminus G_i$ and their influences on the randvars in G_i are missing. To combine the missing factors, one can query for each $\mathbf{S}$ on the factors that lie hidden behind $\mathbf{S}$, using e.g., VE to answer such a query. G_i together with the answer to each queried $\mathbf{S}$ is sufficient to answer a query for Q.

To be able to reuse submodels and queries for $\mathbf{S}$ when given different queries, the idea is to partition G into n submodels once. Each submodel contains a set of randvars, which form a cluster $\mathbf{C}_i$, with the submodel G_i building a local model for $\mathbf{C}_i$. Then, the clusters are arranged into an acyclic graph, i.e., a jtree, s.t. if a randvar appears in two clusters, the randvar appears in every cluster on the path between the two. This graph arrangement highly influences the partitioning of G. Given such a graph, neighbouring clusters $\mathbf{C}_i, \mathbf{C}_j$ share randvars, which form a separator $\mathbf{S}_{ij}$ that renders $\mathbf{C}_i$ independent from all randvars that lie behind $\mathbf{C}_j$ and vice versa. After $\mathbf{C}_i$ has queried each neighbour for $\mathbf{S}_{ij}$, $\mathbf{C}_i$ can answer a query using the answers from the neighbours and G_i. At each $\mathbf{C}_i$, querying neighbours for $\mathbf{S}_{ij}$ recursively spawns queries over separators at neighbours. Instead of implementing these queries naively for each cluster, one may rearrange the queries using dynamic programming: Leaves are the first to provide an answer to their neighbour. From there on, clusters provide answers to neighbours further inward and then back outward. In JT, this scheme is called message passing where a message is the result of a query for a separator. JT calculates a message using VE to eliminate all randvars but the separator from its local model and received message from other neighbours.

Consider the jtree in Fig. 2.2 for the epidemic example where the $Sick_i$ and Nat_j randvars each appear with the *Epid* randvar in an own cluster. Each node in the repre-

sentation has a local model of factors whose arguments appear in the cluster. The local models of ϕ_1's and ϕ_2's partition the input model. The separator between all clusters is *Epid*. Consider the leaf cluster $\{Sick_1, Epid\}$ with ϕ_2 in its local model. Combining the remaining model behind *Epid*, i.e., all ϕ_1's and ϕ_2's into one factor, i.e., ϕ'', renders $\{Sick_1, Epid\}$ independent given ϕ''. Message passing arranges for each local model to also contain these ϕ''s and ϕ'''s, making the clusters independent from each other..

The jtree also shows that repeated structure in a model lead to duplicate nodes and identical messages, i.e., ϕ''s and ϕ'''s. Thus, LJT lifts jtrees to provide a compact representation of a parameterised model. As such, we use the lifting concept including LVE to efficiently incorporate relational aspects of a model in data structures and calculations and exploit the jtree concept to efficiently provide answers to multiple queries. Before specifying LJT, we define FO jtrees as the underlying data structure.

4.2 First-order Jtrees

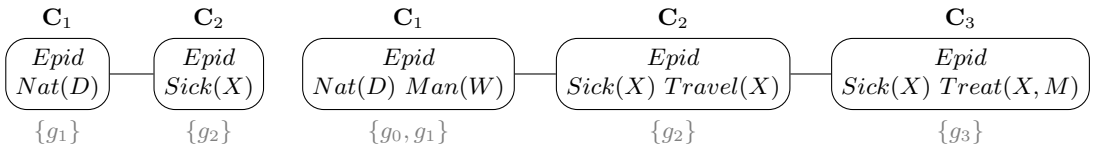
In a first-order version of a jtree, the nodes are parameterised clusters, called parclusters for short. Parclusters are sets of PRVs, which have their origin in the clusters of an FO dtree. We define parclusters and FO jtrees and then look at an FO jtree for G_{ex}.

Definition 4.2.1 (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 3.1.3 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 at least one property. 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.

Empty separators identify independent parts of a model. An FO jtree with empty separators practically dissolves into a forest of FO jtrees, where LJT can work individually on each FO jtree in the forest.

Example 4.2.1 (FO Jtrees)**.** Figure 4.1a shows an FO jtree for the parameterised model of the epidemic example in Chapter 2, which is a much more compact representation than the jtree in Fig. 2.2. Instead of M duplicate clusters with Nat_j and *Epid* as well as N



(a) Example in Chapter 2 (b) Model G_{ex}

Figure 4.1: FO jtrees with local models in grey

duplicate clusters with $Sick_i$ and $Epid$ in the jtree, the FO jtree has two parclusters, involving the two logvars X and D with domains $\mathcal{D}(X) = \{x_i\}_{i=1}^N$ and $\mathcal{D}(D) = \{d_j\}_{j=1}^M$:

$$\mathbf{C}_1 = \forall d \in \{d_j\}_{j=1}^M : \{Nat(d), Epid\}_{|\top} = \{Nat(D), Epid\},$$
$$\mathbf{C}_2 = \forall x \in \{x_i\}_{i=1}^N : \{Sick(x), Epid\}_{|\top} = \{Sick(X), Epid\}.$$

Separator $\mathbf{S}_{12}$ contains $Epid$ as the shared PRV of the two parclusters. For the extended model G_{ex}, Fig. 4.1b shows an FO jtree with three parclusters,

$$\begin{aligned}
\mathbf{C}_1 &= \forall (d, w) \in \mathcal{D}(D) \times \mathcal{D}(W) &&: \{E, Nat(d), Man(w)\}_{|\top} &&= \{E, Nat(D), Man(W)\},\\
\mathbf{C}_2 &= \forall (x) \in \mathcal{D}(X) &&: \{E, S(x), Travel(x)\}_{|\top} &&= \{E, S(X), Travel(X)\},\\
\mathbf{C}_3 &= \forall (x, m) \in \mathcal{D}(X) \times \mathcal{D}(M) &&: \{E, S(x), Treat(x, m)\}_{|\top} &&= \{E, S(X), Treat(X, M)\}
\end{aligned}$$

where E stands for $Epid$ and S for $Sick$. The separators are $\mathbf{S}_{12} = \{Epid\}$ and $\mathbf{S}_{23} = \{Epid, Sick(X)\}_{|\top} = \{Epid, Sick(X)\}$. Each parcluster is a set of PRVs from G_{ex} and the parfactor arguments appear in some node. PRVs that appear more than once, i.e., $Epid$ and $Sick(X)$, appear in each parcluster on the path between appearances. The given FO jtree is minimal as removing any PRV from any of the parclusters leads to either g_1, g_2, or g_3 to no longer have its arguments appear in a parcluster. The local models $G_1 = \{g_0, g_1\}$, $G_2 = \{g_2\}$, and $G_3 = \{g_3\}$ partition G_{ex}.

With the lifted version of a jtree defined, we present LJT itself including how to construct an FO jtree, pass messages, and answer queries.

4.3 Algorithm Description

Algorithm 3 shows LJT, which takes a model G, a set of queries $\{Q_k\}_{k=1}^m$, and evidence $\mathbf{E}$ as inputs. I.e., LJT solves the QA problem for the queries $P(Q_k|\mathbf{E})$ for each (ground) query term Q_k. Like JT, LJT has the steps construction, evidence entering, message passing, and query answering. As we will see, a straight-forward translation of JT into LJT has the effect of unnecessary groundings during message passing for certain models. Thus, the complete specification of LJT includes an operation called fusion after the construction step to remedy the effect, which we present at the end of this chapter.

Algorithm 3 Lifted Junction Tree Algorithm

procedure LJT(Model G, Query terms $\{Q_k\}_{k=1}^m$, Evidence $\mathbf{E}$)
  Construct an FO jtree $J = (V, E)$ for G
  Enter $\mathbf{E}$ into J
  Pass messages on J ▷ LVE as subroutine
  for each $Q_k \in \{Q_k\}_{k=1}^m$ **do**
    Find $\mathbf{C}_i \in V$ s.t. $Q_k \in \mathbf{C}_i$
    $G' \leftarrow G_i \cup \bigcup_{j \in nbs(i)} m_{ji}$
    LVE(G', Q_k, $\emptyset$) ▷ Output or store result

Construction Darwiche (2001) shows that the clusters of the dtree nodes form a jtree, which, however, is not necessarily minimal. The connection also holds for FO dtrees and FO jtrees (see Chapter 5 for a proof), which LJT uses to build an FO jtree for model G. In a jtree that has been built from a dtree, there are non-maximal clusters, i.e., clusters are subsets of each other. But, there are no clusters that are larger than necessary (Darwiche, 2009). First, we define how to convert FO dtree clusters into parclusters, forming an FO jtree. Second, we specify how to minimise an FO jtree. A minimal (FO) jtree ensures space efficiency as well as runtime efficiency during message passing.

Definition 4.3.1 (Conversion)**.** Let $\mathbf{B}$ denote a cluster associated with an FO dtree node T. If T is a descendant of a DPG node $(\mathbf{X}, \mathbf{x}, C)$, then $\mathbf{B}$ is the cluster of T after applying the inverse θ^{-1} of the substitution $\theta = \{X_i \to x_i\}_{i=1}^k$, mapping $\mathbf{x}$ back onto $\mathbf{X}$. θ^{-1} also applies to leaf parfactors. Then, we define the following conversions for building a parcluster $\mathbf{C}_i = \mathbf{A}_{|C}$ with a local model G_i:

- Let $\mathbf{B}$ be the cluster of a DPG node $(\mathbf{Y}, \mathbf{y}, D)$. Then, $\mathbf{B}$ is converted into $\mathbf{C}_i$ by setting (i) $\mathbf{A} = \mathbf{B}$, (ii) $C = D$, and (iii) $G_i = \emptyset$.
- Let $\mathbf{B}$ be the cluster of a VE node T. Then, $\mathbf{B}$ is converted into $\mathbf{C}_i$ by setting (i) $\mathbf{A} = \mathbf{B}$, (ii) $C = \emptyset$, and (iii) $G_i = \emptyset$.
- Let $\mathbf{B}$ be the cluster of a leaf node with parfactor g. Then, $\mathbf{B}$ is converted into $\mathbf{C}_i$ by setting (i) $\mathbf{A} = \mathbf{B}$, (ii) $C = \emptyset$, and (iii) $G_i = \{g\}$.

Example 4.3.1 (Non-minimal FO jtree)**.** Figure 4.2 depicts the FO jtree after converting the clusters of the FO dtree in Fig. 3.4 into parclusters. All parclusters originating from DPG nodes may include DPG logvars in their constraints that do not occur in their PRVs. The parclusters from the leaf nodes have non-empty local models. The FO jtree is valid since all parclusters are sets of PRVs from G_{ex}, all parfactor arguments appear in some node, and all PRVs contained in several nodes appear on all paths between them.

The FO jtree in Fig. 4.2 is not minimal as there exist parclusters that are subsets of other parclusters, e.g., the former root cluster. Non-minimality makes message passing

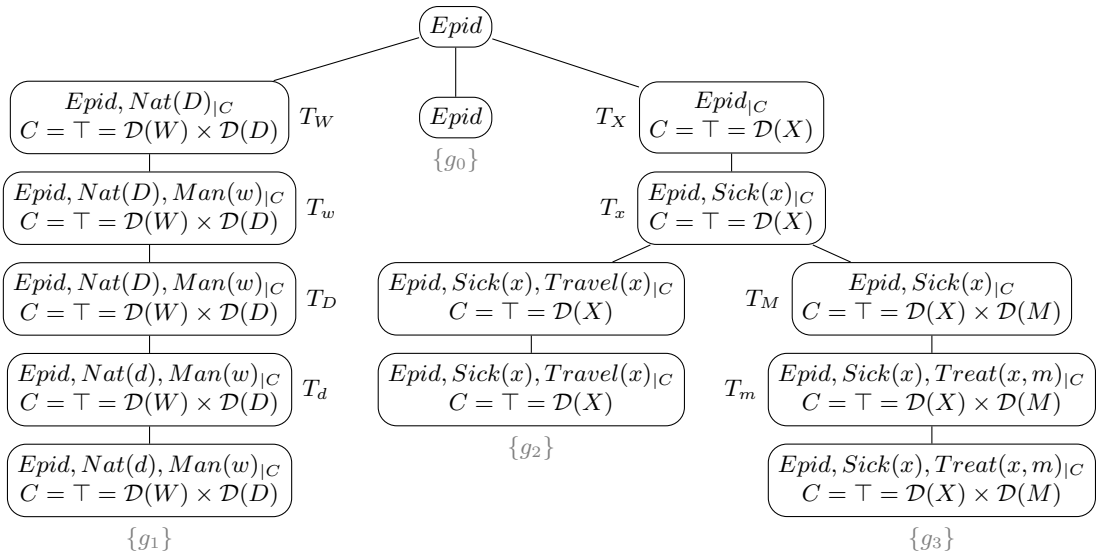


Figure 4.2: Non-minimal FO jtree for G_{ex} with local models in grey (empty models omitted). T labels denote from which node in Fig. 3.4 a parcluster originates.

inefficient and requires unnecessary memory. Therefore, LJT minimises an FO jtree by merging adjacent nodes until no parcluster is a subset of any other parcluster, which was the only violation of minimality. To keep parclusters as small as possible for query answering, LJT does not merge parclusters that are not subsets. Doing so would lead to parclusters larger than necessary. After merging, the FO jtree is minimal. Before we illustrate how to minimise the FO jtree in Fig. 4.2, we define merging.

Definition 4.3.2 (Merging)**.** Parclusters $\mathbf{C}_i$ and $\mathbf{C}_j$ with models G_i and G_j in an FO jtree (V, E) *merge* iff $\mathbf{C}_i \subseteq \mathbf{C}_j \vee \mathbf{C}_j \subseteq \mathbf{C}_i$ including their constraints. The merged parcluster $\mathbf{C}_k$ and its model G_k are given by $\mathbf{C}_k = \mathbf{C}_i \cup \mathbf{C}_j$ and $G_k = G_i \cup G_j$. $\mathbf{C}_k$ replaces parclusters $\mathbf{C}_i$ and $\mathbf{C}_j$ in V. $\mathbf{C}_k$ takes over all neighbours of $\mathbf{C}_i$ and $\mathbf{C}_j$ in E.

Example 4.3.2 (Minimal FO jtree)**.** Minimising the FO jtree in Fig. 4.2 leads to the FO jtree in Fig. 4.1b. LJT merges the parclusters of the former root with the former left child subtree. The resulting parcluster $\{Epid, Nat(D), Man(W)\}$ with local model $\{g_1\}$ is mergeable with the parcluster with local model $\{g_0\}$. The merged parcluster corresponds to parcluster $\mathbf{C}_1$ in Fig. 4.1b. LJT does not continue merging with the parcluster labelled T_X as neither constraint is a subset of the other. LJT merges T_X with the parcluster labelled T_x. It then continues down the left branch merging all parclusters including the leaf. The resulting node corresponds to parcluster $\mathbf{C}_2$ in Fig. 4.1b. LJT does not merge $\mathbf{C}_2$ and T_M as the constraint of T_M is a superset, while its PRVs are a subset of $\mathbf{C}_2$. LJT merges the parclusters labelled T_M, T_m, and the leaf into one

parcluster, yielding parcluster $\mathbf{C}_3$ in Fig. 4.1b. Instead of merging the parcluster of T_X and T_x with the left branch, LJT could have chosen the right branch, which results in the same parclusters but with $\mathbf{C}_3$ as the central parcluster and $\mathbf{C}_2$ as a leaf.

After construction, LJT has assembled an FO jtree with fitting separators and local models that partition the input model G. Before passing messages, LJT handles evidence to pass messages that combine missing parfactors as well as evidence.

Evidence Entering LJT next works on the evidence given in a query $P(Q|\{E_j = e_j\}_{j=1}^m$, which has been encoded in a set of evidence parfactors $\mathbf{E}$. Entering evidence consists of making evidence parfactors available at parclusters and absorbing evidence. A local model G_i absorbs evidence $g_e = \phi_e(R(\mathbf{X}))_{|C_e}$ at a parcluster $\mathbf{C}_i$ iff

$$gr(R(\mathbf{X})_{|C_e}) \cap gr(\mathbf{C}_i) \neq \emptyset. \tag{4.1}$$

As described in Section 3.2, LVE tests each parfactor in G against each evidence parfactor for absorption. Unlike LVE, LJT avoids testing each parfactor in G using the running intersection property of an FO jtree. To distribute g_E, LJT finds a parcluster $\mathbf{C}_i$ that meets Expression (4.1). G_i absorbs g_E and LJT checks for each separator if

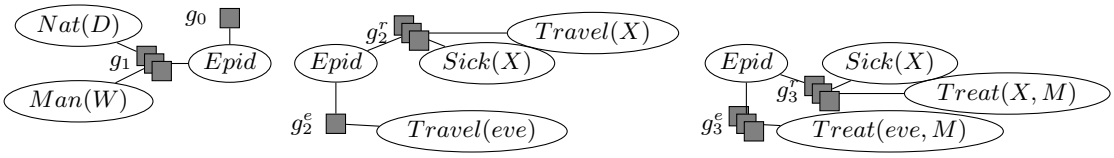
$$rv(g_E) \cap \mathbf{S}_{ij} \neq \emptyset. \tag{4.2}$$

If Expression (4.2) is true for j, distribution continues at $\mathbf{C}_j$. Otherwise, LJT stops distributing g_E in the subgraph behind j as $R(\mathbf{X})$ can no longer appear due to the running intersection property. Let us add the evidence from $P(Treat(eve, injection)|sick(eve))$.

Example 4.3.3 (Evidence in LJT)**.** The evidence is $sick(eve)$, which appears in $\mathbf{C}_2$ and $\mathbf{C}_3$ as shown in Fig. 4.1b. The local model at $\mathbf{C}_1$ is unaffected, shown in Fig. 4.3a. In $\mathbf{C}_2$, LJT splits g_2 into $g_2^e = \phi_2(Epid, Sick(eve), Travel(eve))$ for the evidence part and $g_2^r = \phi_2(Epid, Sick(X), Travel(X))_{|C_2^r}$, C_2^r containing *alice* and *bob*, for the remainder. g_2^e absorbs $sick(eve)$, yielding $g_2^e = \phi_2'(Epid, Travel(eve))$. G_2 is now $\{g_2^r, g_2^e\}$ as shown in Fig. 4.3b. Absorption in $\mathbf{C}_3$ proceeds analogously: LJT splits g_3 into $g_3^e = \phi_3(Epid, Sick(eve), Treat(eve, M))$ and $g_3^r = \phi_3(Epid, Sick(X), Treat(X, M))|C_3^r$, C_3^r containing (X, M) tuples for *alice* and *bob*. g_3^e absorbs $sick(eve)$, resulting in $g_3^e = \phi_3'(Epid, Treat(eve, M))$. G_3 is now $\{g_3^r, g_3^e\}$ as shown in Fig. 4.3c.

Now, each local model G_i also contains evidence about its own PRVs. The next step is message passing which makes $G \setminus G_i$ available at each parcluster.

Message Passing A message is the result of a query over a separator, which each parcluster asks each neighbour. A message m_{ij} from parcluster $\mathbf{C}_i$ to parcluster $\mathbf{C}_j$ combines for j all missing parfactors and evidence that lie behind $\mathbf{S}_{ij}$, making $\mathbf{C}_j$ independent from the submodel behind $\mathbf{S}_{ij}$. LJT uses LVE to compute messages, skipping the last step before returning the result (multiplying and normalising) as the message is only a partial combination of the model and as such does not require normalisation.



(a) G_1 (b) G_2 (c) G_3

Figure 4.3: Parfactor graphs of local models after absorbing evidence

Definition 4.3.3 (Message)**.** A *message* m_{ij} from parcluster $\mathbf{C}_i$ with local model G_i to parcluster $\mathbf{C}_j$ is a set of parfactors, each with a subset of $\mathbf{S}_{ij}$ as arguments, which $\mathbf{C}_j$ stores. LJT computes m_{ij} by passing to LVE a query over $\mathbf{S}_{ij}$ and a model $G' = G_i \cup \bigcup_{k \in nbs(i), k \neq j} m_{ki}$, i.e., LVE($G'$, $\mathbf{S}_{ij}$, $\emptyset$). LVE then eliminates $\mathbf{C}_i \setminus \mathbf{S}_{ij}$ from G'.

As parcluster $\mathbf{C}_i$ can only send a complete message to neighbour $\mathbf{C}_j$ after $\mathbf{C}_i$ has received all messages from other neighbours, sending messages is arranged in an order such that each parcluster sends a message to a neighbour once all necessary messages arrived. The scheme that LJT uses has been introduced by Shafer and Shenoy (1990) for propositional jtrees. Messages are sent based on two conditions, leading to basically an inward and an outward pass:

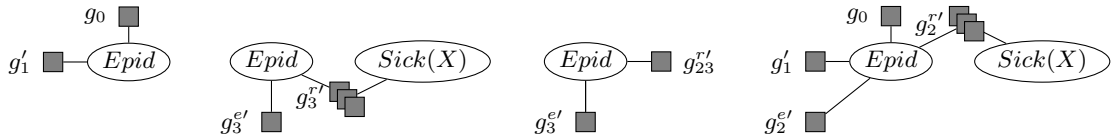
(1) When a parcluster $\mathbf{C}_i$ has received messages from all neighbours but $\mathbf{C}_j$, $\mathbf{C}_i$ sends a message to $\mathbf{C}_j$.

(2) When a parcluster $\mathbf{C}_i$ has received a message from its remaining neighbour $\mathbf{C}_j$, $\mathbf{C}_i$ sends messages to all other neighbours.

Condition (1) is automatically true at leaf parclusters. Condition (1) triggers sending messages from the periphery of the FO jtree inward towards the centre. At the centre, Condition (2) becomes true for a first parcluster, triggering outward sending of messages. Technically, whenever a condition is true, message calculation and sending may commence, which makes message passing highly parallelisable. Doing so, a parcluster $\mathbf{C}_i$ may receive all messages of its neighbours at once, meaning that $\mathbf{C}_i$ immediately calculates messages for all neighbours. Let us look at messages in the FO jtree for G_{ex}.

Example 4.3.4 (Message passing)**.** Four messages flow in the FO jtree in Fig. 4.1b, from parclusters $\mathbf{C}_1$ and $\mathbf{C}_3$ to parcluster $\mathbf{C}_2$ and back. Messages between $\mathbf{C}_1$ and $\mathbf{C}_2$ have the argument *Epid* and between $\mathbf{C}_2$ and $\mathbf{C}_3$ the arguments *Epid* and $Sick(X)$.

Condition 1 is true at $\mathbf{C}_1$ and $\mathbf{C}_3$, leading to message calculations for neighbour $\mathbf{C}_2$. For message m_{12}, LJT eliminates $\mathbf{C}_1 \setminus \mathbf{S}_{12} = \{Nat(D), Man(W)\}$ with LVE from $G' = G_1 = \{g_0, g_1\}$. The PRVs $Nat(D)$ and $Man(W)$ appear in g_1. LVE counts D, sums out $Man(W)$, and sums out $\#_D[Nat(D)]$, resulting in a parfactor $g'_1 = \phi'_1(Epid)$. Parfactor g_0 has only *Epid* as argument, which LVE does not eliminate. As LVE does not multiply



(a) m_{12} (b) m_{32} (c) m_{21} (d) m_{23}

Figure 4.4: Parfactor graphs of the messages

and normalise the parfactors in its model for messages, the message is $m_{12} = \{g_0, g_1'\}$, which is shown in Fig. 4.4a. For message m_{32}, LJT eliminates $\mathbf{C}_2 \setminus \mathbf{S}_{23} = \{Treat(X, M)\}$ from $G' = G_3 = \{g_3^r, g_3^e\}$ with LVE. In g_3^r, LVE eliminates $Treat(X, M)$ leading to a parfactor $g_3^{r'} = \phi_3'(Epid, Sick(X))$. Eliminating $Treat(eve, M)$ in g_3^e leads to a parfactor $g_3^{e'} = \phi_3''(Epid)$. The message is $m_{32} = \{g_3^{r'}, g_3^{e'}\}$ as shown in Fig. 4.4b.

In a parallel execution, $\mathbf{C}_2$ receives m_{12} and m_{32}, which makes Condition 2 true, and starts calculating messages for its neighbours. For message m_{21} back to node $\mathbf{C}_1$, LJT eliminates $\mathbf{C}_2 \setminus \mathbf{S}_{12} = \{Sick(X), Travel(X)\}$ from $G' = G_2 \cup m_{32}$ with LVE. LVE eliminates $Travel(X)$ from g_2^r, multiplies the result with $g_3^{r'}$ from m_{32}, and eliminates $Sick(X)$. The result $g_{23}^{r'}$ as well as $g_3^{e'}$ from m_{32} make up m_{21}. For message m_{23}, LJT eliminates $\mathbf{C}_2 \setminus \mathbf{S}_{23} = \{Travel(X)\}$ from $G' = G_2 \cup m_{12}$, eliminating $Travel(X)$ from g_2^r and $Travel(eve)$ from g_2^e, with both results appearing in m_{23}, depicted in Fig. 4.4c. Since $m_{12} = \{g_0, g_1''\}$ consists of parfactors with argument $Epid$, which LJT does not eliminate, m_{12} appears unchanged in m_{23}. Fig. 4.4d shows a parfactor graph of m_{23}.

In a non-parallel execution, LJT may deliver m_{12} first, which means $\mathbf{C}_2$ may now calculate message m_{23}, which LJT delivers to $\mathbf{C}_3$. After m_{32} arrives at $\mathbf{C}_2$, LJT calculates m_{21} for $\mathbf{C}_1$. The calculations are identical, only the order of sending messages changes.

At this point, each parcluster has received messages from each neighbour, which makes each parcluster independent from all other parclusters. Based on the messages received and its own local model, each parcluster can answer queries about its parcluster PRVs. Before moving on to query answering, let us consider how messages influence LJT.

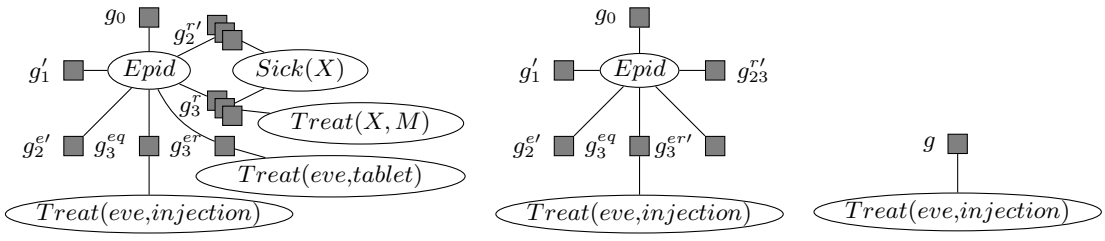
During construction, LJT does not count logvars identifiable in an FO dtree. The reason lies in LJT not being able to always determine beforehand if a count conversion is reasonable for message calculation. A count conversion may be superfluous and thus unnecessarily enlarge a parfactor. Consider a scenario in which a parcluster $\mathbf{C}_i = \{Nat(D), Man(W)\}$ has a separator $\mathbf{S}_{ij} = \{Man(W)\}$ with some neighbour $\mathbf{C}_j$. The local model G_i contains a parfactor $\phi(Nat(D), Man(W))$ with both logvars countable. To compute message m_{ij}, LJT needs to eliminates $Nat(D)$ by letting LVE count W for summing out $Nat(D)$. Assume that the FO dtree used for constructing the underlying FO jtree shows that D is countable. If had LJT counted D leading to a CRV $\#_D[Nat(D)]$ during construction, the following scenario would arise: $\mathbf{C}_i = \{\#_D[Nat(D)], Man(W)\}$, $\mathbf{S}_{ij} = \{Man(W)\}$, and $G_i = \{\phi(\#_D[Nat(D)], Man(W))\}$. In this scenario, LJT would

have to eliminate $\#_D[Nat(D)]$ but could not do so as $\#_D[Nat(D)]$ would not contain all logvars of $\phi(\#_D[Nat(D)], Man(W))$, i.e., W would not appear in $\#_D[Nat(D)]$. Thus, LVE would count W to sum out $\#_D[Nat(D)]$, making counting D superfluous.

Another effect of message passing is that the *heuristics* LVE employs for choosing the next operator no longer works for LJT. Again consider the above scenario, in which $\mathbf{C}_i = \{Nat(D), Man(W)\}$, $\mathbf{S}_{ij} = \{Man(W)\}$, and $G_i = \{\phi(Nat(D), Man(W))\}$. For computing message m_{ij}, LJT needs to eliminate $Nat(D)$. Only after counting W, LVE can sum out $Nat(D)$. But, assume that W has 50 values while D has 10. Based on the LVE heuristics, LVE would count D as the intermediate result would be smaller. Afterwards, LVE still cannot sum out $\#_D[Nat(D)]$ as it does not contain the remaining logvar W. LVE would then count W to sum out $\#_D[Nat(D)]$, which makes counting D superfluous, and unnecessarily blows up the representation. For LJT, we require the heuristics to consider the PRVs in a separator. LJT only selects count operations for PRVs in a separator if the sole other operator applicable is grounding.

Query Answering LJT starts processing queries, which are provided in advance or on the fly in online QA. To answer a query with query term Q_k, LJT finds a parcluster $\mathbf{C}_i$ s.t. $Q_k \in \mathbf{C}_i$. LJT builds a submodel G' consisting of the local model G_i and all messages received at $\mathbf{C}_i$. Using LVE, LJT eliminates all non-query terms in G'. Let us look at query $P(Treat(eve, injection)|sick(eve))$ with query term $Treat(eve, injection)$.

Example 4.3.5 (Query answering)**.** Continuing with Example 4.3.4, LJT basically answers the query $P(Treat(eve, injection))$ as the local models and messages have already absorbed the evidence. Parcluster $\mathbf{C}_3$ covers $Treat(eve, injection)$. The submodel G' is a union of $G_3 = \{g_3^e, g_3^r\}$ and message $m_{23} = \{g_0, g_1', g_2^{e\prime}, g_2^{r\prime}\}$. LJT provides LVE with G' as the input model and $Treat(eve, injection)$ as the query term. LVE shatters G' on $Treat(eve, injection)$, Fig. 4.5a shows the result. Then, LVE eliminates all terms in G' but $Treat(eve, injection)$: LVE first sums out $Treat(eve, tablet)$ from g_3^{er}, resulting in $g_3^{er\prime} = \phi_3'(Epid)$, and $Treat(X, M)$ from g_3^r, resulting in $g_3^{r\prime} = \phi_3''(Epid, Sick(X))$. From the product of $g_2^{r\prime}$ and $g_3^{r\prime}$, LVE sums out $Sick(X)$, resulting in a parfactor $g_{23}^{r\prime}$ with



(a) After shattering (b) Before eliminating *Epid* (c) Result

Figure 4.5: Parfactor graphs for $G' = G_3 \cup m_{23}$ during QA

argument *Epid*. Figure 4.5b shows the remaining parfactors. To sum out *Epid*, LVE multiplies all parfactors into one. Then, LVE sums out *Epid*, resulting in a parfactor g, shown in Fig. 4.5c, which holds the queried distribution after normalising.

When LJT answers the query on $\mathbf{C}_3$, it only has to eliminate the non-query terms in $\mathbf{C}_3$, while LVE, working on G, has to eliminate all non-query terms in G. Specifically, LVE eliminates $Nat(D)$, $Man(W)$, and $Travel(X)$, which are already eliminated in $\mathbf{C}_3$. For the next query, LJT again works on a parcluster, saving eliminations and thus, runtime, as shown in the evaluation in Chapter 6 as well.

The given example works nicely. But, as already indicated, there exist models that lead to an FO jtree that causes unnecessary groundings during message passing. Next, we look at when unnecessary groundings occur and discuss how to avoid them.

4.4 Avoiding Unnecessary Groundings: Fusion

LJT as discussed so far is a rather straightforward translation of JT. But, let us look at an example model that LVE solves without grounding, while LJT grounds a logvar.

Example 4.4.1 (Unnecessary groundings)**.** Consider the FO jtree in Fig. 4.6 for a model identical to G_{ex} except for a new logvar E in the PRVs *Epid*, $Sick(X)$, and $Treat(X, M)$. E encodes different epidemics, e.g., $e_1, \ldots, e_5$. Assume that no evidence is provided. Given a query term, e.g., $Treat(eve, injection, e_1)$, LVE splits g'_3 on the query term. Then, LVE eliminates all non-query terms in a similar way as described in Example 3.2.5.

LJT would pass messages. For message m'_{12} from $\mathbf{C}'_1$ to $\mathbf{C}'_2$, LVE would count D and E and sum out $Man(W)$ and $\#_D[Nat(D)]$, resulting in a message with parfactors g_0 and $\phi(\#_E[Epid(E)])$. For message m'_{23}, LVE would need to sum out $Travel(X)$, which does not include all logvars of g'_2. A count conversion would not apply since both logvars appear in two PRVs. Thus, LVE would ground E to sum out $Travel(X)$. The result would be a parfactor $\phi(Epid(e_1), \ldots, Epid(e_5), Sick(X, e_1), \ldots, Sick(X, e_5))$, which, together with g_0 and $\phi(\#_E[Epid(E)])$, would make up the message to $\mathbf{C}'_3$.

LVE does not need to ground while LJT produces groundings due to message passing. The reason lies in the separators impeding a reasonable elimination order. In the above example, LVE eliminates $Sick(X, E)$ before eliminating $Epid(E)$ or $Travel(X)$. But

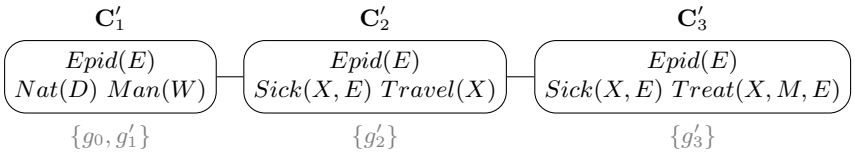


Figure 4.6: Example FO jtree with unnecessary groundings during message passing

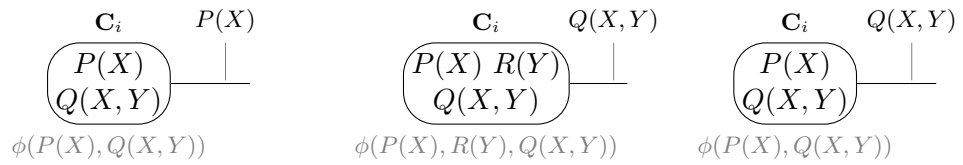
LJT needs to eliminate $Travel(X)$ before $Sick(X, E)$, which causes the grounding. To avoid those algorithm-induced groundings, we first investigate when groundings to occur. The idea is then to build a test to identify possible groundings and lastly, adjust the FO jtree structure after construction such that the groundings do not occur.

Conditions for Unnecessary Groundings This paragraph examines when unnecessary groundings occur. A lifted solution to a query means that an inference algorithm computes an answer without grounding a part of the model. Not all models have a lifted solution, i.e., they only have a solution for which groundings are unavoidable. But additionally to unavoidable groundings, LJT without fusion may require avoidable groundings due to message calculations. Grounding a logvar is expensive and causes further groundings when at the neighbouring parcluster, the grounded PRV of the message and the PRV in the parcluster need to be eliminated. For illustration purposes, we use the examples depicted in Figs. 4.7 and 4.8. Each example is a parcluster with two PRVs and an edge with a separator consisting of one of the PRVs. The local model has one parfactor with both PRVs as inputs. We use names $\mathbf{L} = \{X, Y, Z\}$ and $\mathbf{R} = \{P, Q, R\}$ to build PRVs.

As mentioned before, we are only interested in those groundings that occur due to message passing, when a separator enforces an elimination order that leads to groundings. In such a case, preconditions for summing out a variable are not fulfilled leading LVE to ground a logvar when LVE calculates a message. To recap, lifted summing out of a PRV A has three preconditions (Taghipour *et al.*, 2013c): (i) A may only appear in one parfactor. (ii) A must contain all logvars of the parfactor. (iii) The logvars that are only in A, $\mathbf{X}^{excl}$, need to be count-normalised w.r.t. the remaining logvars in the parfactor. Formally, for message m_{ij} from parcluster $\mathbf{C}_i$ with model G_i to parcluster $\mathbf{C}_j$ given separator $\mathbf{S}_{ij}$, LJT eliminates $\mathbf{A}_{ij}^{E} := \mathbf{C}_i \setminus \mathbf{S}_{ij}$ from $G' := G_i \bigcup_{k \in nbs(i), k \neq j} m_{ki}$. To eliminate $A \in \mathbf{A}_{ij}^{E}$ from G', LJT multiplies all parfactors $g \in G'$ that include A into a parfactor $g^A = \phi(\mathcal{A}^A)|C^A$. Let $\mathbf{S}_{ij}^{A} := \mathbf{S}_{ij} \cap \mathcal{A}^A$ be the set of PRVs in the separator that occur in g^A. Then, the following may happen w.r.t. the preconditions mentioned above. After the multiplication, g^A is the only parfactor containing A, i.e., Precondition (i) is fulfilled. Precondition (ii) may be violated whenever a PRV $S \in \mathbf{S}_{ij}^{A}$ contains more logvars than A, i.e., $lv(S) \supset lv(A)$. Precondition (iii) holds in all cases since either A contains completely different or more logvars than the PRVs in $\mathbf{S}_{ij}^{A}$ and the logvars are count-normalised based on the FO dtree used for FO jtree construction. Or, A contains fewer logvars than the PRVs in $\mathbf{S}_{ij}^{A}$ meaning that no logvars are eliminated, i.e., $\mathbf{X}^{excl}$ is empty and thus, count-normalised by default.

In summary, the only violation of a precondition that may lead to a grounding occurs if S contains more logvars than A. So, for a lifted calculation, it necessarily has to hold for each $S \in \mathbf{S}_{ij}^{A}$ that

$$lv(S) \subseteq lv(A) \tag{4.3}$$



(a) No groundings (b) Groundings (c) $\#_Y[Q(X,Y)]$

Figure 4.7: Conceptual examples with liftable and non-liftable message calculation

i.e., A can be eliminated before S w.r.t. logvars. In other words, LVE could sum out the separator PRVs last. Let us consider a few examples.

Example 4.4.2. Figure 4.7a shows a parcluster $\{P(X), Q(X,Y)\}$ and a separator $P(X)$. For the message, LJT eliminates $Q(X,Y)$ from the local parfactor. $Q(X,Y)$ fulfils all preconditions for lifted summing out. LJT could sum out $P(X)$ last (unnecessary for the message). Since $S = P(X)$ and $A = Q(X,Y)$, Condition (4.3) holds. The set of logvars of $P(X)$ is a subset of the set of logvars of $Q(X,Y)$. No groundings occur.

Condition (4.3) also holds for all parclusters and separators in the FO jtree of G_{ex}. For message m_{12}, separator PRV *Epid* has no logvars, making Condition (4.3) true for both PRVs $Nat(D)$ and $Man(W)$ to eliminate. The same holds for message m_{21} back with PRVs $Sick(X)$ and $Travel(X)$. For the messages between $\mathbf{C}_2$ and $\mathbf{C}_3$, Condition (4.3) also holds for $\mathbf{S}_{23} = \{Epid, Sick(X)\}$ as

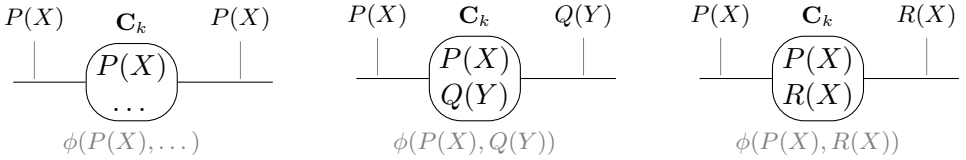
- $lv(Epid) \subseteq lv(Travel(X))$ and $lv(Sick(X)) \subseteq lv(Travel(X))$ for m_{23} and
- $lv(Epid) \subseteq lv(Treat(X,M))$ and $lv(Sick(X)) \subseteq lv(Treat(X,M))$ for m_{32}.

LVE has an operator suite designed to enable lifted summing out by applying one of the transforming operators *count-convert*, *multiply*, *split*, *expand*, or *ground*. To fix that Precondition (ii) may not hold, LVE would need to "eliminate" logvars to have A contain all logvars in g^A. Eliminating logvars is possible by counting the excess logvars in S which depends on those excess logvars to be countable in g^A. Formally, a count conversion may induce Condition (4.3) for a particular S if

$$(lv(S) \setminus lv(A)) \text{ are countable in } g^A. \tag{4.4}$$

Countable here refers to the preconditions of count conversion to apply. Counting $lv(S) \setminus lv(A)$ "eliminates" these logvars from g^A, meaning that A now contains all logvars in g^A. On the flip side, if Condition (4.4) does not hold, LJT has to ground.

Example 4.4.3. In Example 4.4.1, showcasing unnecessary groundings, Condition (4.4) already is at play. A count conversion induces Condition (4.3) for the message m'_{12}. The separator consists of $Epid(E)$ with a logvar E, which is not a subset of both $lv(Nat(D))$



(a) No groundings at $\mathbf{C}_k$ (b) No groundings at $\mathbf{C}_k$ (c) Groundings at $\mathbf{C}_k$

Figure 4.8: Conceptual examples with $\#_X[P(X)]$ in incoming message

and $lv(Man(W))$. But, E is countable in g_1 and thus, Condition (4.4) holds, which means counting E induces Condition (4.3).

Figure 4.7b shows a parcluster $\{P(X), Q(X,Y), R(Y)\}$ and separator $Q(X,Y)$ with $\mathbf{A}^E = \{P(X), R(Y)\}$ and $S = Q(X,Y)$ where LJT grounds. The set $lv(Q(X,Y))$ is not a subset of each PRV in $\mathbf{A}^E$, i.e., Condition (4.3) does not hold. $P(X)$ and $R(Y)$ are not countable in the local parfactor as no logvar appears in only one PRV, leading LJT to ground Z (or Y). Thus, Condition (4.4) does not hold. As described before, LVE would not have this problem as it had eliminated $Q(X,Y)$ before $P(X)$ and $R(Y)$, which makes X and Y countable. Figure 4.7b describes the same setting that leads to groundings for message m'_{23} in Example 4.4.1: The PRVs $Epid(E)$ and $Travel(X)$ need to be eliminated but with $S = Sick(X,E)$, Condition (4.3) does not hold. $Sick(X,E)$ has the logvars X and E, which are not a subset of $lv(Epid(E))$ and $lv(Travel(X))$. Additionally, both X and E are not countable in g'_2, preventing Condition (4.4) to hold.

In Fig. 4.7c, the separator PRV, $Q(X,Y)$, contains a logvar Y that does not appear in the PRV to eliminate ($P(X)$). Y is countable so Condition (4.4) holds, and building CRV $\#_Y[Q(X,Y)]$ is possible. Now, X is the only logvar and $P(X)$ is eliminable. The same holds if P has more logvars not appearing in $Q(X,Y)$, e.g., $P(X,Z)$.

Unfortunately, the newly count-converted PRVs may cause groundings at the receiving parcluster: Counting a logvar L may prevent groundings when calculating message $m_{i,j}$ at parcluster $\mathbf{C}_i$ but S as a CRV may cause problems at neighbour $\mathbf{C}_j$ since S appears in G_j as a PRV. If LVE does not need to sum out S, the CRV does not lead to groundings. But, if LVE has to sum out S for a message or query, LVE needs to multiply S as a PRV and S as a CRV to fulfil Precondition (i) of lifted summing out, which is only possible if the PRV becomes a CRV as well. For the conversion, L needs to be countable in G_j. Otherwise, LVE needs to ground L. Hence, count conversion only prevents a grounding if all following messages can handle the resulting CRV. Formally, for each parcluster $\mathbf{C}_j$ receiving S with counted logvar L from a parcluster $\mathbf{C}_i$, it has to hold $\forall n \in nbs(j), n \neq i$,

$$S \in \mathbf{S}_{jn} \vee L \text{ is countable in } g^S \tag{4.5}$$

where g^S is the product of all parfactors at $\mathbf{C}_j$ including messages that contain S. Let us look at some examples for clarification.

Example 4.4.4. Figure 4.8 shows example nodes as in Fig. 4.7 with another edge and $\#_X[P(X)]$ in an incoming message. In Fig. 4.8a, $\#_X[P(X)]$ does not lead to groundings as $P(X)$ is in the next separator, i.e., the first disjunct in Condition (4.5) holds. However, as $\#_X[P(X)]$ is part of the next message, we have to check if $\#_X[P(X)]$ causes groundings at the receiver. Figure 4.8b exemplifies where $\#_X[P(X)]$ is not in the next separator but X is countable in $\phi(P(X), Q(Y))$, i.e., the second disjunct in Condition (4.5) holds. In Fig. 4.8c, neither disjunct holds. $P(X)$ is not in the separator and X is not countable as X appears in $R(X)$ as well, which is part of the separator. Because X is not countable, we cannot combine $\#_X[P(X)]$ with the local $P(X)$ for summing out. Instead, we need to ground X and sum out each $P(x)$, $x \in C$, individually. In Example 4.4.1, counting E in $Epid(E)$ leads to a CRV $\#_E[Epid(E)]$ in message m'_{12}. As $Epid(E)$ is part of the next message m_{23}, i.e., the first conjunct of Condition (4.5) is true, the CRV version $\#_E[Epid(E)]$ does not lead to groundings. At $\mathbf{C}'_3$, no further message follows (first conjunct true again), meaning the counting of E does not induce groundings.

In conclusion, a message calculation does not lead to unnecessary groundings if Condition (4.3) holds, i.e., PRVS that are eliminated for a message contain at least as many logvars as separator PRVs. If Condition (4.3) does not hold for a PRV that has to be eliminated, Condition (4.3) may be induced by counting excess logvars (Condition (4.4)) but only if the resulting (P)CRVs do not cause groundings at receiving parclusters (Condition (4.5)). If count conversions do not help, either because logvars are not countable or because another parcluster cannot handle the resulting (P)CRV, preventing groundings is only possible by preventing that the message has to be calculated in the first place. And preventing a message is possible by merging the two parclusters between which the offending message flows. Fusion, which we present next, is an additional step at the end of construction to prevent unnecessary groundings based on the conditions given above and merging parclusters.

Fusion The main idea of fusion is to combine parclusters if message calculation needs groundings. Though, LJT does not continue merging parclusters during construction to keep parclusters minimal, fusion presents a compromise. Without fusion, the smaller parclusters yield more work during message passing and, subsequently, query answering through grounding which enlarges a parcluster exponentially w.r.t. domain sizes. Fusion leads to larger parclusters, which has a linear effect on the parcluster size and avoids groundings, and thus, leads to more efficient message passing and query answering. We set up a grounding test and present fusion, using the test to decide merging.

The grounding test checks each message (two checks per edge). For each message, the test checks each separator PRV against each PRV to eliminate. We use the conditions identified before. If Condition (4.3) holds, message calculation does not induce groundings. If Condition (4.3) holds for a particular separator PRV S and a PRV A to eliminate, the elimination does not lead to groundings. If Condition (4.3) does not hold, the test

checks if a count conversion helps, which involves Conditions (4.4) and (4.5), checking if a count conversion is possible at the current parcluster (Condition (4.4)) and if true, whether the resulting PCRV leads to groundings at another parcluster (Condition (4.5)). A formal definition combining Conditions (4.3) to (4.5) follows next.

Definition 4.4.1 (Fusion test)**.** Fusion checks message m_{ij}, PRV A to eliminate, and separator PRV S. The test outcomes for S given m_{ij} and A are:

Condition (4.3) holds	→ No groundings. Check next S.	(i)
Condition (4.3) does not hold	→ Check Cond. (4.4).	(ii)
Condition (4.4) holds	→ Check Cond. (4.5) for each node receiving S.	(iii)
Condition (4.4) does not hold	→ Groundings.	(iv)
Condition (4.5) holds	→ No groundings. Check next S.	(v)
Condition (4.5) does not hold	→ Groundings.	(vi)

Testing A only needs $\mathcal{A}^E$ of g^A, which is easier to build than g^A. But, we need to track the changes w.r.t. arguments in a parfactor after quasi-eliminating A when considering the next PRV A' to eliminate. Since we do not have the actual messages for G', we assume that a message covers the separator. This slight over-approximation may result in a larger $\mathcal{A}^E$ which may lead to more PRVs in $\mathbf{S}_{ij}^E$ that have to fulfil Conditions (4.3) to (4.5). Thus, our test may identify false-positives but no false-negatives.

Fusion uses the test to decide combining two parclusters, which uses the merging operation defined in Definition 4.3.2 for minimising an FO jtree. For each parcluster $\mathbf{C}_i$ in J, fusion merges $\mathbf{C}_i$ with a neighbouring parcluster $\mathbf{C}_j$ until Conditions (4.3) to (4.5) hold if applicable. To illustrate fusion including testing messages, let us look at the FO jtree of G_{ex} as well as the FO jtree that causes groundings in Fig. 4.6 from Example 4.4.1.

Example 4.4.5 (Fusion)**.** Consider the FO jtree of G_{ex}, which causes no groundings. Fusion checks each message, concluding that no groundings occur (line i) as Condition (4.3) holds for each separator PRV and PRV to eliminate.

In the FO tree of Example 4.4.1, message passing in LJT without fusion would cause grounding a logvar during message calculation. With fusion, the messages are tested first. The test goes through lines (ii), (iii), and (v) for m'_{12}, concluding that no groundings

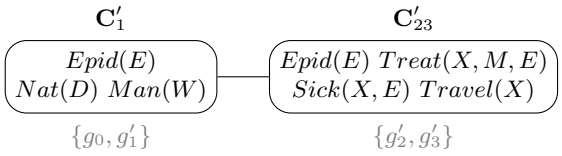


Figure 4.9: Fused FO jtree of the FO jtree in Fig. 4.6

occur. For m'_{23}, the test goes from line (ii) to line (iv) and concludes that groundings occur. Therefore, fusion merges $\mathbf{C}'_2$ and $\mathbf{C}'_3$ into one parcluster with one local model

$$\mathbf{C}'_{23} = \{Epid(E), Sick(X,E), Travel(X), Treat(X,M,E)\}, G_{23} = \{g'_2, g'_3\}.$$

Figure 4.9 shows the resulting FO jtree. The remaining message to check, from $\mathbf{C}'_{23}$ back to $\mathbf{C}'_1$, does not induce groundings, following line (i) for the PRVs $Treat(X,M,E)$ and $Sick(X,E)$ to eliminate and lines (ii), (iii), and (v) for the PRV $Travel(X)$ to eliminate.

$\mathbf{C}'_{23}$ is larger than the two former parclusters but avoids the exponential blowup during message passing. In the FO jtree in Fig. 4.9, two messages flow. For m'_{123}, E and D are counted and then, $Man(W)$ and $\#_D[Nat(D)]$ eliminated. For m'_{231}, LVE sums out $Treat(X,M,E)$ and $Sick(X,E)$ before counting E to sum out $Travel(X)$, avoiding grounding a logvar.

Given a fused FO jtree, LJT performs all calculations during an algorithm run in a lifted way. Merging parclusters based on Conditions (4.3) to (4.5) may lead to more count-conversions than LVE performs during a comparable algorithm run. If merging parclusters only based on Condition (4.3), LJT does not perform any additional count conversions compared to LVE. But, parclusters are larger as well, leading to more eliminations when answering individual queries. Keeping parclusters as small as possible is one of the driving forces of LJT, which is why we accept additional count-conversions. Next, we argue that fusion finds all avoidable groundings while possibly merging more parclusters than necessary.

Theorem 4.4.1. *Fusion prevents all unnecessary groundings (no false-negative) while possibly merging more parclusters than necessary (false-positives).*

Proof. The proof follows the same argument made at the beginning of this section: Violating Precondition (ii) leads to unnecessary groundings, i.e., PRVs that have to be eliminated contain fewer logvars than separator PRVs. Condition (4.3) identifies all messages, for which PRVs have to be eliminated with fewer logvars than separator PRVs. Conditions (4.3) and (4.4) identify all messages that count conversion salvages. I.e., after applying the test, fusion has identified all messages that may lead to groundings. Fusion then prevents that the messages that have groundings are sent by merging parclusters. Afterwards, all separator and parcluster combinations fulfil either Condition (4.3) or Conditions (4.3) and (4.4). Thus, Precondition (ii) is no longer violated.

As we over-approximate the arguments of the parfactors in a message by assuming they contain the whole separator as arguments, fusion may identify a PRV A that has to be eliminated and contains fewer PRVs than a separator PRV S, even though in an actual message, A and S do not occur in a parfactor together. Then, A is eliminable without grounding a logvar in S as S does not occur in a parfactor together with A to inhibit the elimination. Therefore, fusion may merge more parclusters than necessary, i.e., it may

produce false-positives regarding groundings. But, more importantly, fusion merges all parclusters that would send a message that produces groundings during calculation, i.e., fusion prevents false-negatives. □

After fusion, LJT continues with the second step on the fused FO jtree. After entering evidence, LJT passes messages without inducing any avoidable groundings. During query answering, LJT possibly has to work on larger local models if parclusters were fused but parfactors in messages do not contain grounded PRVs, avoiding further groundings at receiving parclusters. A comprehensive theoretical analysis of LJT follows next.

Chapter 5

Theoretical Analysis of LJT

This chapter looks at correctness and complexity aspects of LJT. Correctness consists of the aspects soundness and completeness. In the context of lifted inference algorithms, a sound lifted algorithm produces an answer to a query that is equivalent to an answer with any sound inference algorithm. A complete lifted algorithm refers to an algorithm providing an answer to a query without grounding any logvar for any possible model within a class of models. A class of models is characterised in terms of logvars, e.g., one class consists of all possible models built with two logvars (two-logvar models). First, we show that LJT is sound. Then, we present completeness and complexity results. Last, we look at how model and FO jtree characteristics influence the performance of LJT.

5.1 Soundness

The soundness proof for LJT mainly relies on the work by Shenoy and Shafer (1990). They define three axioms that justify local computations of marginals in Markov trees[1] and show that probability propagation fulfils the axioms. Markov trees are trees whose nodes contain sets of, e.g., randvars, making them basically jtrees. Markov trees also have a property that says that if a randvar appears in two distinct nodes, the randvar has to appear in every node on the path between the two nodes.

The proof for soundness of LJT has the following components. We start by showing that LJT constructs a tree that fulfils the properties an FO jtree has to fulfil. Then, we show that an FO jtree fulfils the properties of a jtree and that a jtree is a Markov tree. Next, we show that minimising and fusing an FO jtree preserves the properties of an FO jtree. After we have shown that LJT constructs a valid data structure, we move on to show that evidence handling, message passing, and query answering for single query terms is sound. We complete the proof by combining all components.

Lemma 5.1.1. *Let G be a model and T be a corresponding FO dtree, i.e., $mod(T) = G$. Then, the structure of T combined with the clusters of T form an FO jtree for G.*

Proof. An FO jtree is defined as a cycle-free graph (V, E), where the nodes in V are parclusters. An FO jtree fulfils three properties: (i) $\forall \mathbf{C}_i \in V : \mathbf{C}_i \subseteq rv(G)$. (ii) $\forall g \in G :$

[1] Shenoy and Shafer formulate the axioms for hypertrees and show that Markov trees are hypertrees.

$\exists \mathbf{C}_i \in V : rv(g) \subseteq \mathbf{C}_i$. (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).

We rely on Taghipour *et al.* (2013a) to build a valid FO dtree T of model G. As T is a tree, it is cycle-free by definition. Now, we show that each property holds reformulating a proof by Darwiche (2001), in which he shows that a dtree and its clusters form a jtree. As $mod(T) = G$, all PRVs in T are from G, i.e., $rv(T) = rv(G)$. The definitions for cutsets, contexts, and clusters use $rv(T')$ of subtrees T' of T with $rv(T') \subseteq rv(T) = rv(G)$. Thus, $cluster(i)$ of some node i contains a set of PRVs from G, fulfilling the first property. By definition of an FO dtree, its leaves contain factors and as $mod(T) = G$, each factor appears in some leaf. As leaf clusters are given by the arguments of its factors, the PRVs appear in a single cluster and thus, the second property is fulfilled. Showing that the third property holds is more involved. If a PRV appears in the cutset, context, or cluster of a node i, the PRV must appear in $rv(i)$. Suppose that i, j, and k are three nodes from T with k on the path connecting i and j and PRV A appears in $cluster(i)$ and $cluster(j)$. We have to show that A also appears in $cluster(k)$. We consider two cases.

- Assume that j is an ancestor of i. Then, k is an ancestor of i. As $A \in cluster(i)$, $A \in rv(i)$. $A \in rv(k)$ also holds since i is a descendant of k and therefore, $rv(i) \subseteq rv(k)$. $A \in cluster(j)$ means that $A \in cutset(j)$ or $A \in context(j)$. If $A \in cutset(j)$, then A is in rv for two distinct descendants of j, one of which has i in its subtree. If $A \in context(j)$, then $A \in acutset(j)$. As k is a descendant of j and $A \in cutset(j)$ or $A \in acutset(j)$, it holds that $A \in acutset(k)$. Since $A \in acutset(k)$ and $A \in rv(k)$, $A \in context(k)$ and with that, we have that $A \in cluster(k)$.

- Assume j is not an ancestor of i. Then, we have a common ancestor c. k may either be the common ancestor c or is on the path between j or i to c. It is enough to show that $A \in cluster(c)$. It follows from the first case that if $A \in cluster(c)$ and $A \in cluster(i)$, that $A \in cluster(k)$ with k on the path from i to c. The same holds if k is on the path between j and c. Since $A \in rv(i)$ and $A \in rv(j)$ and i and j are descendants of c in different subtrees, we have that $A \in cutset(c)$ or $A \in acutset(c)$. Either way, with $A \in rv(c)$, $A \in cluster(c)$.

Thus, T and its clusters fulfil all three properties. The parcluster syntax does not invalidate the properties since it does not discard or alter any information. □

Lemma 5.1.2. *Let J be an FO jtree for model G. Minimising and fusing J preserves the FO jtree properties.*

Proof. Minimising and fusing an FO jtree J alters J if two nodes merge. Therefore, we show that merging preserves the properties of an FO jtree. Darwiche (2009) states that a transformation that merges clusters $\mathbf{C}_i$ and $\mathbf{C}_j$ into a cluster $\mathbf{C}_k = \mathbf{C}_i \cup \mathbf{C}_j$ where $\mathbf{C}_k$ inherits the neighbours of $\mathbf{C}_i$ and $\mathbf{C}_j$ preserves all three properties. Since we define merging in the same way, minimising and fusing FO jtrees produces valid FO jtrees. □

Lemma 5.1.3. *Let J be an FO jtree for model G. Then, J fulfils all properties a propositional jtree fulfils and is a Markov tree.*

Proof. The properties for a propositional jtree and model F are: (i) A cluster $\mathbf{C}_i$ is a set of randvars from F. (ii) For every factor $\phi(\mathcal{R})$ in F, randvars $\mathcal{R}$ appear together in a cluster $\mathbf{C}_i$. (iii) If a randvar from F appears in clusters $\mathbf{C}_i$ and $\mathbf{C}_j$, it must appear in every cluster $\mathbf{C}_k$ on the path between nodes i and j in J. Each PRV A appearing in G represents a set of randvars $gr(A_{|C})$ and each parfactor $\phi(\mathcal{A})_{|C} \in G$ represents a set of factors f whose arguments come from the set $gr(\mathcal{A}_{|C})$. Therefore, a grounded randvar R of A is in $rv(G)$ and a grounded factor f derived from a parfactor g is in G.

Each parcluster $\mathbf{C}$ in the FO jtree J represents a set of randvars $gr(\mathbf{C}) \subseteq G$, fulfilling the first property. For every parfactor g of G, its arguments $\mathcal{A}$ appear in some $\mathbf{C}_i$, i.e., $\mathcal{A} \subseteq \mathbf{C}_i$, meaning $gr(\mathcal{A}_{|C}) \subseteq \mathbf{C}_i$. Thus, for every instance f in $gr(g)$ with arguments $\mathcal{R} \in gr(\mathcal{A}_{|C})$, $\mathcal{R} \subseteq \mathbf{C}_i$. The third property holds by the same argument: If PRV A appears in $\mathbf{C}_i$, $\mathbf{C}_j$, and in every $\mathbf{C}_k$ on the path connecting i and j, it also holds for each ground randvar $R \in gr(A_{|C_i})$ and $R \in gr(A_{|C_j})$ that R appears in every $\mathbf{C}_k$ in between.

Next, we show is that J is a Markov tree. A Markov tree as defined by Shenoy and Shafer (1990) is a hypertree, an acyclic hypergraph, which is a non-empty set of non-empty subsets from some finite set of, e.g., randvars. If there is an edge between a set of nodes, which itself are sets, then their intersection is not empty. If there are two distinct nodes with an element X, then X is in every node on the path between the two nodes. Regarding J, J is a tree, i.e., an acyclic graph. It is a hypergraph since each node is a set of PRVs from the finite set of $rv(G)$. J is a tree, thus, the intersections of neighbouring nodes are not empty. The last point about two nodes containing X follows from the running intersection property that J fulfils as an FO jtree. Therefore, J is a Markov tree. We conclude that J fulfils the jtree properties and that J is a Markov tree. □

At this point, we have shown that LJT constructs a valid FO jtree, which is also a Markov tree. Next, we show that the remaining steps of LJT are sound.

Lemma 5.1.4. *Let J be an FO jtree for model G. Then, entering evidence $\mathbf{E}$ in J is sound, i.e., equivalent to absorbing evidence with LVE.*

Proof. Evidence handling in LVE means absorbing each evidence parfactor in $\mathbf{E}$ at each affected parfactor in model G using lifted absorption. Taghipour *et al.* (2013c) show that lifted absorption and the overall handling of evidence within LVE is sound. As the parfactors in J are equal to G, we handle evidence as Taghipour *et al.* (2013c) do. Each evidence parfactor is distributed through FO jtree J and absorbed at each parfactor affected by it using lifted absorption. Distributing evidence using the running intersection property is sound as J fulfils the property. Thus, entering evidence $\mathbf{E}$ into J is sound. □

Lemma 5.1.5. *Let J be an FO jtree for model G. Then, message passing and query answering in J is sound.*

Proof. Message passing and query answering correspond to probability propagation by Shenoy and Shafer (1990). They show that probability propagation on Markov trees fulfils their axioms for local computations. As a result, one may propagate information from one node to another in a message and use the information afterwards to compute marginals within nodes. The axioms concern two main operations, combination and marginalisation, which correspond to multiplication and summing out for probability propagation, carried out on potential functions with non-negative values that are not all zero. The axioms, adapted to our setting and syntax, are:

(A1) *Commutativity and Associativity of Combination* Given three factors f_1, f_2, and f_3 respectively, then $f_1 \cdot f_2 = f_2 \cdot f_1$ and $f_1 \cdot (f_2 \cdot f_3) = (f_1 \cdot f_2) \cdot f_3$.

(A2) *Consonance of Marginalisation* Given a factor $f(\mathcal{A})$ and two sets of randvars $\mathbf{D}$ and $\mathbf{B}$ with $\mathbf{D} \subseteq \mathbf{B} \subseteq \mathbf{A}$, $\mathbf{A} = rv(\mathcal{A})$, then

$$\sum_{\mathcal{R}(\mathbf{A} \setminus \mathbf{D})} \sum_{\mathcal{R}(\mathbf{A} \setminus \mathbf{B})} f = \sum_{\mathcal{R}(\mathbf{A} \setminus \mathbf{D})} f.$$

(A3) *Distributivity of Marginalisation over Combination* Given two factors $f_1(\mathcal{A})$ and $f_2(\mathcal{B})$, $\mathbf{A} = rv(\mathcal{A})$, and $\mathbf{B} = rv(\mathcal{B})$, then

$$\sum_{\mathcal{R}(\mathbf{B} \setminus \mathbf{A})} f_1 \cdot f_2 = f_1 \cdot \sum_{\mathcal{R}(\mathbf{B} \setminus \mathbf{A})} f_2.$$

LJT takes a model that consists of parfactors, which represent factors that are potential functions with non-negative values that are not all zero as given above. With LVE as a subroutine, we follow Shenoy and Shafer (1990): Lifted multiplication fulfils (A1). Lifted summing out fulfils (A2). Distributivity of multiplication over summation leads to (A3) being fulfilled. Thus, we are justified to carry out local computations in J to pass messages and compute query answers after two passes as shown by Lauritzen and Spiegelhalter (1988). Given that LVE in the form of the suite of operators by Taghipour *et al.* (2013c) is sound, the results of local computations are sound. □

Next, we combine all previous lemmas to prove that LJT is sound.

Theorem 5.1.1. *Let G be a model, $\{Q_k\}_{k=1}^m$ a set of queries, and $\mathbf{E}$ evidence. Then, LJT is sound, i.e., computes correct answers for each $Q_k \in \{Q_k\}_{k=1}^m$ based on G and $\mathbf{E}$.*

Proof. LJT consists of the steps construction including fusion, evidence entering, message passing, and query answering. Lemmas 5.1.1 to 5.1.3 show that LJT constructs a valid minimal FO jtree and maintains a valid FO jtree after fusion. Lemma 5.1.4 shows that evidence is handled correctly. Lemma 5.1.5 shows that message passing is sound, which allows to compute a correct answer to each query. Thus, LJT is sound. □

The proof for Theorem 5.1.1 concludes the argument for LJT being sound. Next, we argue that LJT is complete for two classes of models.

5.2 Completeness

The completeness analysis is based on the notion of domain-liftability and the definition of completeness by Van den Broeck (2011) as well as the completeness results of LVE by Taghipour *et al.* (2013d). For lifted inference, domain-liftability characterises the goal of providing algorithms with a time complexity polynomial in the domain sizes of the model logvars. If an algorithm provides lifted solutions for a class of models with a given number of logvars, then the algorithm is considered complete for this class.

Definition 5.2.1 (Domain-lifted, complete)**.** For a model G, query Q, and evidence $\mathbf{E}$, a probabilistic inference algorithm is *domain-lifted* iff it runs in polynomial time in $|\mathcal{D}(X_1)|, \ldots, |\mathcal{D}(X_k)|$, $X_i \in lv(G)$. An algorithm is *complete* for a class $\mathcal{M}$ of models if it is domain-lifted for all models $G \in \mathcal{M}$, all ground queries $Q \in \mathcal{Q}$ and evidence $\mathbf{E} \in \mathcal{E}$.

Van den Broeck (2011) presents completeness results for FOKC. Taghipour *et al.* (2013d) provide results for LVE with generalised counting as well as an operator called group inversion. They refer to this LVE version as C-FOVE$^+$. Specifically, FOKC and C-FOVE$^+$ are complete w.r.t. the class of two-logvar models $\mathcal{M}^{2lv}$. Two-logvar models have only two logvars per parfactor but allow for modelling relations such as symmetry, e.g., $\phi(Friend(X,Y), Friend(Y,X))$, reflexivity, e.g., $\phi(Knows(X,X))$, and homophily, e.g., $\phi(Sick(X), ShareOffice(X,Y), Sick(Y))$. C-FOVE$^+$ is also complete w.r.t. the class of models $\mathcal{M}^{prv1}$ in which each PRV has at most one logvar. In this class, parfactors may have more than two logvars as long as PRVs only have one logvar. There exist other models with more logvars that LVE and LJT compute in a lifted way. The model from the fusion example contains three logvars in one parfactor (E, X, M) and allows for lifted computation. But, C-FOVE$^+$ is not complete for the class of three-logvar models. Three-logvar models allow for modelling transitivity, e.g., $\phi(Friends(X,Y), Friends(Y,Z), Knows(X,Z))$, which lead to groundings. More recently, Kazemi *et al.* (2017) show that using another lifting rule known as domain recursion allows for lifted runs of models representing transitivity. For LJT using C-FOVE$^+$ as a subroutine, we argue for a completeness w.r.t. the same models as C-FOVE$^+$.

Definition 5.2.2. Model class $\mathcal{M}^{2lv}$ refers to models with parfactors with at most two logvars and model class $\mathcal{M}^{prv1}$ to models with PRVs with at most one logvar.

Theorem 5.2.1. *LJT using C-FOVE$^+$ is complete for model classes $\mathcal{M}^{2lv}$ and $\mathcal{M}^{prv1}$.*

Proof. As C-FOVE$^+$ is complete for these models, the models have a lifted solution, which can be represented by an FO dtree. LJT turns the FO dtree into an FO jtree that also allows for a lifted solution after fusion. We show that message passing does not lead to new groundings. The following cases arise for two-logvar models:

1. The FO jtree has a single node. Then, LJT coincides with C-FOVE$^+$. Thus, no groundings occur while answering queries with C-FOVE$^+$.

2. The FO jtree has two or more nodes, with the separators having (a) only zero-logvar PRVs, (b) at most one-logvar PRVs, or (c) at most two-logvar PRVs. We look at the separators as they determine the messages, whose calculation has to remain lifted.
   (a) The local models of neighbouring parclusters may contain two-logvar, one-logvar, and zero-logvar PRVs to eliminate. The zero-logvar PRVs in separators do no interfere with eliminating PRVs with logvars. Thus, PRVs with logvars are eliminated with C-FOVE$^+$ without grounding.
   (b) The local models of neighbouring parclusters have two-logvar and one-logvar PRVs to eliminate. Zero-logvar PRVs do not need to be eliminated as otherwise, the nodes would have been merged. All two-logvar PRVs are eliminable with C-FOVE$^+$ as the one-logvar separators do not interfere. One-logvar PRVs either concern only a single logvar, making the PRVs eliminable using lifted summing out. If one-logvar PRVs concern logvars with different domains, count conversions enable lifted summing out, with generalised counting (Taghipour and Davis, 2012) allowing for counting logvars that appear in more than one PRV, e.g., logvars X and Y in parfactor $\phi(Q(X), R(X), S(Y), T(Y))$, resulting in $\phi(\#_X[Q(X), R(X)], S(Y), T(Y))$ after count-converting X. If one-logvar PRVs concern logvars of the same domain but with an inequality constraint, Taghipour and Davis (2012) also formalise how to merge-count a PRV and a CRV with an inequality constraint into one CRV, e.g., a CRV $\#_X[Q(X)]$ and a PRV $R(Y)$, $X \neq Y$, in a parfactor $\phi(\#_X[Q(X)], R(Y))$, leading to $\phi(\#_X[Q(X), R(X)])$. For a CRV over multiple PRVs, an operator exist to eliminate individual PRVs, meaning, it is possible to eliminate PRV $Q(X)$ or $R(X)$ from a CRV $\#_X[Q(X), R(X)]$. Thus, one-logvar PRVs are eliminable and the messages do not lead to groundings.
   (c) The local models of neighbouring parclusters have two-logvar PVRs to eliminate. PRVs with one or zero logvars occur in the separator or not at all due to fusion. Two-logvar PRVs may occur in groups, e.g., $P(X, Y)$ and $P(Y, X)$, or alone, e.g., $Q(X, Y)$. Groups appear as a whole in the separator or are to eliminate. LJT eliminates two-logvar PRVs with C-FOVE$^+$ in groups or alone, leaving the two-logvar PRVs in the separator. Thus, messages do not lead to groundings.

Overall, no groundings in any case for all two-logvar models. Query answering then works on submodels that are two-logvar models again, which have a lifted solution with C-FOVE$^+$. Therefore, LJT is complete for two-logvar models.

For models with one-logvar PRVs, cases 1, 2(a), and 2(b) apply and messages do not lead to groundings with the same arguments. Query answering then works on local models and messages of one-logvar PRVs using C-FOVE$^+$, which is complete. Hence, LJT is complete for models with one-logvar PRVs. □

Now that we have shown that LJT is sound for all inputs and complete for two classes, we look at complexity, drawing from the complexity results of LVE and JT.

5.3 Complexity

The complexity analyses of LVE and LJT mirror the complexity analyses of VE and JT, using the notion of tree width to characterise the complexity of inference. Tree width w refers to the "largest" cluster, i.e., the cluster with the most randvars, in a dtree or jtree (Darwiche, 2001). The largest cluster determines the worst case size a factor at a cluster can have, namely, if the factor has all cluster randvars as arguments. The size of such a factor is r^w, r being the largest range size in a model. Eliminating all but one randvar is bounded by $O(r^w)$ as there are at most $r^w - 1$ sum-out operations. JT has a complexity for a single query without preprocessing that is in $O(r^w)$. For VE, the complexity also depends on the overall number of eliminations for a single query.

In a worst case scenario, the lifted versions of VE and JT ground all logvars and perform inference at a propositional level for correct results. The interesting case arises when a model allows for a lifted inference solution. First, we characterise liftable models, which have a lifted inference solution and present the complexity results for LVE based on work by Taghipour *et al.* (2013a). Then, we analyse the complexity of LJT stepwise and as a whole. Last, we compare the complexity results with LVE and JT.

Liftability Taghipour *et al.* (2013a) show that the FO dtree of a model allows for a liftability test to check for a lifted inference solution.

Theorem 5.3.1 (Taghipour *et al.* 2013a)**.** *An FO dtree T has a lifted inference solution if its clusters only consist of PRVs with representative constants and PRVs with one logvar. T is called* liftable *and its one-logvar PRVs are countable.*

Counting the one-logvar PRVs leads to a *counted liftable FO dtree* of PRVs with representative constants and CRVs. In a counted liftable FO dtree, all eliminations are lifted. The FO dtree of G_{ex} in Fig. 3.4 has only clusters of PRVs with representative constants and one-logvar PRVs. That is, a lifted solution for G_{ex} is possible, which we have seen during the example calculations. For the complexity results, we concentrate on models with counted liftable FO dtrees as these models have a lifted solution.

Complexity of LVE Given the lifting setup, Taghipour *et al.* (2013a) introduce the notion of a lifted width to accommodate lifted calculations, which is defined as follows:

Definition 5.3.1 (Lifted width)**.** The *lifted width* w_T of an FO dtree T is a pair $(w_g, w_\#)$, w_g is the largest ground width and $w_\#$ the largest counting width of the clusters of T.

The largest ground width is the largest number of PRVs with representative objects in any cluster in T. The largest counting width is the largest number of CRVs in any cluster in T. Per cluster, the complexity depends on the largest possible size a factor can have, which depends on the largest range of its PRVs and CRVs as well as on the number of PRVs and CRVs there are, i.e., w_g and $w_\#$.

Theorem 5.3.2 (Taghipour 2013)**.** *In a counted liftable FO dtree T of a model G, the* node complexity *is*

$$O(\log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}), \tag{5.1}$$

with $(w_g, w_{\#})$ being the lifted width of T, n the largest domain size in $lv(G)$, r the largest range size among the PRVs in T, $n_{\#}$ the largest domain size among the counted logvars, and $r_{\#}$ the largest range size among the PRVs in the CRVs.

Let n_T be the number of nodes in T. Then, the complexity of LVE is

$$O(n_T \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{5.2}$$

In a worst case, all parfactors at a node are multiplied into one large parfactor with w_g PRVs and $w_{\#}$ CRVs appearing in it. The product $r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}$ bounds the worst case size of this large parfactor. The term r^{w_g} refers to the size of the range of the PRVs with representative objects. The term $n_{\#}^{w_{\#} \cdot r_{\#}}$ refers to the size of the range of the $w_{\#}$ CRVs. The term $n_{\#}^{r_{\#}}$ over-approximates the range size of a CRV, which is given by $\binom{n_{\#}+r_{\#}-1}{n_{\#}-1}$.

Example 5.3.1 (Worst case size)**.** Given a cluster $\{Q, R(X), \#_Y[S(Y)]$ of boolean PRVs $(r = r_{\#} = 2)$ and a parfactor $\phi(Q, R(X), \#_Y[S(Y)])$, the widths are $w_g = 2$ and $w_{\#} = 1$. Assume that the domains sizes are $n = |\mathcal{D}(X)| = 10$ and $n_{\#} = |\mathcal{D}(Y)| = 5$ Then, $r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}} = 2^2 \cdot 5^{1 \cdot 2} = 100$ bounds the size of the parfactor, which is $2^2 \cdot \binom{5+2-1}{5-1}^1 = 60$.

The parfactor size bounds the number of summations. For computing a count conversion or an exponentiation after an elimination, i.e., a number of exponentiations and multiplications for each line of a parfactor, the factor of $\log_2(n)$ appears. Given Expression (5.1), the complexity of LVE in Expression (5.2) follows.

Complexity of LJT Assume that we have a minimal FO jtree $J = (V, E)$ from a counted liftable FO dtree T for a model G. The effort for constructing J from T is in $O(n_T + n_J)$, n_T being the number of nodes in T and n_J being the number of nodes in J after minimising. Setting clusters as parclusters and minimising the result each visits all nodes in T, leading to a complexity of $O(n_T)$.

Fusion guarantees that message calculations do not lead to groundings for a model that allows for a lifted solution. LJT performs the fusion check for each edge twice. The check itself as well as merging does not depend on the factor size, only on the number of PRVs and CRVs, i.e., $w_g + w_{\#}$. There are $n_J - 1$ edges, leading to a complexity of $O(n_J \cdot (w_g + w_{\#}))$. Fusion may decrease n_J and increase w_J. As minimising J leads to a smaller number of clusters, which may further decrease with fusion, n_T is usually much larger than n_J. After fusion, J is a minimal FO jtree that does not induce groundings with $n_J = |V|$ nodes. The notion of a lifted width also applies to FO jtrees, with $w_J = (w_g, w_{\#})$ 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.

Analogously to Thm. 5.3.2, the largest possible factor in J is given by $r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}$. *Evidence entering* consists of absorbing evidence at each node if applicable.

Lemma 5.3.1. *The complexity of absorbing an evidence parfactor is*

$$O(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{5.3}$$

Absorbing an evidence parfactor at a node has a complexity $O(\log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ as in Expression (5.1) since LJT manipulates the largest possible factor and exponentiates the result if a logvar disappears. Absorbing an evidence parfactor at all nodes leads to n_j times the node complexity. For a set of evidence parfactors G_E, the complexity comes along with a factor of $|G_E|$. *Passing messages* consists of calculating messages with LVE.

Lemma 5.3.2. *The complexity of passing messages is*

$$O(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{5.4}$$

The calculation of a message at a node is in $O(\log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ as in Expression (5.1) since a message acts like a query. As LJT sends two messages per edge, of which there are $n_J - 1$, the complexity has a factor n_J. The last step is *query answering*, which consists of finding a parcluster and answering a query on an assembled submodel.

Lemma 5.3.3. *The complexity of answering a set of queries $\{Q_k\}_{k=1}^m$ is*

$$O(m \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{5.5}$$

Comparable to JT, the complexity of LJT depends only on the largest (par)cluster (and not on the number of nodes like LVE), which means a complexity of $O(\log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ as in Expression (5.1) for a single query Q_k. For the set of m queries, the complexity has a factor of m. We now combine the stepwise complexities to arrive at the complexity of LJT by adding up the complexities in Expressions (5.3) to (5.5) without construction as $O(n_T \cdot (w_g + w_{\#}))$ is much smaller than the other complexities.

Theorem 5.3.3. *The complexity of LJT is*

$$O((n_J + m) \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{5.6}$$

Comparison to LVE LVE and LJT spend the same effort on evidence. The overhead of LJT includes constructing and fusing an FO jtree with a complexity of $O(n_T \cdot (w_g + w_{\#}))$, which is negligible compared to Expression (5.6). Message passing makes up the remainder of the overhead, which has the same complexity as LVE for one query. The complexity of LVE for m queries is

$$O(n_T \cdot m \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{5.7}$$

Comparing Expressions (5.6) and (5.7) shows the difference in factor $(n_J + m)$ for LJT and factor $(n_T \cdot m)$ for LVE. Given one query, i.e., $m = 1$, LJT has the additional complexity of answering the query on a submodel. With $m > 1$, LJT has the complexity in Expression (5.1) for each query, while LVE has the factor of n_T for each query. A special case is $n_J = 1$ where both complexities coincide. With $n_J > 1$, the complexity of answering a query with LJT is lower than the complexity of LVE.

Let us illustrate the complexities by looking at G_{ex}, its FO dtree, and its FO jtree where $n_T = 6$ is the number of VE nodes, $n_J = 3$ is the number of parclusters, $n = 3$ is the largest domain size, $r = 2$ is the largest range size, $w_g = 3$ is the largest ground width, $n_{\#} = 2$ is the largest domain size of counted logvars (D), $w_{\#} = 1$ is the largest counting width ($\#_D[Nat(D)]$), and $r_{\#} = 2$ the largest range size of a PRV in a CRV ($Nat(D)$ in $\#_D[Nat(D)]$ has a boolean range). Then, Expressions (5.6) and (5.7) amount to

$$\begin{aligned} &(3{+}m) \cdot \log_2 3 \cdot 2^3 \cdot 2^{(1\cdot 2)} \quad \text{(LJT)} \\ &6{\cdot}m \ \ \cdot \log_2 3 \cdot 2^3 \cdot 2^{(1\cdot 2)} \quad \text{(LVE).} \end{aligned}$$

Without evidence, there are $m = 6$ representative queries, $Epid$, $Nat(flood)$, $Man(war)$, $Sick(eve)$, $Travel(eve)$, and $Treat(eve, injection)$. With evidence, there exist representative queries for the groups that have evidence and for the groups that do not. LJT should offset its static overhead with the second query and save even more time compared to LVE over the remaining four queries as the parclusters are relatively similar in size.

Comparison to the Junction Tree Algorithm Given the above formalisations and the description from the introduction of this complexity analysis, the complexity of VE is $O(n_{T_P} \cdot r^{w_{T_P}})$, where n_{T_P} is the number of nodes in a propositional dtree T_P for $gr(G)$ and w_{T_P} the width of T_P, i.e., the largest number of randvars in any cluster. The complexity of JT given a jtree J_P from a dtree T_P is

$$O((n_{J_P} + m) \cdot r^{w_{J_P}}), \tag{5.8}$$

where n_{J_P} is the number of nodes in J_P and m the number of queries. Comparing Expressions (5.6) and (5.8), one can observe that n_{J_P} is much larger than n_J as J leverages the potential of lifting by avoiding duplicate nodes. Additionally, using LVE and CRVs allows LJT to avoid duplicate calculations and more compact representations through CRVs as well, which means that $r^{w_{J_P}}$ is much larger than $r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}$.

5.4 Effect of Model Characteristics

The complexity results show the influence of the lifted width on the complexity and thus, the runtime. This subsection looks at effects of model characteristics beyond the worst case factor size. Before looking at the individual steps of LJT, we focus on an observation

about the lifted width. For both LVE and LJT, the lifted width is bounded from below by the parfactor with the most arguments.

Lemma 5.4.1. *The ground width w_g is bounded from below by the number of arguments in a parfactor $g_{max} = \phi(\mathcal{A}_{max})|C \in G$ s.t. $\forall g = \phi(\mathcal{A}) \in G, g \neq g_{max} : |\mathcal{A}| \leq |\mathcal{A}_{max}|$, i.e.,*

$$w_g \geq |\mathcal{A}_{max}|. \tag{5.9}$$

As a cluster at a leaf in a corresponding FO dtree is given by the arguments of the parfactor at the leaf, the largest size of a parcluster is at least the largest number of arguments. A parfactor with a high number of arguments will lead to a large parcluster that may absorb many smaller parclusters when minimising.

Construction depends on the one hand on logvars and their potential for DPGs and on the other hand on the arguments of the parfactors. After FO dtree construction, there exist as many leaf nodes as there are parfactors (N leaves). The resulting FO jtree may have between $N + 1$ and $2N - 1$ nodes if all leaves appear at the same level. Minimising the FO jtree may result into one node as a worst case, meaning LJT matches with LVE. In a best case, we have a parcluster for each parfactor, meaning small local models and thus, fast QA. Effects of fusion range from no change to a collapse into one node. Without a change, LJT checks all nodes without merging. Collapsing into one node with the input model in its local model is a worst case scenario. We add overhead for construction and fusion without a payoff since query answering compares to LVE.

Evidence has an effect on message passing and query answering since the local models change with absorption. In a worst case, entering evidence means checking Expression (4.2) for each evidence parfactor g_E at each node and each parfactor in a local model absorbing g_E. With more evidence, the size of intermediate results decreases and runtimes decrease. Passing messages, LJT calculates a message for each neighbour at a node. A high degree of neighbours may mean that computations repeat itself. A popular propositional message passing scheme is called Hugin (Jensen *et al.*, 1990), which multiplies incoming messages into a single factor at each node. Doing so, Hugin saves multiplications when calculating different messages, but requires a division of factors to avoid sending a message back to its sender and needs more memory for larger factors at nodes. In LVE, multiplication is more involved due to logvars and the number of instances they represent, which also means that specifying a division is more challenging. Additionally, one may multiply parfactors with different logvars into one parfactor, in which a count conversion may become necessary, which leads to larger intermediate results as before, all of which makes the Hugin scheme less favourable for LJT.

For query answering, the best case is an FO jtree with few PRVs per parcluster. With a clever access function, e.g., some form of index, LJT quickly identifies a parcluster for the query and sums out the few non-query PRVs. The next chapter compares the runtimes of lifted QA algorithms for prototype implementations.

Chapter 6

Empirical Evaluation of LJT

This section presents an empirical evaluation of LJT. We compare LJT to its propositional counterpart JT as well as LVE, an algorithm tailored for answering single queries. To evaluate the performance of LJT w.r.t. repeated inference, we consider FOKC. LJT, JT, and FOKC share the idea of compiling a model into a helper structure for faster answering of individual queries. FOKC compiles a model into a so-called circuit (model circuit) and computes a count in the model circuit. Given a query, FOKC compiles the model including the query into another circuit (query circuit) and computes a count in the query circuit. The query result is then given by the division of the two counts.

We have implemented a prototype of LJT and JT. Taghipour provides an implementation of LVE including the propositional version VE (available at `https://dtai.cs.kuleuven.be/software/gcfove`). Van den Broeck provides an implementation of FOKC (available at `https://dtai.cs.kuleuven.be/software/wfomc`). The implementation handles only boolean range values and does not allow for lifted handling of evidence. The programs are run on a virtual machine with 16GB working memory. We test the implementations w.r.t. the parameters that influence the complexity of LJT, namely, (i) largest domain size n, (ii) number of parclusters n_J, and (iii) lifted width $(w_g, w_\#)$ in which w_g is the ground width and $w_\#$ the counting width.

The basic input model is G_{ex} with boolean ranges, $n = 1000$, which we use for each logvar, $n_J = 3$, $w_g = 3$, and $w_\# = 1$. Evidence is empty and $Sick(x_1)$ is a representative query. When varying one parameter, the remaining parameters stay fixed. Additionally, for varying n_J, each new parcluster has a lifted width of $(3, 0)$. For varying w_g, each of the $n_J = 3$ parclusters has a ground width of w_g. For varying $w_\#$, each of the $n_J = 3$ parclusters has a counting width of $w_\#$. Doing so ensures that each message is affected and that each possible query leads to a parcluster of worst case size. The overall number of parfactors $|G|$ varies between 3 and 70 with $|gr(G)|$ ranging from 11 to 24,000,000,001. We also test the effect of evidence as well as the effect of unnecessary groundings and fusion. For evidence, we use the same basic setup as before, varying the amount of evidence entered. For fusion, we use a slight variation of G_{ex} with $n_J = 4$, which has unnecessary groundings with LJT. After fusion, $n_J = 3$ and $w_g = 5$. See Appendix B for a closer look at the inputs. Taghipour *et al.* (2013c) include tests on two real-world data sets, one modelled with four parfactors and the other with five parfactors. These data

sets, however, present rather plain overall settings as each model leads an FO jtree with two parclusters and a lifted width of $(2, 0)$. Our evaluation setup subsumes both models.

We present the empirical evaluation in five parts, one for each of the following topics (1) the effect of the given parameters on the *steps* of LJT, (2) runtimes for *query answering*, (3) the *tradeoff* between compilation runtime versus fast query answering, (4) the effect of *evidence*, and (5) the effect of unnecessary groundings and *fusion* on LJT.

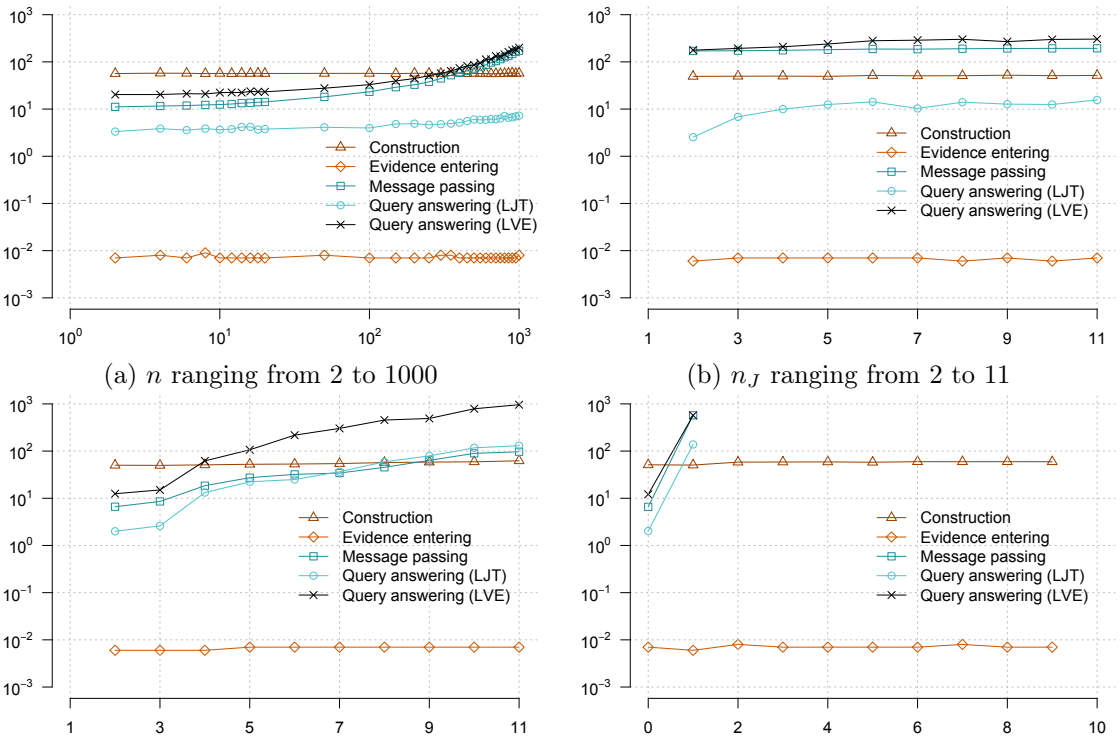
6.1 Step-wise Evaluation

This part of the evaluation looks at the different steps of LJT while varying (i) the domain size n, (ii) the number of parclusters n_J, and (iii) the lifted width $(w_g, w_\#)$. Given the complexity of LJT with one query, $n = n_\#$, and boolean ranges, i.e., $O(n_J \cdot \log_2 n \cdot 2^{w_g} \cdot n^{2w_\#})$, the following effect should occur from a theoretical view point: Varying n should have no effect on construction while the other parameters may have a slight effect as the model gets bigger. Without evidence, evidence entering should be unaffected, only showing minimal runtimes from some if-statements checking for evidence. Message passing should be affected in the order of magnitude $\log_2 n \cdot n^{2w_\#}$ when varying n. The effect of varying n_J should be linear and varying $(w_g, w_\#)$ should be exponential for each component. For query answering, the same as for message passing should hold except w.r.t. n_J, which should not have an effect on query answering.

Figure 6.1 shows runtimes for each LJT step as well as answering the given query with LVE in milliseconds [ms], averaged over five runs. Triangles mark construction runtimes, diamonds mark evidence entering runtimes, squares mark message passing runtimes, circles mark query answering runtimes with LJT, and crosses mark query answering runtimes with LVE. Evidence entering appears constant through each variation with no evidence given, with the runtime clocking in around 0.007 ms.

Figure 6.1a shows runtimes for n varying from 2 to 1000 for all logvars on log-scaled x- and y-axes. Construction appears constant w.r.t. n at around 56 ms whereas message passing and query answering with LJT show an expected increase. Message passing has a slightly steeper slope compared to query answering, with the effect of varying n applying to each message. Message passing shows a similar runtime as LVE as they have similar effort. Figure 6.1b shows runtimes on a log-scaled y-axis for n_J varying from 2 to 11. Construction has a slight linear increase (invisible in the log-scaled plot), going from 49.4 ms to 51.6 ms. The same holds for message passing though at a larger scale, going from 170.4 ms to 193.7 ms. Surprisingly, query answering appears not as constant as expected w.r.t. varying n_J, going from 2.5 ms to 15.6 ms. A reason may lie in LJT selecting a parcluster with many neighbours for query answering, leading to a large submodel with many messages, in contrast to a leaf parcluster with only one received message.

Figure 6.1c shows runtimes on a log-scaled y-axis for w_g varying from 2 to 11. Construction has a slightly stronger linear increase going from 49.8 ms to 62.4 ms. Message



(a) n ranging from 2 to 1000

(b) n_J ranging from 2 to 11

(c) w_g ranging from 2 to 11

(d) $w_\#$ ranging from 0 to 9

Figure 6.1: Step-wise runtimes [ms] for LJT and LVE (baseline); defaults: domain size $n = 1000$, number of parclusters $n_J = 3$, ground width $w_g = 3$, counting width $w_\# = 1$

passing and query answering show a steeper increase when varying w_g than when varying n_J as expected (exponential versus linear). The increase is again mirrored by LVE with a higher runtime than message passing but a similar shape of the curve. Figure 6.1d show runtimes on a log-scaled y-axis for $w_\#$ varying from 0 to 9. While construction shows a linear increase in runtime, message passing and query answering, just like LVE, show a steep increase, leading to memory errors with $w_\# \geq 2$. The reason lies in the count conversion. Though count conversions allow for lifted computations that would otherwise mean groundings, $w_\#$ means $w_\# - 1$ count conversions. Each CRV has a range of $\binom{1000+2-1}{2-1}$ which is possibly multiplied with other CRVs of the same range to eliminate a CRV. Thus, memory consumption quickly rises as intermediate parfactors get larger and operations take longer. Complexity-wise, the increase follows $n^{2w_\#}$ asymptotically, which is better than $2^{n^{2w_\#}}$ in the propositional case but may still be prohibitively large for certain domain sizes. With $n = 1000$, the effect of $n^{2w_\#}$ is rather prominent. The effect of varying w_g, though following 2^{w_g} asymptotically, is not as prominent as $2^{w_g} \ll 1000^{2w_\#}$ for the values tested. Consider Fig. 6.2 which shows the same setting but with $n = 10$.

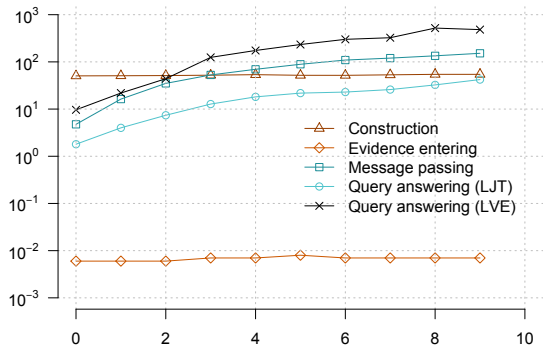


Figure 6.2: Runtimes [ms] for LJT steps; counting width $w_{\#}$ ranging from 0 to 9; domain size $n = 10$, number of parclusters $n_J = 3$, ground width $w_g = 3$

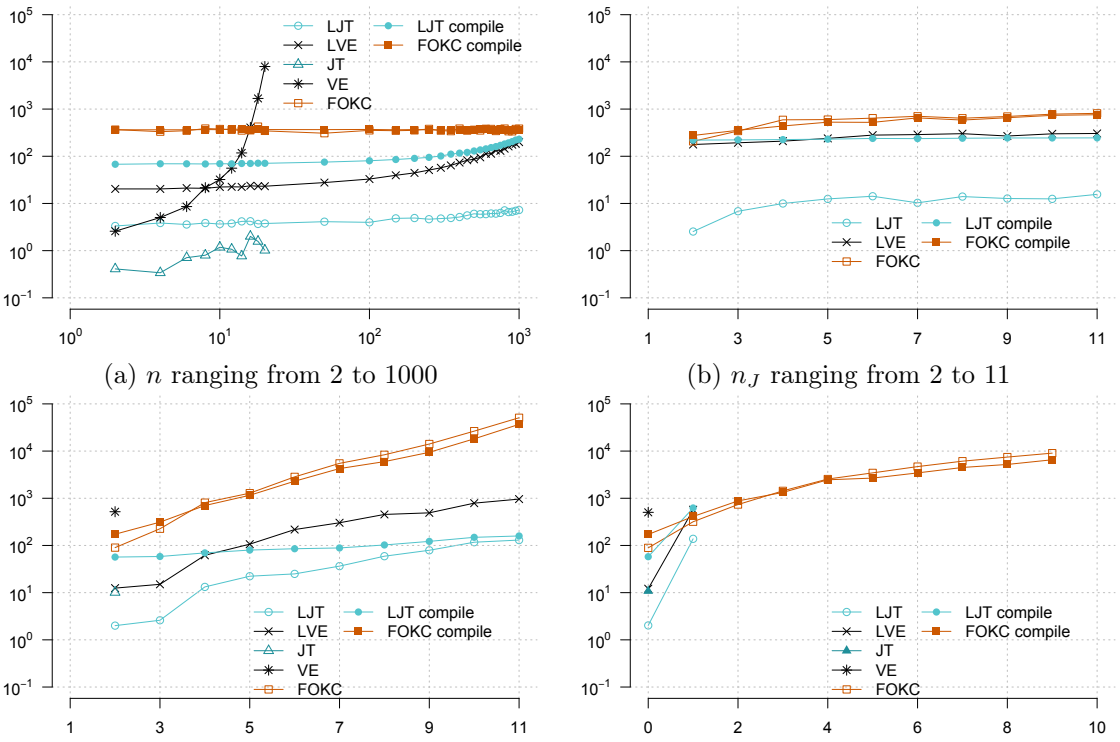
Here, LJT completes each run, exhibiting a comparatively moderate increase in runtimes for message passing and query answering with the domain size not weighing so harshly.

As the step-wise evaluation shows, an implementation allows for producing runtimes that follow the expected behaviour given the complexity analysis in Chapter 5. Construction is rather robust against parameter changes. Message passing and query answering runtimes increase with larger n, n_J, w_g, and $w_{\#}$, the last one having the greatest impact on runtime and memory consumption given our setup, while n_J has the smallest effect. Next, we evaluate LJT, JT, LVE, VE, and FOKC w.r.t. query answering.

6.2 Query Answering Evaluation

This part of the evaluation focuses on answering queries, supporting the claims that lifting allows for faster runtimes for LJT compared to JT, that compiling an FO jtree enables faster QA compared to LVE for repeated inference, and that LJT may provide faster query answering compared to FOKC, another QA algorithm geared towards repeated inference. We look at runtimes for answering a single query by again varying (i) the largest domain size n, (ii) the number of parclusters n_J, and (iii) the lifted width $(w_g, w_{\#})$. LVE and VE answer the query by eliminating all non-query terms of the original model, VE after grounding the model. LJT and JT answer the query by finding a (par)cluster and answering the query on a corresponding submodel. FOKC builds a query circuit, computes a count on this circuit, and divides the count by a precomputed count. We also consider compile time for LJT and FOKC. The compile time refers to runtime for preprocessing, which an algorithm has to trade off over multiple queries. Compiling includes construction, evidence entering, and message passing for LJT and building a model circuit and computing a count on the model circuit for FOKC.

Figure 6.3 shows runtimes for query answering in milliseconds [ms] on log-scaled y-axes, averaged over five runs. In each subfigure, hollow marks identify query answering



(a) n ranging from 2 to 1000

(b) n_J ranging from 2 to 11

(c) w_g ranging from 2 to 11

(d) $w_\#$ ranging from 0 to 9

Figure 6.3: Runtimes [ms] for query answering for FOKC, JT, LJT, LVE, and VE as well as compile time for LJT and FOKC; defaults: domain size $n = 1000$, number of parclusters $n_J = 3$, ground width $w_g = 3$, counting width $w_\# = 1$

runtimes, filled marks identify compile runtimes for LJT and FOKC. Circles mark LJT, triangles mark JT. Crosses mark LVE, stars mark VE. Squares mark FOKC.

Figure 6.3a shows runtimes for varying n from 2 to 1000 for all logvars on log-scaled x- and y-axes. VE has the steepest increase having to eliminate the grounded model almost completely. JT has the smallest runtimes for domain sizes between 2 and 20, which is due to small cluster sizes and fast elimination operations. When domain sizes get larger, VE and JT run into memory problems. LVE exhibits the same increase in runtime that we witnessed in the previous part of the evaluation as the runtimes are identical. LJT has runtimes over one order of magnitude faster for query answering starting with $n = 200$, exhibiting a similar increase with larger n. The FOKC runtimes show that FOKC is even less susceptible to varying domain sizes. Still, runtimes of LJT are over one order of magnitude smaller compared to FOKC for query answering. FOKC has a compile time similar to query answering. LJT has a compile time that is larger than LVE as it now includes construction, evidence entering, and message passing, exhibiting the same

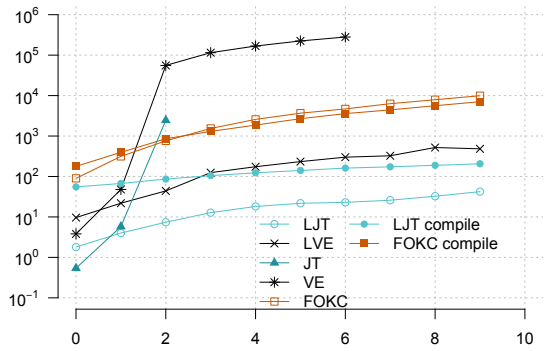


Figure 6.4: Runtimes [ms] for query answering with counting width $w_{\#}$ ranging from 0 to 9; domain size $n = 10$, number of parclusters $n_J = 3$, ground width $w_g = 3$

increase that LVE does due to message passing, the main contributor to the compile time. But, the compile time is smaller than the one for FOKC, making it faster during compilation as well as query answering. Figure 6.3b shows runtimes for n_J varying from 2 to 11. VE and JT do no appear in the figure as they do not produce a result within 5 minutes. FOKC exhibits an increase over n_J rising, going from 207 ms to 812 ms. LVE shows a slightly lesser increase (177.5 to 304.1). LJT again has runtimes that do not appear as constant as expected but are still an order of magnitude faster than FOKC and LVE. Compile times for FOKC almost coincide with runtimes for query answering, while LJT has a slightly faster compile time, which almost coincide with LVE.

Figure 6.3c shows runtimes for w_g varying from 2 to 11. The increase in runtimes is steeper than before as the effect is exponential (cf. complexity analysis in Section 5.3). VE and JT only produce a result for the first model. The fastest implementation is the one of LJT. LJT allows for query answering runtimes of close to two orders of magnitude faster than FOKC and almost one order of magnitude faster than LVE. Compile times of LJT show an increase that is much more flat than the increase FOKC compile times exhibit (56.8 to 159.0 versus 173 to 36,856). Figure 6.3d show runtimes for $w_{\#}$ varying from 0 to 9. Again, VE and JT only produce results for the first model, while LVE and LJT produce results for the first two models as discussed before. In this setting, FOKC shines as it treats scenarios in which LVE counts logvars in a less time- and memory-consuming way. Runtimes increase from 89 ms to 9046 ms over $w_{\#}$ rising, with compile times exhibiting a similar behaviour. But, FOKC at least produces a result compared to the other implementations.

Consider Fig. 6.4, which shows the same setup for $n = 10$. As count conversions are much more manageable for LVE, both LVE and LJT finish their runs and exhibit runtimes that are faster than the FOKC runtimes. LJT is again two orders of magnitude faster than FOKC for query answering. Here, VE and JT are able to produce some results, with JT running into memory problems earlier than VE.

Evaluating query answering runtimes has shown that LJT provides the fastest runtimes for individual queries for all setups except varying $w_{\#}$. For an increasing $w_{\#}$ combined with a large domain size $n = 1000$, FOKC has an advantage. With smaller n, the roles reverse again and LJT provides answers fastest for $w_{\#} \geq 1$.

6.3 Tradeoff Evaluation

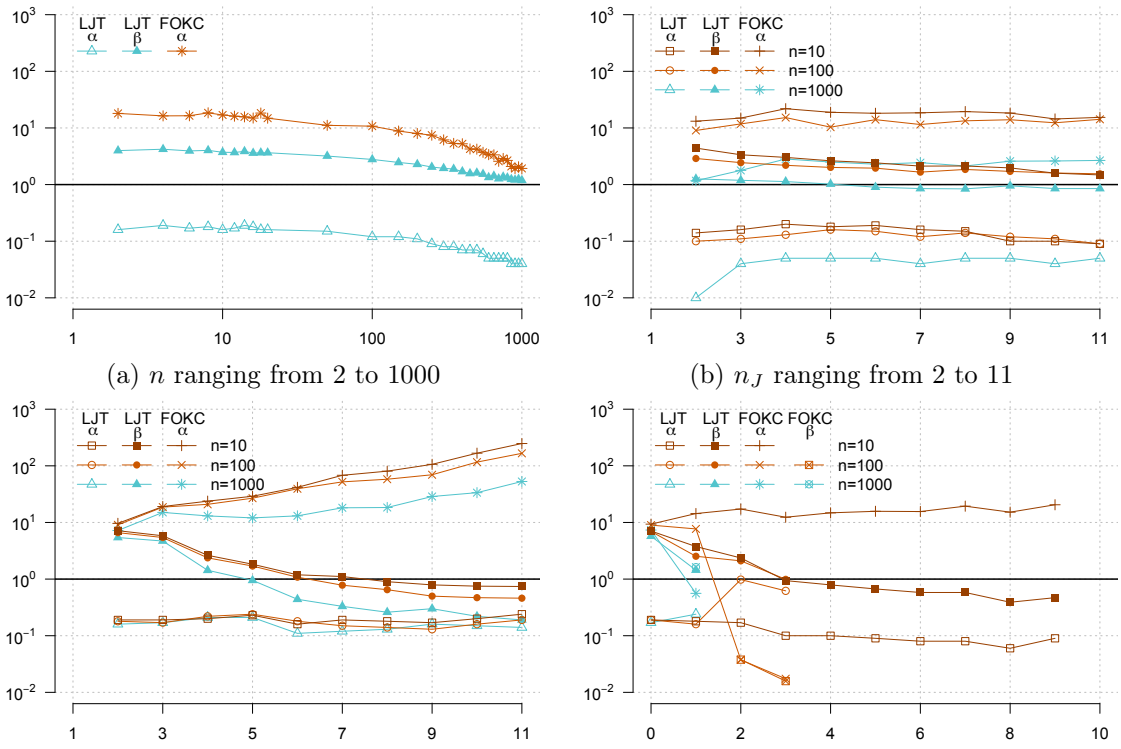
This part of the evaluation considers the tradeoff between compiling a helper structure and faster answering of individual queries. We look at the following two ratios α and β, which appear frequently when evaluating KC algorithms

$$\alpha = \frac{t_{q,cpl}}{t_{q,uncpl}} \qquad\qquad \beta = \frac{t_{c,cpl}}{t_{q,uncpl} - t_{q,cpl}}$$

where $t_{c,cpl}$ is the compile runtime of an algorithm using compilation, $t_{q,cpl}$ is the runtime for answering a single query with such an algorithm, and $t_{q,uncpl}$ is the runtime for answering a single query with an algorithm without compilation. If $\alpha < 1$, α indicates how much faster the compiled version is compared to the uncompiled one. If $\alpha < 1$, $\lceil \beta \rceil$ provides the number of queries needed to trade off compile time. $\beta < 1$ is the rare case that compilation pays off with the first query. We look at α and β for LJT and FOKC with LVE as the uncompiled QA algorithm. For LJT, $t_{c,cpl}$ includes construction, evidence entering, and message passing and $t_{q,cpl}$ finding a parcluster and answering a query on it. For FOKC, $t_{c,cpl}$ includes building a model circuit and computing a WMC on the model circuit and $t_{q,cpl}$ building a query circuit, computing a WMC on this circuit, and dividing the count by a precomputed count for FOKC. $t_{q,uncpl}$ includes eliminating all non-query terms of the original model for LVE. We again vary (i) the largest domain size n, (ii) the number of parclusters n_J, and (iii) the lifted width $(w_g, w_{\#})$.

Figure 6.5 shows α and β values of LJT and FOKC, averaged over five runs. β values are only given if the corresponding α values are below 1. The thick, black line marks 1, under which α has to lie for the compiling to pay off. Figure 6.5a shows α and β values for n varying from 2 to 1000 for all logvars on log-scaled x- and y-axes. LJT α values are marked by a hollow triangle, while LJT β values are filled triangles. Stars identify FOKC α values, no β values exist. For each n, each α is below 1 showing that LJT compilation pays off. The β values approach 1 with increasing n, going down from 3.99 to 1.2, meaning LJT needs 4 down to 2 queries to trade off its overhead.

The remaining figures have α and β values for $n = 10$ (squares/pluses), $n = 100$ (circles/crosses), and $n = 1000$ (triangles/stars) on a log-scaled y-axis. Hollow marks identify LJT α values, while filled marks identify LJT β values. The plus, cross, and star mark FOKC, which are boxed or circled if a β value exists. Figure 6.5b shows α and β values for n_J varying from 2 to 11. FOKC has α values above 1, again exhibiting smaller values with $n = 1000$ compared to $n = 10$ and $n = 100$. The α values of LJT are below 1



(a) n ranging from 2 to 1000

(b) n_J ranging from 2 to 11

(c) w_g ranging from 2 to 11

(d) $w_\#$ ranging from 0 to 9

Figure 6.5: α and β values for LJT and FOKC (β only if $\alpha > 1$); domain size n as stated; defaults: number of parclusters $n_J = 3$, ground width $w_g = 3$, counting width $w_\# = 1$

for each n, with $n = 1000$ providing the lowest α values, which mirrors the observations in Fig. 6.3a. LJT β values decrease with rising n_J, so far that for $n = 1000$, LJT is faster than LVE with the first query. Figure 6.5c shows α and β values for w_g varying from 2 to 11. FOKC only has α values, which increase noticeably with rising w_g as FOKC runtimes increase more than the LVE runtimes. LJT has α values that are similar given the different n, but with the β values decreasing. LJT reaches for all three n a point where it is faster than LVE with the first query. Figure 6.5d show α and β values for $w_\#$ varying from 0 to 9. Again, as count conversions lead to memory errors with large n, LJT only completes runs for $n = 10$ or small enough $w_\#$. For $n = 1000$, only values for the first two models exist for LVE and LJT, which also means one can only compute α and β values for FOKC for these two models, even though FOKC completed all runs. For $n = 100$, the first four models are completed, yielding β values for LJT between 1 and 10 and for FOKC well below 1 for $w_\# = 2$ and $w_\# = 3$. For $n = 10$, LVE, LJT, and FOKC complete their runs. FOKC again has α values above 1, whereas LJT has α values below 1, leading to β values below 1 with $w_\# \geq 3$.

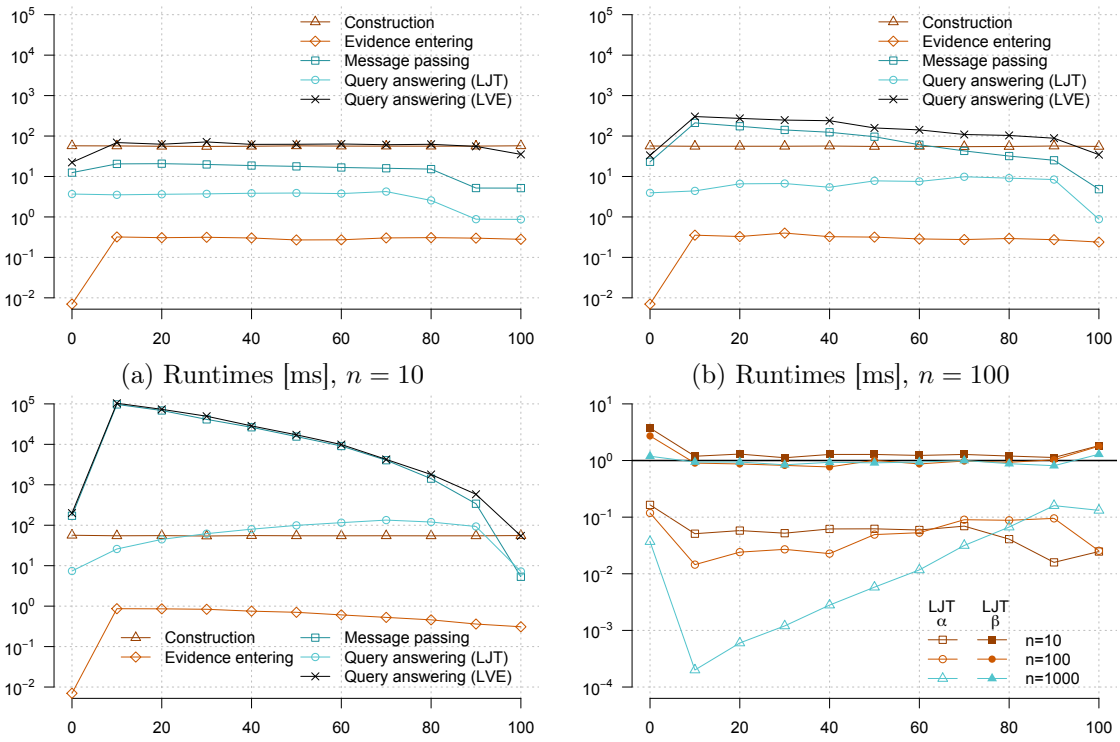
Evaluating the tradeoff between compile time and faster QA shows that FOKC does not provide a speed-up with the given setups except for a combination of large domain sizes and high counting width compared to LVE. LJT provides a speed-up of up to two orders of magnitude by building an FO jtree and preparing messages for fast QA. Especially with larger models and domain sizes, the overhead of compilation pays off for LJT with the second or even the first query. At this point, we have shown that it is possible to implement LJT to achieve the theoretically possible speed-up for QA given models with various domain sizes, numbers of parclusters, and lifted widths. But, so far we have not considered evidence and its effect on LJT, which we look at next.

6.4 Evidence Coverage

This part of the evaluation investigates how evidence affects LJT step-wise and in terms of the tradeoff between compiling and QA. The runtimes have been collected on the basic model from the beginning with boolean ranges, $n_J = 3$, $w_g = 3$, $w_{\#} = 1$, and one query. We add evidence for 1-logvar PRVs, from 0% to 100% evidence in 10% steps, excluding the query term $Sick(x_1)$. We also vary the domain sizes, setting n to 10, 100, and 1000. We do not compare against JT as the evaluation so far showed the power of lifting compared to ground computations. We do not add evidence for 2-logvar PRVs as the LVE implementation does not handle evidence for 2-logvar PRVs in a lifted way, instead grounding one of the two logvars in the evidence. We also do not consider FOKC as the implementation provided deals with each observation in a grounded fashion.

With evidence, we expect the runtimes of evidence entering, message passing, and query answering to go up as the number of parfactors basically doubles. With higher evidence coverage, we expect the runtimes of message passing and query answering to go down slightly as a higher ratio of instances is assigned a value. With 100% evidence, we expect runtimes to drop as evidence applies to whole parfactors and leads to a dimension reduction after absorption, which decreases the complexity of LJT.

Figure 6.6 shows results averaged over five runs with evidence coverage ranging from 0% to 100%. Figures 6.6a to 6.6c display runtimes on a log-scaled y-axis for each LJT step as well as answering the given query with LVE in milliseconds [ms] for $n = 10$, $n = 100$, and $n = 1000$ respectively. Triangles mark construction runtimes, diamonds mark evidence entering runtimes, squares mark message passing runtimes, circles mark query answering runtimes with LJT, and crosses mark query answering runtimes with LVE. Construction appears nearly constant through each variation, with the runtime clocking in around 56 ms. Evidence entering shows the expected jump, going from 0% to 10% evidence, for all three domain sizes. After the initial jump, runtimes decline, with $n = 1000$ showing the largest decline from 0.86 ms to 0.31 ms (64.0% down) and $n = 10$ the smallest from 0.32 ms to 0.28 ms (12.5% down). Message passing mirrors LVE again with similar computations carried out. The initial jump in runtimes also



(a) Runtimes [ms], $n = 10$

(b) Runtimes [ms], $n = 100$

(c) Runtimes [ms], $n = 1000$

(d) α and β values

Figure 6.6: Evidence coverage ranging from 0% to 100% for 1-logvar PRVs; domain size n as stated, number of parclusters $n_J = 3$, ground width $w_g = 3$, counting width $w_{\#} = 1$

appears here, followed by a decline over the course of more evidence, most prominent for $n = 1000$ again. Query answering exhibits a slow increase in runtimes until runtimes start to decline after a peak at 70% evidence. Due to the additional splits for the query term, the decline in runtime gets visible later compared to message passing.

Figure 6.6d shows α and β values of LJT with LVE as the algorithm for uncompiled QA. The thick, black line marks 1, under which α has to lie for the compiling to pay off. For each domain size and evidence set, α lies below 1. As the difference between LVE runtimes and LJT query answering runtimes is largest with 10% evidence, α is smallest at this point, with a value of 0.0002 for $n = 1000$. The β values are between 0.8 and 1.8 for all setups except for 0% evidence with $n = 10$ and $n = 100$. For these two settings, LJT needs four and three queries respectively for compiling to pay off. For all other settings, LJT needs one to two queries. Overall, we conclude that lifted evidence handling pays off. It does add runtime for all steps except construction, more so with larger domain sizes, comparing the three graphs for $n = 10$, $n = 100$, and $n = 1000$. But, again, answering queries after incorporating observed events is faster with LJT compared

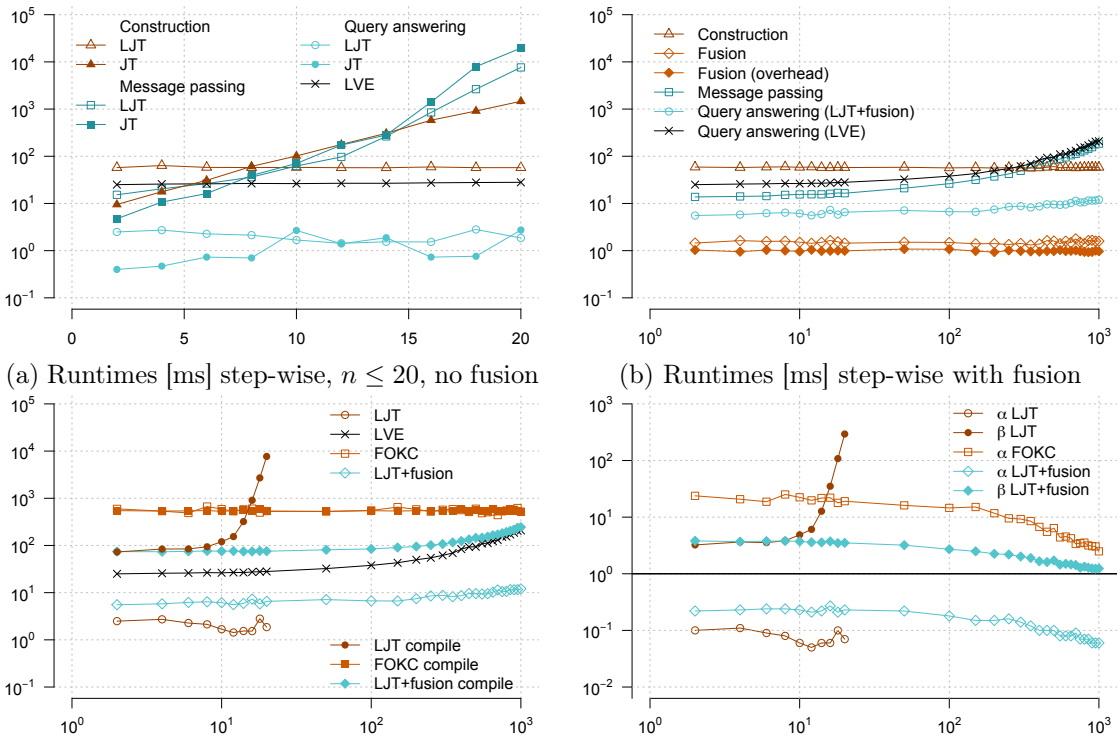
to LVE. Considering all parameters influencing LJT, we expect the results for evidence entering in the basic model to translate to other setups with varying n_J or $(w_g, w_\#)$. Runtimes for evidence entering, message passing, and query answering initially will rise but decline again with more evidence. With the same behaviour expected for LVE, the speed-up for repeated inference continues to hold for LJT. The last aspect to look at concerns unnecessary groundings and fusion.

6.5 Fusion Effect

This part of the evaluation looks at the effect of unnecessary groundings on the runtimes of LJT steps as well as fusion as a means to avoid groundings. The evaluation is based on a model that is a modification of the base model with four parcluster, i.e., $n_J = 4$, with one message leading to groundings. Thus, fusion merges two parclusters leading to an FO jtree with three parclusters and a ground width $w_g = 5$. LVE computes a solution in a purely lifted way. To illustrate the effect of the groundings, the domain size n varies from 2 to 1000. We do not consider evidence. We test LJT without fusion, LVE, FOKC, and LJT with fusion. For the smaller domain sizes ($n \leq 20$), we also test JT.

Figure 6.7 shows results averaged over five runs with the domain size ranging from 2 to 1000. Fig. 6.7a shows runtimes of the steps of LJT and JT in milliseconds [ms] on a log-scaled y-axis for domain sizes $n \leq 20$. Triangles mark construction runtimes, squares mark message passing runtimes, and circles mark query answering runtimes. Hollow marks identify LJT runtimes, while filled marks identify JT runtimes. LVE occurs as a baseline, marked by crosses. LVE exhibits a slight linear increase over the rising domain sizes, going from 25.0 ms to 28.0 ms. LJT construction is nearly constant, around 56 ms, as expected since the groundings only appear with message passing. JT construction already shows an exponential increase as the number of instances rises. Message passing shows a similar exponential increase over rising domain sizes for both LJT and JT, the unnecessary groundings being noticeable in the runtimes. Query answering itself is faster with both LVE and LJT for smaller domain sizes but with rising runtimes for message passing, both algorithms will not be able to amortise compile times. With larger domain sizes, neither the LJT nor JT implementations complete their runs. Thus, we look at step-wise runtimes of LJT with fusion, which avoids unnecessary groundings.

Fig. 6.7b shows runtimes for each LJT step including fusion as well as answering the given query with LVE in milliseconds [ms], both axes log-scaled. Triangles mark construction runtimes, diamonds mark fusion runtimes, squares mark message passing runtimes, circles mark query answering runtimes with LJT, and crosses mark query answering runtimes with LVE. The filled diamond identifies fusion runtimes on the basic model, essentially providing the overhead that fusion generates when no groundings occur. Construction again appears constant as neither domain size nor groundings affect construction. Both fusion runtimes have a minor increase, with runtimes where fusion



(a) Runtimes [ms] step-wise, $n \leq 20$, no fusion

(b) Runtimes [ms] step-wise with fusion

(c) Runtimes [ms] for query answering

(d) α and β values

Figure 6.7: Fusion with domain size n ranging from 2 to 1000; number of parclusters $n_J = 4$ (3 after fusion), ground width $w_g = 3$ (5 after fusion), counting width $w_{\#} = 1$

has an effect being slightly higher. Fusion adds runtime to the overall runtime but for the case where fusion avoids groundings, message passing is able to significantly save runtime. Message passing then shows a linear increase consistent with LVE and no longer the exponential surge without fusion. Query answering then exhibits a similar increase at a smaller scale with rising domain sizes. Overall, the added overhead without groundings is minimal, while the gain is extensive if fusion leads to avoiding groundings.

Figure 6.7c shows runtimes for query answering in milliseconds [ms] on log-scaled x- and y-axes. Hollow marks identify query answering runtimes, filled marks identify compile runtimes for LJT with and without fusion as well as FOKC. Circles mark LJT without fusion. Crosses mark LVE. Squares mark FOKC. Diamonds mark LJT with fusion. LVE runtimes are identical to Fig. 6.7b, exhibiting a linear increase over rising domain sizes. LJT with fusion runtimes are identical to query answering runtimes in Fig. 6.7b, being around one order of magnitude faster compared to LVE. FOKC again shows that it is less susceptible to varying domain sizes but still slower than LVE and LJT with fusion. LJT with fusion is close to two orders of magnitude faster despite the fusion overhead.

Compile times show a similar behaviour already encountered during the second part of this evaluation. FOKC has a compile time similar to query answering. LJT has a compile time that is larger than LVE as it now includes construction, evidence entering, and message passing, exhibiting the same increase that LVE does due to message passing, the main contributor to the compile time. But, LJT compile time is smaller than the one for FOKC, making it faster during compilation as well as query answering.

Overall, we have shown that it is possible to implement LJT such that the theoretical gain of faster QA for efficient repeated inference is realisable. Except for the combination of large domain sizes and high counting width, LJT shows a promising performance, surpassing LVE and JT as well as even FOKC, a prominent representative for repeated inference, by providing faster query answering.

Chapter 7

Part I: Interim Conclusion

This part has presented LJT to answer multiple queries efficiently in the presence of symmetries in a model. LJT builds on the propositional junction tree algorithm and LVE as a lifted QA algorithm, applying the idea of lifting to jtrees, introducing FO jtrees. LJT avoids duplicate nodes in jtrees by parameterising the cluster nodes, which allows for using LVE in its calculations, avoiding duplicate calculations for messages and queries. This part includes a comprehensive theoretical analysis of LJT with all its steps as well as an empirical evaluation. The theoretical analysis contains proofs of the soundness of the overall algorithm, its completeness w.r.t. two classes of models, and shows the complexity of LJT in terms of its lifted width. Whereas existing methods have not focussed on exact repeated inference, accumulating unnecessary repetition, or have not fully exploited relational aspects, LJT allows for exact repeated inference in relational probabilistic models for a variety of models efficiently.

There exist many possible roads for further work on optimising LJT. One interesting algorithm optimisation is parallelisation, for which message passing is predestined. Whenever Condition (1) of message passing triggers, a thread could start calculating a message to the remaining neighbour. Whenever Condition (2) triggers, a set of threads could start calculating messages to neighbours. One could also set up message passing within the MapReduce framework like Ahmadi *et al.* (2013) do for LBP. LJT adopted a message passing based on a scheme by Shafer and Shenoy (1990). In the propositional setting, Jensen *et al.* (1990) also provide a well known PP scheme, Hugin. As discussed before, the Hugin scheme by Jensen *et al.* (1990) is not as suitable for the first-order setting as the Shafer-Shenoy scheme. But, nonetheless, one could investigate whether the theoretical drawbacks of the Hugin scheme regarding divisions of parfactors and a blowup through count conversions come into play in actual FO jtrees.

Another optimisation, which is quasi orthogonal to parallelising message passing and also deals with optimising message calculation, regards caching to reuse calculations for messages. Kazemi and Poole (2016a) use a cache to optimise KC. In LJT, when calculating messages, a PRV that does not appear in any separator is eliminated first for each message. Using caching, the elimination operation does not have to be carried out multiple times. A challenge would be to set up a procedure that is fast in detecting

whether a result to an elimination already exists, e.g., through a hash function to check if a result to a computation exists and access said result.

Regarding construction, one could turn to hypergraph partitioning. A hypergraph is a graph whose edges can connect more than two nodes (hyperedges). Considering a parfactor graph, transforming factor nodes and their edges into hyperedges leads to a hypergraph. A k-way hypergraph partitioning is another method to construct a propositional jtree by partitioning the nodes of a hypergraph into k partitions of approximately equal size (Papa and Markov, 2007).

Additionally, one could look into exploiting local symmetries. Local symmetries in a factor are characterised by various input valuations mapping to the same real number. Chavira and Darwiche (2007) specify factors through algebraic decision diagrams (ADDs) instead of a list or table and thus, encode local symmetries explicitly. Then, they define the VE operations of multiplying and summing out for ADDs, which allow for fast inference given many local symmetries. An interesting avenue would be to investigate whether LVE with ADDs is feasible.

At the end of this first part, we have covered the first three contributions of this dissertation, (1) lifting of jtrees, (2) lifted QA on FO jtrees incorporating LVE, and (3) completeness and complexity results for LJT. But, so far, queries concern either a marginal (conditional) distribution with only a single randvar in the margin. A problem that the next part tackles regards repeated inference for various types of queries, ranging from a probability of a set of randvars (conjunctive query) to the most probable assignment of a subset of model randvars (MAP query).

Part II

Extending the Query Language

Chapter 8

Conjunctive Queries

The second part of this dissertation is devoted to extending the query language of LJT from allowing queries with single query terms to allowing a set of parameterised query terms in a query for retrieving probability distributions as well as MAP assignments from a model. This first chapter of Part II extends the query language to conjunctive queries.

In many scenarios, one is not only interested in one query term, but rather multiple query terms at a time, e.g., what is the probability of a person travelling *and* being sick. Such a query is a conjunctive query. Given a conjunctive query, LVE still follows its imperative of eliminating all non-query terms. However, conjunctive queries necessitate changes for LJT as query terms may no longer occur in one parcluster. The changes occur in the query answering step. It now involves finding a subtree whose clusters cover the query terms, extracting a submodel from this subtree, and answering the query on the submodel. Answering a conjunctive query in such a way follows the idea of out-of-clique inference for a variant of JT by Koller and Friedman (2009), with cliques being another name for clusters. The following paper presented LJT for conjunctive queries:

> Tanya Braun and Ralf Möller. Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm - Extended. In *Postproceedings of the 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning at the 26th International Joint Conference on Artificial Intelligence*, pages 54–72. Springer, 2018

The remainder of this chapter is structured as follows. Section 8.1 presents LJT for conjunctive queries in detail, which contains Contribution (4a) regarding lifted QA for conjunctive queries on FO jtrees. Sections 8.2 and 8.3 contribute a theoretical analysis and an empirical evaluation respectively, which partly covers Contribution (4c) about completeness and complexity results for complex queries. A conclusion wraps up this first chapter of the second part before moving on to parameterised queries.

8.1 LJT for Conjunctive Queries

Before looking into query answering, we adapt the definition of the query language to allow for representing conjunctive queries.

Definition 8.1.1 (Query)**.** A *query* $P(\mathbf{Q}|\{E_j = e_j\}_{j=1}^m)$ consists of a set of query terms $\mathbf{Q}$ and a set of events $\{E_j = e_j\}_{j=1}^m$, with $\mathbf{Q} \subseteq rv(G)$ and $E_j \in rv(G)$ being grounded PRVs or propositional randvars. If $|\mathbf{Q}| = 1$, the query is called a *singleton query*.

Example 8.1.1 (Conjunctive query)**.** A query asking for the conditional distribution of $Travel(eve)$ and $Treat(eve, injection)$ given the event $Sick(eve) = true$ is given by the expression $P(Travel(eve), Treat(eve, injection)|sick(eve))$. The query from Example 3.1.5, $P(Treat(eve, injection))$, is a singleton query without evidence.

Based on the updated query definition, LJT now takes a model G, a set of conjunctive query terms $\{\mathbf{Q}_k\}_{k=1}^m$, and evidence $\mathbf{E}$. Algorithm 4 shows LJT for answering the queries $P(\mathbf{Q}_k|\mathbf{E}), k \in \{1, \ldots, m\}$. The first three steps (construction, evidence entering, message passing) remain the same: LJT constructs an FO jtree J for G, enters evidence into J, and passes messages on J. Conjunctive queries lead to changes in the last step of LJT (query answering): For each query $\mathbf{Q}_k$, LJT finds a subtree of J that covers all query terms $\mathbf{Q}_k$. From the parclusters in the subtree, LJT extracts a submodel to answer $\mathbf{Q}_k$ with LVE. The upcoming paragraphs provide more details on each task.

Subtree Identification The goal is to find a subtree J' of the FO jtree J in which the subtree parclusters contain all query terms $\mathbf{Q}$, i.e., $\mathbf{Q} \subseteq rv(J')$. Since J' is the basis for model extraction, and the number of PRVs determines the worst-case complexity, J' should be of such form that the number of PRVs in J' is minimal. Formally, one needs to solve the following minimisation problem:

$$\arg\min_{J'} |rv(J')|, \tag{8.1}$$

$$\text{s.t. } \mathbf{Q}_k \subseteq rv(J') \tag{8.2}$$

Solving Expression (8.1) under the constraint in Expression (8.2) means trading off finding a subtree with a minimum number of PRVs and finding a subtree fast. The search space for the problem consists of all possible subtrees that fulfil Expression (8.2).

Algorithm 4 Lifted Junction Tree Algorithm with Conjunctive Queries

```
procedure LJT(Model G, Query terms {Q_k}_{k=1}^m, Evidence E)
    Construct an FO jtree J = (V, E) for G
    Enter E into J
    Pass messages on J                                  ▷ LVE as subroutine
    for each Q_k ∈ {Q_k}_{k=1}^m do
        Find a subtree J' s.t. Q_k ⊆ rv(J')
        Extract a submodel G' from J'
        LVE(G', Q_k, ∅)                                 ▷ Output or store result
```

A straightforward method to identifying a subtree J' starts with finding a parcluster $\mathbf{C}_i$ that covers at least parts of $\mathbf{Q}$ and using $\mathbf{C}_i$ as the first node in J'. Then, LJT finds further parclusters $\mathbf{C}_j$ that cover still missing query terms and adds to J' all parclusters that lie on the path from a parcluster in J' to $\mathbf{C}_j$, including $\mathbf{C}_j$. A simple heuristics is doing a breadth first search from the current J' to add parclusters that contain missing query terms. Future work includes developing a heuristics for finding a subtree with a close to optimal number of PRVs and a reasonable runtime. For singleton queries and conjunctive queries whose query terms appear in one parcluster, the subtree contains a single parcluster. Here, the minimisation problem means finding the one parcluster covering the query term(s) that has the minimum number of PRVs. From the subtree covering the query terms, LJT needs to extract a model to answer the query.

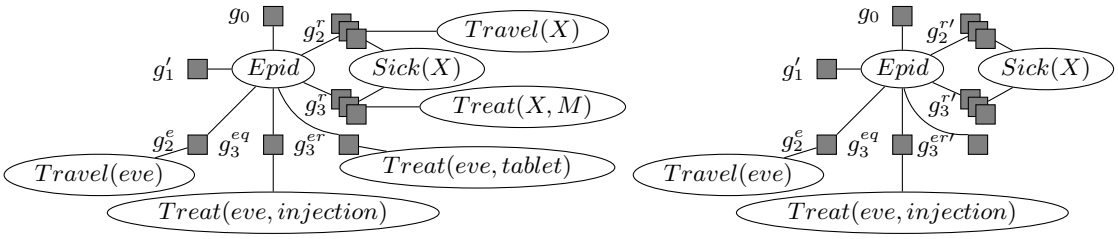
Model Extraction LJT builds a submodel G' from subtree J'. J' should be independent from the model parts in J that are outside of J'. Therefore, G' consists of the local models of the parclusters in J' and the messages that the parclusters at the borders of J' received from outside J'. The border messages combine all parfactors from outside the subtree, making the PRVs in G' independent from the remaining model PRVs.

Given J' for a singleton query or a query whose query terms appear in one parcluster, the submodel G' is the local model of the one parcluster in J' and all its received messages, which coincides exactly with the submodel definition of the original LJT.

Conjunctive Query Answering Using the model G' built during model extraction, LJT uses LVE to answer a query over the randvars $\mathbf{Q}_k$. Though LVE as described by (Taghipour *et al.*, 2013c) does not explicitly mention conjunctive queries, the formalism allows for multiple query terms. Answering a conjunctive query with LVE follows the same procedure as described in Alg. 1, now with conjunctive query terms $\mathbf{Q}$ as second input. Shattering G' on $\mathbf{Q}$ works as before, splitting parfactors based on the terms in $\mathbf{Q}$. However, multiple query terms mean a finer granularity in the model after shattering, which is an unavoidable consequence of dealing with conjunctive queries. After shattering, LVE eliminates all non-query terms and normalises the result. Given a singleton query and the respective submodel, LVE works as before, shattering the submodel on a single query term, eliminating all non-query terms, and normalising the result.

Next, we look at the query $P(Travel(eve), Treat(eve, injection) | sick(eve))$ with query terms $\mathbf{Q} = \{Travel(eve), Treat(eve, injection)\}$ as well as evidence $sick(eve)$ in G_{ex}. In Examples 4.2.1, 4.3.3 and 4.3.4, we have looked at an FO jtree J for G_{ex}, entered $sick(eve)$ as evidence into J, and passed messages on J. Thus, we can reuse J with its local models and messages for the query above as model and evidence are the same.

Example 8.1.2. A subtree of parclusters $\mathbf{C}_2$ and $\mathbf{C}_3$ covers $\mathbf{Q}$. The extracted model G' consists of the local models G_2 and G_3, shown in Figs. 4.3b and 4.3c, as well as the message from outside, m_{12}, from parcluster $\mathbf{C}_1$, shown in Fig. 4.4a. LJT uses LVE to



(a) After shattering (b) After three sum-out operations

Figure 8.1: Parfactor graph of $G' = G_2 \cup G_3 \cup m_{12}$

eliminate all non-query terms from G': LVE begins with shattering G' on $\mathbf{Q}$, the result being shown in Fig. 8.1a. Proceeding with an increasing size of intermediate results, LVE sums out $Treat(eve, tablet)$ from g_3^{er}, resulting in $g_3^{er\prime} = \phi_3'(Epid)$, $Travel(X)$ from g_2^r, resulting in $g_2^{r\prime} = \phi_2''(Epid, Sick(X))$, and $Treat(X, M)$ from g_3^r, resulting in $g_3^{r\prime} = \phi_3''(Epid, Sick(X))$. Fig. 8.1b shows the remaining model after performing the three sum-out operations. From the product of $g_2^{r\prime}$ and $g_3^{r\prime}$, LVE sums out $Sick(X)$, resulting in a parfactor $g_{23}^{r\prime}$ with argument $Epid$. To sum out $Epid$, LVE multiplies all parfactors into one. Then, LVE sums out $Epid$, resulting in a parfactor with arguments $Travel(eve)$ and $Treat(eve, injection)$, which holds the queried distribution after normalising.

Given singleton query $P(Treat(eve, injection)|sick(eve))$, query answering proceeds as shown in Example 4.3.5: Submodel G' is the union of G_3 and message m_{23} and LJT uses LVE to eliminate all terms that are not $Treat(eve, injection)$ from G'.

With the adapted query answering step, LJT is now able to answer conjunctive queries reusing the local models and messages in a given FO jtree as long as model and evidence do not change. At the same time, singleton queries are still as efficient to answer as before. Next, we look into a theoretical discussion of LJT for conjunctive queries as well as a discussion of answering conjunctive queries in cluster representations.

8.2 Theoretical Discussion

This section looks at soundness, completeness, and complexity aspects given the extended query language for LJT. First, we show that LJT for conjunctive queries is sound. Then, we present completeness and complexity results for LJT for conjunctive queries. Last, we discuss other approaches for answering conjunctive queries in cluster representations in relation to the out-of-clique based approach of LJT.

Theorem 8.2.1. *Let G be a model, $\{\mathbf{Q}_k\}_{k=1}^m$ a set of queries, and $\mathbf{E}$ evidence. Then, LJT is sound, i.e., computes a correct answer for each $\mathbf{Q}_k \in \{\mathbf{Q}_k\}_{k=1}^m$ given G and $\mathbf{E}$.*

Proof. We assume LVE in the form of the LVE operator suite (Taghipour *et al.*, 2013c) to be sound. The original LJT is sound as shown in Section 5.1. LJT for conjunctive queries is identical to the original LJT in the first three steps. Thus, LJT constructs for model G a valid FO jtree J, which fulfils the three FO jtree properties. The local models in J partition G. A valid FO jtree allows for local computations based on local models and received messages. Given that LVE is sound, evidence entering and message passing is sound. After message passing, each parcluster contains parfactors in its local models and messages that render it independent from the remaining model. The valid FO jtree allowing for local computations also allows for answering queries locally.

By way of constructing the submodel G' for the query terms $\mathbf{Q}_k$ in a query, LJT combines all necessary parfactors without duplicates. The local models do not contain duplicates as they partition G. By only considering messages from outside the subtree J', on which G' is based, still missing parfactor parts from outside J' are added to G', making the PRVs in G' independent from the outside PRVs. Ignoring the messages within J' avoids duplicating any part of any parfactor in the local models and outside messages. Thus, again given that LVE is sound, LJT computes a correct answer for a query on the extracted submodel. □

Completeness The completeness results presented in Section 5.2 hold for the class of singleton queries $\mathcal{Q}$ as well as evidence $\mathcal{E}$ that does not cause groundings. I.e., the algorithms have a complexity that is in polynomial time depending on the model only. That is the results regard model complexity. With conjunctive queries, an algorithm runtime no longer only depends on the input model only but the query terms appearing in the query, meaning that a combined complexity is relevant. For the class $\mathcal{CQ}$ of all ground conjunctive queries, the following negative result holds.

Theorem 8.2.2. *LVE and LJT are not complete for the class $\mathcal{CQ}$ of all ground conjunctive queries and all model classes that contain at least one logvar.*

Proof. Consider a conjunctive query $P(Sick(alice), Sick(eve), Sick(bob))$ for model G_{ex}. Assume LVE and LJT are complete. After passing messages in the FO jtree in Fig. 4.1b without evidence, LJT uses $\mathbf{C}_3$ and passes on to LVE a submodel G', which contains the local model G_3 and a message m_{23}. Shattering G' on the query terms $Sick(alice)$, $Sick(eve)$, and $Sick(bob)$ leads to grounding X in g_3 and m_{23}, no longer permitting a lifted solution to the query s.t. the runtime is no longer in polynomial time in $|\mathcal{D}(X)|$. □

Even with on-demand shattering, i.e., shattering a model on query terms as late as possible, the problem is only delayed. But, even though the completeness results no longer hold for all conjunctive queries, we are able to state completeness results if restricting the set of queries to ones that then allow LVE and LJT to run in polynomial time.

Lemma 8.2.1. *LVE and LJT are domain-lifted given a liftable model G, query terms $\mathbf{Q}$ containing at most one constant of each logvar in $lv(G)$, and liftable evidence $\mathbf{E}$.*

Proof. LVE and LJT are domain-lifted given a liftable model G, liftable evidence $\mathbf{E}$, and a single ground query term. What remains to be shown is that query terms $\mathbf{Q}$ with at most one constant of each logvar do not lead to a runtime depending on domain sizes.

Given $\mathbf{Q}$, LVE shatters the input model w.r.t. $\mathbf{Q}$. Given a PRV, the splitting procedure defined by Taghipour *et al.* (2013c) splits any parfactor into at most two parts. With at most one constant per logvar in $\mathbf{Q}$, a parfactor is split into at most two parfactors for each constant in a query term covered by its arguments: In a parfactor affected by some query terms $\mathbf{Q}' \subseteq \mathbf{Q}$, those query terms in $\mathbf{Q}'$ with the same logvar constant lead to one split. Query terms in $\mathbf{Q}'$ of PRVs with l different logvars lead to a split for each logvar, resulting in 2^l splits at most. With a bounded number of splits not being dependent on domain sizes, but on the number of logvars, $\mathbf{Q}$ does not lead to groundings and thus, runtimes depending on domain sizes. I.e., LVE and LJT are domain-lifted given query terms $\mathbf{Q}$, model G, and evidence $\mathbf{E}$. □

The query terms in query $P(Sick(alice), Sick(eve), Sick(bob))$ contain more than one constant for logvar X. However, queries may reference PRV instances with the same constant, e.g., $P(Sick(eve), Travel(eve))$, which uses constant *eve* for X in $Sick(X)$ and $Travel(X)$. Other liftable queries are $P(Travel(eve), Treat(eve, injection))$ and $P(Nat(flood), Man(virus))$. Limiting constants, completeness results are salvageable.

Definition 8.2.1. Query class $\mathcal{CQ}^{lift}$ refers to query terms with at most one constant of each logvar.

Theorem 8.2.3. *Completeness results of LVE and LJT hold for query class* $\mathcal{CQ}^{lift}$.

Proof. Directly from Lemma 8.2.1, it follows that the queries in $\mathcal{CQ}^{lift}$ do not lead to runtimes depending on domain sizes. Thus, the completeness results hold. □

In summary, for general conjunctive queries, positive completeness results cannot be obtained. But, by restricting conjunctive queries to $\mathcal{CQ}^{lift}$, we are able to salvage previous completeness results for LVE and LJT. Next, we look at complexity results.

Complexity For conjunctive queries, we investigate combined complexity. To acquire complexity results different from JT, we require a liftable model as well as a liftable query, as otherwise LJT may revert to propositional inference.

LJT for conjunctive queries is identical to the original LJT in the first three steps. Therefore, the complexity results up until query answering still hold, mounting up to a complexity as in Expression (5.4). The adapted query answering step consists of the tasks subtree identification, model extraction, and conjunctive query answering.

Subtree identification assembles a subtree J' with n'_J parclusters. In a worst case, a conjunctive query requires a subtree that coincides with the original FO jtree, i.e., $n'_J = n_J$. Considering the approach based on breadth-first search, the complexity depends on

the number of nodes and edges being visited. The complexity of breadth-first search is $|V| + |E|$, with V being the set of nodes and E the set of edges. As there are n_J nodes and $n_J - 1$ edges, the complexity of subtree identification can be summarised by $O(n_J)$. The result of subtree identification is a subtree J' with n'_J parclusters, which is equal to n_J in a worst case based on the query.

Model extraction walks over the subtree with n'_J parclusters to collect the local models and messages without manipulating any factors. Thus, the complexity of model extraction is $O(n'_J)$. Of course, as $n'_J = n_J$ in the worst case, $O(n_J)$ would also be correct but to distinguish between cases where n_J holds irregardless of a query, e.g., in message passing, and cases where n_J depends on the query, we use n'_J. The result of model extraction is a submodel G' that covers the query terms of a liftable conjunctive query.

Conjunctive query answering follows model extraction and manipulates factors in G'. Of course, given a conjunctive query $\mathbf{Q}$ with $q = |\mathbf{Q}|$ query terms, the result has a size exponential in q. But as q is a fixed, small number, which is outweighed by the complexity of the lifted computations in the remaining model after shattering the submodel on the query terms, we can neglect q (fixed parameter tractability in the liftable case).

The question is how large a factor gets in a worst case when computing $\mathbf{Q}$ on G'. G' implicitly contains the n'_J parclusters with factors that have a worst case size of $r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}$. Parclusters encode an elimination order, in which LVE can eliminate PRVs. From the periphery of the underlying J' inward, LVE eliminates non-separator PRVs, resulting in factors that are smaller than before. LVE multiplies the resulting factors into the factor at the inward neighbours, which keeps the factor at the neighbours at the worst case size. At the inward neighbours, LVE eliminates non-separator PRVs again and multiplies the result into the new inward neighbour, keeping the factor at the worst case size. At some central node, LVE eliminates the remaining PRVs from a factor of worst case size. Thus, the worst case size of a factor remains $r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}$, which bounds the number of summations. As there are n'_J parclusters in the underlying subtree where LVE eliminates PRVs, the overall complexity is then bound by $O(n'_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$, which outweighs the complexities $O(n_J)$ and $O(n'_J)$ of the two previous tasks. Thus, the following overall complexity arises for answering a liftable conjunctive query.

Lemma 8.2.2. *The complexity of answering a liftable conjunctive query is*

$$O(n'_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{8.3}$$

If $n'_J = n_J$, Expression (8.3) becomes $O(n_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ which coincides with the complexity of LVE for a single query. The result is reasonable as in the worst case, the extracted submodel coincides with the original input model G, on which LJT performs LVE. If $n'_J = 1$, i.e., a given query can be answered on a single parcluster, Expression (8.3) becomes $O(\log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ which coincides with the complexity of query answering in the original LJT. Combining the complexity results of the steps of LJT in Expressions (5.4) and (8.3) leads to the following asymptotic complexity.

Theorem 8.2.4. *The complexity of LJT given a set of liftable queries* $\{\mathbf{Q}_k\}_{k=1}^m$ *is*

$$O((n_J + m \cdot n'_J) \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{8.4}$$

Comparing the complexity of the original LJT in Expression (5.6) and the complexity of LJT for conjunctive queries in Expression (8.4) shows that a factor of n'_J occurs in the query answering portion. This factor varies from query to query, which the empirical evaluation in Section 8.3 reflects. Given a set of liftable conjunctive queries, LJT is able to trade off runtimes for queries with larger subtrees with runtimes for queries with smaller subtrees, preserving its runtime advantage over LVE.

Approaches towards Conjunctive Queries LJT for conjunctive queries is based on the out-of-clique idea by Koller and Friedman (2009), which uses local models and messages present after preprocessing. LJT extracts a submodel from a subtree to perform LVE on.

To bypass the problem of query terms over multiple parclusters, one could set up an FO jtree that contains a parcluster with all query terms of a given query, an approach discussed by Koller and Friedman (2009) as well. For LJT to build such an FO jtree, it would require adding a parfactor with the query terms as arguments to the input model (with potentials of 1). Consequently, LJT may need to rebuild an FO jtree for a new query, making this approach inefficient for repeated inference.

Another approach to answering conjunctive queries solves the problem by performing a message pass adapted to a query. When calculating a message, LJT normally eliminates all PRVs that do not appear in the separator. For query-induced messages, LJT eliminates all PRVs that do not appear in the separator *and* the query, basically adding the query terms to each separator. Once the query-induced message pass is finished, any parcluster may output an answer by eliminating the non-query terms from the corresponding submodel. While the approach is correct, two main disadvantages are that (i) messages become larger, with the query terms leading to a shattering for message calculation, and (ii) messages require recalculation with a subsequent query. The first disadvantage might be slightly mitigated by adding the PRVs behind the query terms to a message. But given the preconditions for lifted summing out, the new PRVs may cause groundings and thus require additional PRVs in the message to avoid grounding. This kind of mitigation easily leads to a model for query answering that is identical to the submodel LJT extracts. The second disadvantage might be mitigated by storing original messages to reuse for other queries, which requires additional memory, however.

In summary, LJT for conjunctive queries, based on out-of-clique inference by Koller and Friedman (2009), requires the least amount of changes to the overall procedure of LJT. Construction, evidence entering, and message passing remain the same. Query answering still assembles a submodel and answers the query on the submodel, with the assembly of the submodel taking slightly more effort. The result of the first three steps is reused as much as possible during the adapted query answering step, allowing LJT to answer singleton queries as efficiently as before, without requiring additional memory.

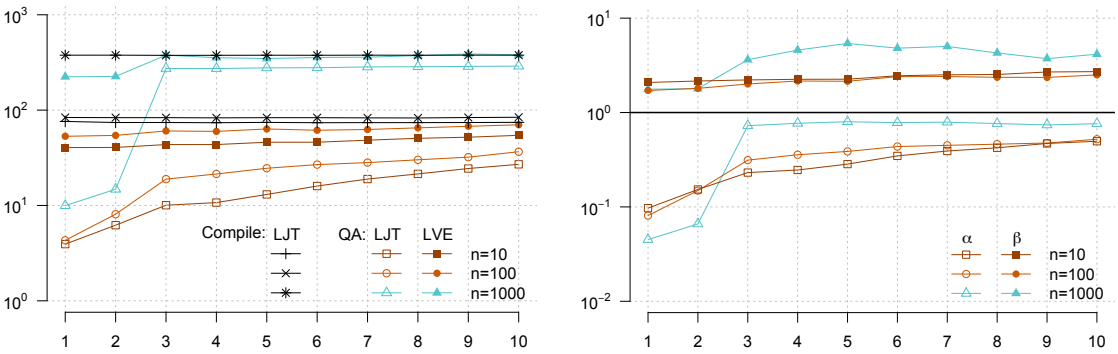
8.3 Empirical Evaluation

This section presents an empirical evaluation regarding *conjunctive queries*. As LJT for conjunctive queries is identical to the original LJT w.r.t. construction, evidence entering, and message passing, we focus on how conjunctive queries influence LJT. Based on the results of our theoretical discussion, we look at liftable and grounding queries, specifically, (i) the influence of the subtree size on query answering given a liftable query and (ii) the effect of a grounding query, including preemptive as well as on-demand shattering. As a baseline, we use LVE for conjunctive queries. We do not include the propositional versions of LJT and LVE as the lifted versions easily outperform the propositional ones as shown in Chapter 6. We have implemented LJT for conjunctive queries and extended the LVE implementation by Taghipour (available at `https://dtai.cs.kuleuven.be/software/gcfove`). The programs are run on a virtual machine with 16GB working memory. We do not compare against FOKC as the implementation by Van den Broeck (available at `https://dtai.cs.kuleuven.be/software/wfomc`) does not allow for conjunctive queries. The input model has an FO jtree with $n_J = 10$ parclusters and a lifted width of $(3, 1)$. We do not consider evidence as evidence does not have a specific effect on conjunctive queries.

Subtree Size Based on the complexity analysis, the subtree size has a linear influence on the overall runtime for answering a liftable conjunctive query. Compilation, i.e., steps one to three of LJT, does not depend on specific queries. We use ten liftable queries with a subtree size of n'_J ranging from 1 to $10 = n_J$. Each individual query consists of n'_J query terms. Each query term falls into one parcluster.

We look at the following aspects, (i) runtimes for query answering without compilation and (ii) tradeoff between FO jtree compilation runtime and query answering runtime (α, β). Aspect (i) highlights how subtree sizes have an influence on the LJT steps. Aspect (ii) showcases how LJT still trades off its static overhead.

Figure 8.2 shows the results for LJT and LVE answering the ten queries of various size. The x-axes show the increasing subtree size. Figure 8.2a depicts runtimes in milliseconds [ms] on a log scale for the last step of LJT, query answering, abbreviated QA, and query answering with LVE for three different domain sizes $n \in \{10, 100, 1000\}$ of all logvars in the input model. The figure also shows compile times for LJT (black crosses/stars), which do not depend on the subtree size. Even though compile times are plotted over all queries, LJT could have constructed an FO jtree once and then had been able to reuse its messages for each query. With a larger domain size, QA runtimes are higher as expected from the complexity analysis. But in all three cases, behaviours w.r.t. subtree size mirror each other, with LVE expectedly taking longer than LJT. Runtimes show an increase with rising subtree sizes. For LVE (filled symbols), the increase is steady over all subtree sizes, except for crossing from $n'_J = 2$ to $n'_J = 3$. The reason for the larger jump in runtimes at this point lies in one query term causing a split in a



(a) Runtimes [ms] for query answering for LJT and LVE as well as compile time for LJT

(b) α and β values for LJT runtimes compared to LVE runtimes

Figure 8.2: Subtree size n'_J ranging from 1 to 10; domain size n as stated

parfactor that requires a count conversion, which means that from this point on, query answering requires two count conversions. Runtimes of LJT (hollow symbols) show a similar behaviour. There is a larger increase between $n'_J = 2$ and $n'_J = 3$ as from n'_J onwards, the local models contain one parfactor that requires a count conversion, which was not necessary before as the count conversion and subsequent elimination had been done during message passing. Over all ten settings, LJT runtimes are smaller than LVE runtimes, although the LJT runtimes approach LVE runtimes with larger subtree sizes as expected. Thus, LJT leverages most of its potential with small queries.

We consider α and β to see how subtree sizes influence the capacity of LJT to tradeoff its static overhead. Figure 8.2b shows α (hollow symbols) and β (filled symbols) values on a log scale for LJT compared to LVE with increasing subtree size and the three domain sizes $n \in \{10, 100, 1000\}$. For all three domain sizes, α is below 1, meaning that compilation pays off. The plot also shows that with increasing subtree size, α gets closer to 1 as LJT saves less runtime w.r.t. LVE than with smaller queries. Overall, the β values lie between 1.7 and 5.4, meaning two to six queries are necessary to trade off the compilation overhead given queries of size n'_J. Though it takes more queries, LJT is able to trade off its overhead even with large queries. Two small queries are enough to trade off its overhead. With each additional query, LJT increasingly outperforms LVE.

The queries considered so far are liftable conjunctive queries over different subtrees. Next, we consider a grounding query within one parcluster.

Groundings In this part of the evaluation, we look at the effect of a grounding query on the runtime of LJT. We increase domain sizes and with it the number of grounding query terms, i.e., $\mathbf{Q} = \bigcup_{x \in gr(X)} \{R(x)\}$ for some PRV $R(X)$ in the input model. A grounding query affects LVE and LJT alike, although the consequences for LVE may be more serious as more PRVs may get grounded in the input model compared to the smaller submodel

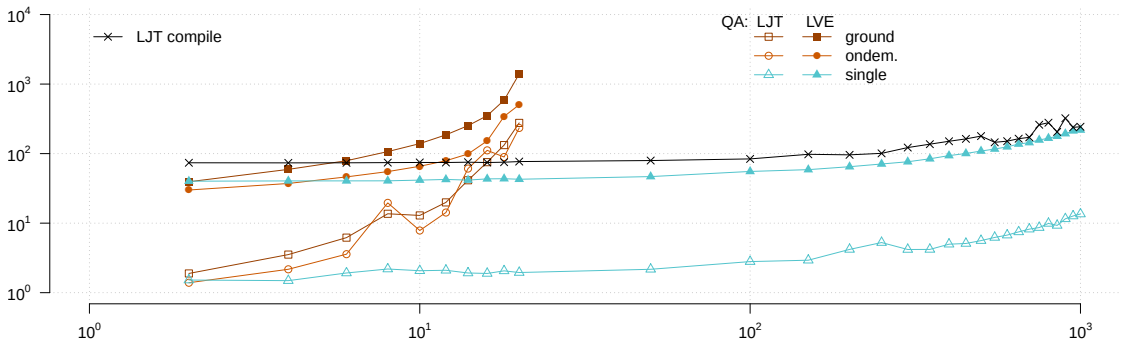


Figure 8.3: Runtimes [ms] for query answering for LJT and LVE as well as compile time for LJT w.r.t. domain size n ranging from 2 to 1000; subtree size $n'_J = 1$

that LJT uses for query answering. We compare the runtime behaviour of LJT and LVE during query answering with runtime behaviour for a singleton query, i.e., $R(x)$, as well as runtime behaviour using on-demand shattering instead of preemptive shattering.

Figure 8.3 shows runtimes in milliseconds [ms] on a log scale for query answering with LJT (hollow symbols) and LVE (filled symbols) for the grounding query with preemptive shattering (squares, "ground"), for the grounding query with on-demand shattering (circles, "ondem."), and the singleton query (triangles, "single"). The x-axis plots the increasing domain size for model logvars, ranging from 2 to 1000, on a log scale. The singleton query runtimes show the polynomial increase with rising domain sizes that we expect from the runtime complexity. This query is the only query that allows both LVE and LJT to complete its computations for all domain sizes. In both setups with the grounding query, LVE and LJT compute an answer to the query up to a domain size of 20, exhibiting a sharp increase with rising domain sizes. With a domain size of 50, both programs do not generate an answer within twenty minutes. Even though runtimes with on-demand shattering are usually lower than runtimes with preemptive shattering, the gain is not significant. The corresponding compile runtime of LJT (black crosses) displays an expected polynomial increase given larger domain sizes as message passing depends on domain sizes polynomially.

The empirical evaluation shows that LJT is able to trade off its overhead even with larger queries in an implementation given a set of queries. At the same time, implementations get into difficulties if queries force a grounding. Overall, LJT for conjunctive queries is able to leverage its cluster representation, saving runtime during query answering.

8.4 Interim Conclusion: A Set of Conjunctive Queries

Conjunctive queries allow for querying for more complex events and distributions. Handling them efficiently is of great importance. LJT for conjunctive queries permits posing

conjunctive queries without compromising its performance for singleton queries. The first three steps of LJT remain the same, only query answering sees changes in assembling a submodel for a given query based on a subtree that covers the query terms. Then, LJT uses LVE for answering the query. The whole procedure is inspired by out-of-clique inference by Koller and Friedman (2009). In a worst case, LJT and LVE have the same runtime for answering a query, with the submodel identical to the input model. But, in all other cases, LJT maintains its advantage compared to LVE, working on smaller submodels, thus, providing answers faster. While arbitrary conjunctive queries can lead a lifted algorithm to ground, conjunctive queries that use only a restricted set of logvar constants still allow for algorithm runs with lifted calculations only.

This chapter has covered Contribution (4a), lifted QA for conjunctive queries on FO jtrees, and partially covered Contribution (4c), combined completeness and complexity results for complex queries. As mentioned before, developing heuristics for finding a minimal submodel given a conjunctive query is one possible road for research in the future. One avenue of work that came out of looking at conjunctive queries concerns a specific subset of those conjunctive queries leading to groundings, namely queries with interchangeable query terms like $P(Sick(alice), Sick(eve), Sick(bob))$. One is able to establish lifted algorithm runs by applying lifting to a query as well, which the next chapter delves into, extending our query language even further.

Chapter 9

Lifting for Queries

Though LVE and LJT realise lifted inference using logvars as parameters to represent sets of interchangeable objects, queries concern query terms that are instances. As we have seen in the previous section, a set of query terms can lead to groundings in a model if, e.g., query terms reference a majority of some logvar domain. Consider the epidemic example. One might not only be interested in the probability of one individual but a specific fraction of people being sick or even in a distribution over all possible fractions from none to all. Posing a query within the query definition so far would require explicitly listing instances of $Sick(X)$ to form a conjunctive query, those instances being interchangeable given a model. The following shattering of a model on a query leads to a grounding of the affected logvars. The result also shows potential for more compact encoding: Given interchangeable query terms, it does not matter in a result in which half the query terms are assigned the value true and half of them are assigned the value false, which explicit query instances are true and which are false.

Therefore, to avoid groundings that a query induces, we have presented the notion of *parameterised queries*, introducing logvars in queries. With parameterised queries, we compute answers more efficiently and provide compact representations for queries and answers. We first presented parameterised queries in the following paper:

> 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. IJCAI Organization, 2018

The remainder of this chapter starts with a closer look at conjunctive queries of interchangeable query terms. Then, we present parameterised queries and how LVE handles such queries, which covers Contribution (4b). We establish the adapted LVE version in LJT to continue leveraging the advantages of LJT for repeated inference. In the theoretical discussion, we analyse soundness and correctness of lifted inference with parameterised queries, characterising which queries allow for lifted runs and which queries lead to unavoidable groundings, including a comparative discussion of queries with free variables in probabilistic databases (PDBs). The theoretical discussion makes up the remaining part of Contribution (4c). We close this chapter with an empirical evaluation showing the benefit of parameterised queries in terms of runtime and memory.

9.1 Parameterised Queries

As discussed in the proof of Thm. 8.2.2, a query like $P(Sick(alice), Sick(eve), Sick(bob))$ leads to grounding a logvar, namely X, which elicits inefficiencies (i) in a query, (ii) during QA, and (iii) in a result. Parameterised queries compactly represent queries, which enables efficient QA with LVE and facilitates a compact representation of the result. We first look at the inefficiencies and then define parameterised queries.

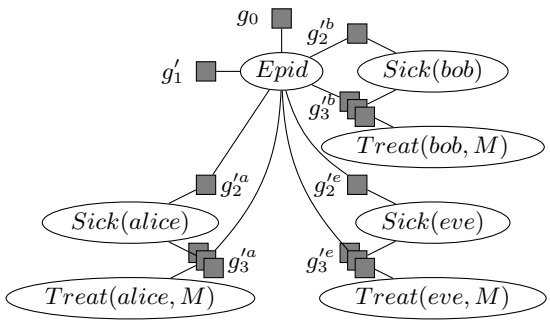
Consider $Sick(X)$ grounded as interchangeable query terms for G_{ex}. We do not include evidence at this point for ease of exposition.

Example 9.1.1 (Grounding query)**.** Given a query $P(Sick(alice), Sick(eve), Sick(bob))$ and the local models and messages from the ongoing example, LJT performs a message pass because evidence has changed, sending four messages:

- $m_{12} = \{g_0, g_1'\}$ as before since the previous evidence had no effect on $\mathbf{C}_1$,
- $m_{32} = \{g_3'\}$; g_3' is the result of eliminating $Treat(X, M)$ from g_3,
- $m_{23} = \{g_2', g_0, g_1\}$; g_2' is the result of eliminating $Travel(X)$ from g_2, and
- $m_{21} = \{g_{23}'\}$; g_{23}' is the result of eliminating $Sick(X)$ and $Travel(X)$ from g_2, g_3'.

For the query terms $\mathbf{Q} = \{Sick(alice), Sick(eve), Sick(bob)\}$, LJT can choose parclusters $\mathbf{C}_2$ or $\mathbf{C}_3$. Choosing $\mathbf{C}_3$, LJT lets LVE shatter $G' = G_3 \cup m_{23}$ on $\mathbf{Q}$, which leads to effectively grounding X, shown in Fig. 9.1a. Next, LVE eliminates the $Treat$ PRVs, each elimination a copy of the other. To eliminate *Epid*, LVE multiplies the remaining parfactors into one parfactor with arguments *Epid*, $Sick(alice)$, $Sick(eve)$, and $Sick(bob)$, which means a size of $2^4 = 16$. LVE eliminates *Epid* from this product and normalises the result. The resulting parfactor $g = \phi(Sick(alice), Sick(eve), Sick(bob))$ has $2^3 = 8$ mappings from input range values to output real numbers, which is exponential in the number of query terms. Figure 9.1b shows the mappings in a table, without explicit potentials as they depend on the concrete potentials in the parfactors in G_{ex}. The potentials exhibit a symmetry that we have already observed when introducing CRVs: Two times a $false$ value and one time a $true$ value map to v_1. It is irrelevant whether *alice*, *eve*, or *bob* is the one being sick, as long as one has the value $true$ assigned. The same holds for $false$ being assigned once and $true$ being assigned twice. A CRV $\#_X[Sick(X)]$ compactly encodes the same information, with histograms $[0, 3], [1, 2], [2, 1], [3, 0]$ as range values that map to v_0, v_1, v_2, v_3, respectively.

The example exemplifies three issues, (i) a large set of interchangeable query terms, (ii) inefficiencies during LVE, leading to (partial) groundings w.r.t. the referenced constants, to identical eliminations, and to large intermediate results, and (iii) a large result representation with symmetries. Parameterising a query allows for using existing lifting techniques to enable a lifted calculation of queries over interchangeable query terms.



(a) $G' = G_3 \cup m_{23}$ after shattering

$Sick(alice)$	$Sick(eve)$	$Sick(bob)$	ϕ
$false$	$false$	$false$	v_0
$false$	$false$	$true$	v_1
$false$	$true$	$false$	v_1
$false$	$true$	$true$	v_2
$true$	$false$	$false$	v_1
$true$	$false$	$true$	v_2
$true$	$true$	$false$	v_2
$true$	$true$	$true$	v_3

(b) Result

Figure 9.1: Answering query $P(Sick(alice), Sick(eve), Sick(bob))$ on G_{ex}

A parameterised query contains logvars in query terms to combine interchangeable query terms, achieving a compact query representation. To still allow for conjunctive queries, a parameterised query concerns a set of query terms $\mathbf{Q}$, which may contain PRVs with logvars, making $\mathbf{Q}$ a parameterised set. The formal definition follows:

Definition 9.1.1 (Query)**.** A *query* $P(\mathbf{Q}_{|C}|\{E_j = e_j\}_{j=1}^m)$ consists of a set of query terms $\mathbf{Q} \subseteq rv(G)$, a constraint C restricting the logvars in $\mathbf{Q}$, thereby, specifying the instances of the query terms, and a set of events $\{E_j = e_j\}_{j=1}^m$, where $E_j \in rv(G)$ are grounded PRVs or propositional randvars. We omit $|C$ if $C = \top$. $P(\mathbf{Q}_{|C}|\{E_i = e_i\})$ compactly represents a conjunctive query $P(gr(\mathbf{Q}_{|C})|\{E_i = e_i\})$ with query terms $gr(\mathbf{Q}_{|C})$.

Example 9.1.2 (Parameterised queries)**.** The expression $P(Sick(X)_{|\top})$ is a parameterised query equivalent to the conjunctive query $P(Sick(alice), Sick(eve), Sick(bob))$, with a query term $Sick(X)$ and a $\top$ constraint. To query for $P(Sick(alice), Sick(bob))$ with *eve* excluded, $\{Sick(X)\}$ requires a constraint $C = (X, \{(alice), (bob)\})$.

The definition allows for conjunctive queries as before by explicitly naming instances of a PRV in the query terms. Next, we present LVE for parameterised queries, which avoids inefficiencies elicited from shattering a model on interchangeable query terms.

9.2 Lifted Inference Algorithms for Parameterised Queries

LVE for parameterised queries still follows the idea of first eliminating all non-query terms and second, normalising the result to get a joint distribution over query terms. The second part is more involved as the elimination result may not be a joint distribution but rather a set of local distributions that LVE has to transform into a joint distribution. LJT for parameterised queries including conjunctive queries proceeds as described in

Algorithm 5 LVE for Parameterised Queries

```
1:  procedure LVE(Model G, Query terms Q, Evidence E)
2:      G ← Shatter G on Q
3:      G ← Absorb E in G                          ▷ Shatters G on E if necessary
4:      while G contains non-query PRV do
5:          if there exists a PRV A eliminable then
6:              G ← Sum-out A in G
7:          else
8:              G ← Apply transforming operator applicable in G
9:      while lv(G) ≠ ∅ do
10:         if ∃X ⊆ lv(G) s.t. X is countable then
11:             G ← Count X in G
12:         else
13:             G ← Apply other transforming operator applicable in G
14:     G ← Multiply remaining parfactors in G and normalise the result
15:     return G                                   ▷ Contains one parfactor
```

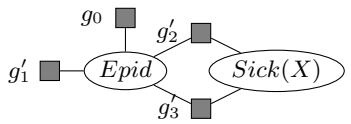
Chapter 8: LJT finds a subtree that covers the (parameterised) query terms, extracts a submodel G', and then uses LVE for parameterised queries to answer the query on G'.

Algorithm 5 shows LVE for parameterised queries with input model G, query terms $\mathbf{Q}$, and evidence $\mathbf{E}$ to answer the query $P(\mathbf{Q}|\mathbf{E})$. Lines 2 to 8 are identical to the previous LVE version. LVE shatters G on $\mathbf{Q}$ and absorbs $\mathbf{E}$ in G. Then, LVE eliminates all non-query randvars (lines 4 to 8). Lines 9 to 13 contain steps specific to parameterised queries and materialise remaining logvars in the result. The final step is normalisation, which also occurs in the previous LVE version and is the step that LJT skips for messages. For parameterised queries, LVE needs to account for CRVs occurring in the result.

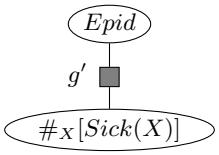
The following paragraphs detail the tasks of eliminating non-query terms, inducing a joint distribution, and normalising a result. Afterwards, we take a closer look at evidence.

Eliminating Non-query Terms Parameterised queries no longer cause groundings w.r.t. domain values appearing in a query during shattering, thus, allowing for more lifted eliminations. Of course, shattering may still result in many parfactors to be split given multiple query PRVs with logvars using subdomains of model logvars. During elimination, a parameterised query works as a compact representation of the terms not to eliminate. The application of LVE operators is not affected, still aimed at eliminating non-query terms. Let us look at eliminating non-query terms for the query $P(Sick(X)_{|\top})$.

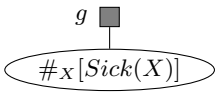
Example 9.2.1 (Elimination)**.** The query term is $Sick(X)$. Using parcluster $\mathbf{C}_3$ with messages as in Example 9.1.1, the submodel is $G' = G_3 \cup m_{23} = \{g_3, g_2', g_0, g_1'\}$. LJT uses LVE to answer the query, passing along G', $\{Sick(X)\}$, and $\emptyset$ as inputs. Shattering



(a)



(b)



(c)

$\#_X[Sick(X)]$	ϕ
$[0,3]$	v_0
$[1,2]$	v_1
$[2,1]$	v_2
$[3,0]$	v_3

(d)

Figure 9.2: Evolution of G' while answering $P(Sick(X)_{|\top})$ and the result in a table

G' on the query term $Sick(X)$ does not change G' as the query constraint coincides with the constraints regarding X in the model. As there is no evidence to absorb, LVE eliminates $Treat(X, M)$ from g_3, resulting in g_3' with arguments $Sick(X)$ and $Epid$ as shown in Fig. 9.2a. To eliminate $Epid$, all occurrences of $Epid$ need to be combined into one parfactor: LVE multiplies g_0 and g_1' into g_{01} and g_2' and g_3' into g_{23}. Since g_{23} stands for $n = 3$ factors for each instance of g_{01} due to X with its three constants, when multiplying g_{01} and g_{23}, LVE needs to rescale the potentials from g_{01} by taking the n-th root with $n = 3$. The result is a parfactor g' with arguments $Epid$ and $Sick(X)$. In g', LVE cannot eliminate $Epid$ as $Epid$ does not contain X. To enable the elimination, LVE counts X in g'. Figure 9.2b shows a parfactor representation of the result. Now, LVE eliminates $Epid$. The end result is a parfactor $\phi(\#_X[Sick(X)])$, which no longer contains non-query terms, depicted in Fig. 9.2c.

The example query uses the same logvar and domain as the logvar in the model for readability. A query term referencing a subdomain of X would be $\{Sick(Y)\}_{|C}$ with $C = (Y, \{(alice), (bob)\})$. In this case, LJT still extracts G' as a submodel. Shattering leads to two versions of g_3 and g_2', one for $Sick(Y)$ under C and one for eve. Elimination works as before with a result parfactor $\phi(\#_Y[Sick(Y)_{|C}])$ in which $\#_Y[Sick(Y)_{|C}]$ has the range values $[0, 2], [1, 1], [2, 0]$. LVE for parameterised queries shares similarities with LVE that uses on-demand shattering. On-demand shattering shatters a model only when non-query terms are no longer eliminable. Given a query over interchangeable query terms, on-demand shattering saves the shattering effort at the beginning and salvages lifted computations as well. But, eventually, shattering occurs, possibly grounding the remaining model and yielding an overblown result representation. LVE for parameterised queries also saves the shattering effort at the beginning. But, LVE only performs a grounding if preconditions enforce it.

Inducing a Joint Distribution After eliminating all non-query terms, G may still be a set of parfactors. Each parfactor may contain logvars and thus, represent a set of local factors and not a joint distribution over query terms. Count conversions for the remaining logvars in G materialise the query terms in the result to encode a joint distribution instead

of local distributions. Forming CRVs of the remaining PRVs also enables a compact result representation. But, count conversions have preconditions as well. If the logvars are not countable, LVE applies one of the remaining operators to enable a count conversion in contrast to enabling a lifted summing out, which no longer applies. In a worst case, LVE has to ground to ensure a joint distribution over all query terms.

Example 9.2.2 (Joint distribution)**.** At the end of Example 9.2.1, LVE has produced a single parfactor $g = \phi(\#_X[Sick(X)])$ for the query $P(Sick(X))$. The parfactor also no longer contains any logvars since logvar X has already been counted during the elimination of non-query terms. Thus, no further operations are required.

Assume elimination outputs a parfactor $\phi(Sick(X))_{|C}$ with the mappings $true \mapsto t_1$ and $false \mapsto t_2$. The mappings hold for each $x \in gr(X_{|C})$. To build a joint distribution, one could ground X, multiply the grounded factors, and would come to a similar result as seen in Fig. 9.1b. To avoid the blowup, LVE aims at applying count conversions to build CRVs for the query terms. With X as the single logvar in a single PRV, X is trivially countable. Counting X, LVE produces a parfactor $\phi(\#_X[Sick(X)])$.

A parameterised query does not necessarily yield a result containing CRVs for exactly the PRVs and constraint in the query. While eliminating all non-query PRVs as well as inducing a joint distribution, the query PRVs in the model may be affected by operators rewriting constraints (splits, groundings). They may appear fully grounded in the result simply through the application of operators to compute a correct result. The LVE operators that usually enable a lifted summing out can have the following effects visible in the result if applied to a PRV or logvar that is (partially) covered by a query term:

- Count conversion: enable a compact result representation (as in Example 9.2.1)
- Splitting, expansion (i.e., splitting for CRVs): may lead to groups in the result (see next paragraph for an example w.r.t. evidence)
- Grounding: leads to grounded query terms in the result

Though the LVE operators may be applied because the model demands an application, some parameterised queries ultimately lead to groundings, which means LVE produces a result for the parameterised query grounded out. We further discuss such query-induced groundings during the completeness analysis in Section 9.3.

Parameterised queries already appear in LJT in a certain way. For a message, LJT performs LVE with the local model plus other messages as the input model and the separator as the (parameterised) query. Inducing a joint distribution is the difference between a parameterised query and a message. For a message, LJT does not count or ground logvars but keeps the information as is, i.e., LJT does not establish a joint distribution. As messages are specified over PRVs with the same constraints in the parcluster and the separator (due to construction), shattering does not occur during message calculation and logvars of the same name refer to the same restricted domain.

Normalising the Result The result after eliminating non-query terms and building a joint distribution is not yet normalised. To normalise the resulting parfactor, LVE cannot simply divide the potentials by the sum of all potentials if CRVs are involved. A histogram h stands for $Mul(h)$ assignments, with $Mul(h)$ referring to a multinomial coefficient as defined in Eq. (3.1). Thus, normalising a potential w_i in a mapping $h_i \mapsto w_i$ with m overall mappings results in a normalised potential v_i as follows

$$v_i = \frac{w_i}{\sum_{i=0}^{m-1} Mul(h_i) w_i} \tag{9.1}$$

The resulting v_i only add up to 1 if multiplying $Mul(h_i)$ with v_i. To illustrate normalisation, let us continue our example of answering $P(Sick(X))$.

Example 9.2.3 (Normalisation)**.** At the end of Example 9.2.1, LVE has produced a single parfactor $g = \phi(\#_X[Sick(X)])$, with mappings of the form $[0,3] \mapsto w_0$, $[1,2] \mapsto w_1$, $[2,1] \mapsto w_2$, and $[3,0] \mapsto w_3$, in which the histograms $[1,2]$ and $[2,1]$ stand for $\frac{(1+2)!}{1!\cdot 2!} = \frac{(2+1)!}{2!\cdot 1!} = 3$ assignments. Thus, the normalised potentials are given by $w_i/(w_0 + 3 \cdot w_1 + 3 \cdot w_2 + w_3)$ which is then equal to the v_i given in Fig. 9.2d. Assuming that $w_0 = 1$, $w_1 = 2$, $w_2 = 3$, and $w_3 = 4$, the divisor is given by

$$\begin{aligned} & Mul([0,3]) \cdot 1 + Mul([1,2]) \cdot 2 + Mul([2,1]) \cdot 3 + Mul([3,0]) \cdot 4 \\ = \quad & 1 \cdot 1 + \quad 3 \cdot 2 + \quad 3 \cdot 3 + \quad 1 \cdot 4 = 20 \end{aligned}$$

The normalised potentials follow by dividing the w_i by 20, e.g., $[0,3]$ maps to $\frac{1}{20} = 0.05$:

$$[0,3] \mapsto 0.05,\ [1,2] \mapsto 0.1,\ [2,1] \mapsto 0.15,\ [3,0] \mapsto 0.2$$

E.g., $[1,2] \mapsto 0.1$ provides a probability (0.1) for each of the three assignments behind $[1,2]$. All three assignments together have a probability of 0.3. After multiplying the multinomial coefficients with the potentials, the sum is 1:

$$\begin{aligned} & Mul([0,3]) \cdot 0.05 + Mul([1,2]) \cdot 0.1 + Mul([2,1]) \cdot 0.15 + Mul([3,0]) \cdot 0.2 \\ = \quad & 1 \cdot 0.05 + \quad 3 \cdot 0.1 + \quad 3 \cdot 0.15 + \quad 1 \cdot 0.2 = 1 \end{aligned}$$

Normalisation concludes LVE for parameterised queries. Next, we discuss evidence and its effect on a result parfactor, before moving on to a theoretical discussion of parameterised queries.

Evidence Evidence may lead to groups of constants in a model, e.g., when observations exist for some of the constants of a logvar domain, leading to groups in a model through splits. The groups may surface in a result. We consider the following cases for query terms $\mathbf{Q}_{|C}$ and evidence PRV $E(\mathbf{X})_{|C_E}$.

(i) $\mathbf{Q}_{|C} \cap E(\mathbf{X})_{|C_E} \neq \emptyset$

(ii) $\mathbf{Q}_{|C} \cap E(\mathbf{X})_{|C_E} = \emptyset \wedge gr(\pi_{\mathbf{X}}(C)) \cap gr(C_E) \neq \emptyset$

(iii) $\mathbf{Q}_{|C} \cap E(\mathbf{X})_{|C_E} = \emptyset \wedge \pi_{\mathbf{X}}(C) \cap gr(C_E) = \emptyset$

In the first case, instances of the evidence PRV appear in a parameterised query. The posterior probability of the instances of $E(\mathbf{X})_{|C_E}$ is 1. LVE absorbs evidence in a parfactor by eliminating all mappings in which the range value of $E(\mathbf{X})_{|C_E}$ is unequal to the observed value and dropping $E(\mathbf{X})_{|C_E}$ from the parfactor arguments. Thus, the instances of $E(\mathbf{X})_{|C_E}$ appearing in $\mathbf{Q}_{|C}$ do not appear in the result since LVE drops them during lifted absorption. Let us look at query $P(Sick(X)|sick(eve))$, which contains the evidence previously used in the running example and $Sick(X)$ as a query term.

Example 9.2.4 (Evidence in Query Terms)**.** The query term $Sick(X)$ covers the evidence term $sick(eve)$ under the $\top$ constraint. The evidence leads to splits (results for local models as given Fig. 4.3), with message passing proceeding as described in Chapter 4. Eliminating non-query terms from $G_3 \cup m_{23}$ results in a parfactor with argument $\#_X[Sick(X)]$ under a constraint $(X, \{(alice), (bob)\})$, mapping the histograms $[0, 2]$, $[1, 1]$, and $[2, 0]$ to unnormalised potentials, which LVE then normalises:

$$[0, 2] \mapsto v_0,\ [1, 1] \mapsto v_1,\ [2, 0] \mapsto v_2. \tag{9.2}$$

One could artificially insert $sick(eve)$ into the result (or keep it during absorption with potentials set to 0) and have the following result for arguments $sick(eve), \#_X[Sick(X)]$:

$$\begin{aligned}(\neg sick(eve), [0, 2]) \mapsto 0,\ (\neg sick(eve), [1, 1]) \mapsto 0,\ (\neg sick(eve), [2, 0]) \mapsto 0\\ (sick(eve), [0, 2]) \mapsto v_0,\ \ (sick(eve), [1, 1]) \mapsto v_1,\ \ (sick(eve), [2, 0]) \mapsto v_2.\end{aligned}$$

The first line with $\neg sick(eve)$ has mappings to potentials of 0. The second line has mappings for the original potentials, the inputs extended by $sick(eve)$ as in the evidence. Both lines together hold the same information as Expression (9.2). Summing out $\#_X[Sick(X)]$ leads to a posterior probability of 1 for $sick(eve)$. I.e., $P(Sick(X)|sick(eve))$ and $P(Sick(Y)_{|C}|sick(eve))$ with constraint $C = (Y, \{(alice), (bob)\})$ have the same result.

The second case involves constants of an evidence PRV appearing in query terms that reference PRVs without evidence. Given such a setup, the result has groups, one for constants that also appear in evidence and one for the constants that do not appear in evidence, which is a consequence of the splitting due to evidence. Let us look at query $P(Travel(X)|sick(eve))$ with evidence $sick(eve)$ and query term $Travel(X)$.

Example 9.2.5 (Evidence constants in a query)**.** While query term and evidence reference different PRVs, their constants overlap in the form of $\{eve\}$. $Travel(X)$ occurs in parcluster $\mathbf{C}_2$, in the form of $Travel(eve)$ and $Travel(X)$ with a constraint $C =$

$(X, \{(alice), (bob)\})$. LVE eliminates all non-query terms from submodel $G_2 \cup m_{12} \cup m_{32}$. The result is a parfactor with arguments $Travel(eve)$ and $\#_X[Travel(X)]$, constraint C, and mappings of the form

$$(\neg travel(eve), [0, 2]) \mapsto v_0, (\neg travel(eve), [1, 1]) \mapsto v_1, (\neg travel(eve), [2, 0]) \mapsto v_2$$
$$(travel(eve), [0, 2]) \mapsto v_3, \ \ (travel(eve), [1, 1]) \mapsto v_4, \ \ (travel(eve), [2, 0]) \mapsto v_5.$$

The result has a "group" for *eve* and a group for *alice* and *bob*. With larger domains, the result has two groups as well. Assume 100 constants for X and the observations $\{Sick(x_i) = true\}_{i=1}^{10}$ stored in an evidence parfactor $\phi_e(Sick(X'))_{|C_e}$. Constraint C_e holds that X' is restricted to the constants $x_1, \ldots, x_{10}$ as in Example 3.1.6. The submodel has the same structure as before but with $Travel(X)$ appearing as $Travel(X')_{|C_e}$ and $Travel(X)_{|C}$, where C restricts X to the remaining constants $x_{11}, \ldots, x_{100}$. The result of eliminating all non-query terms is a parfactor $\phi(\#_{X'}[Travel(X')], \#_X[Travel(X)])_{|C_e \bowtie C}$, which explicitly shows the two groups emerging from evidence.

In the last case, evidence PRVs and constants have no connection to PRVs or constants in a query. The effect of evidence is hidden in calculations. Consider the query $P(Nat(D)|sick(eve))$ with evidence $sick(eve)$ and query term $Nat(D)$.

Example 9.2.6 (Hidden evidence)**.** Given a query term $Nat(D)$, LVE uses parcluster $\mathbf{C}_1$ to compute a result parfactor with argument $\#_D[Nat(D)]$. The evidence is already hidden from $\mathbf{C}_1$ as $Sick(eve)$ does not appear in $\mathbf{C}_1$ and the separator with neighbour $\mathbf{C}_2$ does not contain logvar X. Thus, the evidence does not influence the result's structure, but possibly the potentials only, and thus, the effect of $sick(eve)$ is hidden.

The effects of evidence can also be observed in messages: Evidence PRVs may disappear or groups may appear because of evidence covered by a separator. A result allows for some immediate interpretation regarding groups in a model and, e.g., their most likely assignment. Next, we argue why LVE for parameterised queries is sound and we analyse the consequences that parameterised queries have for completeness and complexity.

9.3 Theoretical Discussion

This discussion considers correctness and complexity aspects of LVE and LJT for parameterised queries. It builds on the results of the theoretical analysis in Section 8.2. First, we show that LVE as extended in this chapter is sound, which makes LJT using the extended LVE version sound. Then, we discuss completeness w.r.t. parameterised queries, characterising a new set of liftable queries as well as grounding queries. Last, we consider the effect of parameterised queries on runtime complexity.

Theorem 9.3.1. *LVE is sound given a query with parameterised query terms $\mathbf{Q}_{|C}$, i.e., computes a result equivalent to a query with query terms $gr(\mathbf{Q}_{|C})$.*

Proof. We assume that the original LVE in the form of the operator suite defined by Taghipour *et al.* (2013c) is sound. A parameterised query $\mathbf{Q}_{|C}$ consists of PRVs $\mathbf{Q}$ and a constraint C. It correctly represents a query over interchangeable objects since constraints allow for representing instances of a PRV. With the same syntactical constructs for query terms as for models, LVE is able to interpret $\mathbf{Q}_{|C}$ and eliminate all randvars not occurring in $\mathbf{Q}_{|C}$. The remaining steps until normalisation consist of applying sound LVE operators. Thus, the results of shattering, non-query term elimination, and joint distribution induction are correct. The final result parfactor over $\mathbf{Q}_{|C}$ consists of CRVs for parameterised query terms or grounded query terms. As $Mul(h)$ correctly represents the number of assignments of a histogram h (Taghipour *et al.*, 2013c), the normalisation correctly incorporates multiple assignments of a histogram for its normalisation as defined in Expression (9.1), leading to a correct joint probability distribution. Grounding out the CRVs in the result leads to an answer equivalent to one computed for $gr(\mathbf{Q}_{|C})$. □

Completeness If an algorithm is complete w.r.t. a class of models, a lifted algorithm run is possible given any model from that class. A run is lifted if a runtime depends polynomially on the domain sizes of the model logvars. LVE and LJT are complete for two-logvar models $\mathcal{M}^{2lv}$ and models of one-logvar PRVs $\mathcal{M}^{prv1}$ given the query class $\mathcal{Q}$ of single ground query terms as well as the query class $\mathcal{CQ}^{lift}$ of query terms with at most one constant of each logvar. The query class $\mathcal{PCQ}$ contains all parameterised conjunctive queries. Though parameterised queries aim at making queries with interchangeable query terms liftable, completeness results for $\mathcal{PCQ}$ are negative as a query may still induce groundings by blocking a reasonable elimination order or by having a constraint that causes groundings. The following theorem formulates the negative result.

Theorem 9.3.2. *LVE and LJT are not complete for the query class $\mathcal{PCQ}$ of parameterised conjunctive query terms and all model classes that contain at least one logvar.*

Proof by counterexample. Assume both algorithms are complete, i.e., have runtimes polynomial in domain sizes for all possible models with at least one logvar. Consider parameterised conjunctive query $P(Sick(X_1)_{|C_1}, Travel(X_2)_{|C_2})$, $C_1 = ((X_1), \{(alice), (eve)\})$ and $C_2 = ((X_2), \{(eve), (bob)\})$. LJT answers the query using parcluster $\mathbf{C}_2$ of the FO jtree for G_{ex} and passes $\mathbf{Q} = \{Sick(X_1), Travel(X_2)\}_{|C_1 \bowtie C_2}$ and $G' = G_2 \cup m_{12} \cup m_{32}$ to LVE. Shattering G' on $\mathbf{Q}$ leads to splitting off *bob* and then *alice* in g_2 and m_{32}, which grounds X, no longer permitting a lifted solution to the query. Therefore, LVE and LJT are not complete for $\mathcal{PCQ}$ and models with at least one logvar. □

The results of such grounding queries are still correct. Only, query PRVs may be grounded in the result. The counterexample of the proof works with constraints to induce a grounding. For $\mathcal{M}^{2lv}$, there is a counterexample using logvars to induce a grounding.

Example 9.3.1. Consider a model $\{\phi(Q(X,Y),R(X),S(Y))\} \in \mathcal{M}^{2lv}$ and a query term $\{Q(X,Y)\} \in \mathcal{PCQ}$. LVE needs to eliminate $R(X)$ and $S(Y)$ but neither PRV contains both logvars. Additionally, neither X nor Y are countable as they appear twice in the arguments of ϕ. So, LVE grounds X or Y and thus, does not run in time polynomial in the domain size of one of the two logvars. The reason lies in a precondition for lifted summing out of a PRV A: A has to contain all logvars in a parfactor. If PRVs to eliminate contain fewer logvars than a query PRV and no count conversion applies, LVE grounds a logvar. The same applies to a conjunctive query with query terms $\{Q(X,Y),R(X),S(Y)\}$. Here, LVE grounds a logvar when inducing a joint distribution.

The example model allows for a lifted run given a liftable query $Q(x_1,y_1)$ as LVE eliminates $Q(X,Y)$ after splitting off $Q(x_1,y_1)$ before eliminating $R(X)$ and $S(Y)$. Note that both $R(X)$ and $S(Y)$ as a query term allow for a lifted run since LVE eliminates $Q(X,Y)$ first, then counts the logvar of the PRV in the query, and lastly, eliminates the remaining non-query PRV. In fact, positive completeness results are possible if restricting query terms to containing one logvar per PRV and one set of constants per logvar. The example uses two logvars in one PRV and the counterexample from the proof uses two sets of constants for logvar $X \in lv(G')$.

Definition 9.3.1. Query class $\mathcal{PCQ}^{lift}$ refers to query terms with one logvar per PRV and one set of constants per logvar.

Theorem 9.3.3. *LVE and LJT are complete for the query class $\mathcal{PCQ}^{lift}$ and the model classes $\mathcal{M}^{2lv}$ and $\mathcal{M}^{prv1}$.*

Proof. To show that LVE is complete for the query class $\mathcal{PCQ}^{lift}$, we need to show that LVE is domain-lifted for all queries in $\mathcal{PCQ}^{lift}$.

When LVE preemptively shatters a model on a parameterised query, each query term appearing in a parfactor leads to at most one split, without any groundings. After shattering, LVE eliminates all non-query terms with two logvars first since $\mathcal{PCQ}^{lift}$ does not contain query terms with two logvars. Eliminating two-logvar PRVs either uses standard lifted summing out or involves an operator called group inversion, defined by Taghipour *et al.* (2013d), which allows for eliminating two-logvar PRVs from a parfactor with an inequality constraint such as $\phi(Friends(X,Y),Friends(Y,X))_{|C}$, where C encodes $X \neq Y$. After eliminating all two-logvar PRVs, the model contains only one-logvar PRVs and ground randvars. Such a model falls into $\mathcal{M}^{prv1}$.

The remainder of this proof is based on the proof of the completeness results of LVE for $\mathcal{M}^{prv1}$ by Taghipour (2013). One is able to count all logvars in a model consisting of only one-logvar PRVs given the generalised count conversions and elimination procedures by Taghipour and Davis (2012). We already mentioned the generalisations in Section 5.2. For full definitions and examples, refer to the work by Taghipour and Davis (2012). To briefly summarise, the generalisations allow for

- counting logvars that appear in more than one PRV,
  e.g., logvars X and Y in parfactor $\phi(Q(X), R(X), S(Y), T(Y))$,
  resulting in a parfactor $\phi(\#_X[Q(X), R(X)], S(Y), T(Y))$ after counting X,
- merging CRVs with counted logvars of the same domain into one CRV,
  e.g., the CRVs in parfactor $\phi(\#_X[Q(X), R(X)], \#_Y[Q(Y), R(Y)])$, $\mathcal{D}(X) = \mathcal{D}(Y)$,
  yielding a parfactor $\phi(\#_X[Q(X), R(X)])$ after merging, and
- merge-counting a PRV and a CRV with an inequality constraint into one CRV,
  e.g., CRV $\#_X[Q(X)]$ and PRV $R(Y)$, $X \neq Y$, in a parfactor $\phi(\#_X[Q(X)], R(Y))$,
  leading to a parfactor $\phi(\#_X[Q(X), R(X)])$.

For a CRV over multiple PRVs, an operator exist to eliminate individual PRVs, meaning, it is possible to eliminate PRV $Q(X)$ or $R(X)$ from a CRV $\#_X[Q(X), R(X)]$. The generalised count conversions preserve original PRVs within a CRV, apart from a renaming of a logvar, allowing for identifying them as a query term.

Using generalised counting, LVE counts all logvars, multiplies the resulting parfactors into one, and merges CRVs with counted logvars of the same domain. The result is one parfactor with CRVs as arguments and counted logvars of disjoint domains,

$$\phi(\#_{X_1}[R_{1,1}(X_1), \ldots, R_{1,k_1}(X_1)], \ldots, \#_{X_m}[R_{m,1}(X_m), \ldots, R_{1,k_m}(X_m)], \mathcal{A}_0)$$

where $\mathcal{D}(X_i) \neq \mathcal{D}(X_i)$ and $\mathcal{A}_0$ contains ground randvars. Since all logvars are counted, LVE can eliminate CRVs, PRVs within CRVs, and ground randvars alike. Elimination results in a parfactor ϕ, which, containing only CRVs for the query terms, represents a joint distribution. The remaining task is normalisation, which only manipulates potentials. As LJT performs LVE for query answering, the result extends to LJT as well. □

The proof shows that once a model contains PRVs with at most one logvar, LVE is domain-lifted given any query regarding the remaining PRVs. We can use the results so far to identify grounding queries as well as liftable queries for any model.

A query in itself can cause a problem if query terms share some but not all logvars and co-occur in a parfactor before their logvars are counted, inhibiting a count conversion.

Proposition 9.3.1. *If* $\exists Q_1, Q_2 \in \mathbf{Q}$ *and* $\exists g \in G$ *with* $Q_1, Q_2 \in rv(g)$ *s.t.*

$$(lv(Q_1) \cap lv(Q_2) \neq \emptyset) \wedge ((lv(Q_1) \cap lv(Q_2)) \subset lv(Q_1) \vee (lv(Q_1) \cap lv(Q_2)) \subset lv(Q_2)), \tag{9.3}$$

then Q_1 *and* Q_2 *cause groundings during inducing a joint distribution.*

Expression (9.3) formalises that query terms such as $\{Q(X, Y), R(X)\}$ cause groundings in combination with a parfactor $\phi(Q(X, Y), R(X), S(X))$. With $Q_1 = Q(X, Y)$, $lv(Q_1) = \{X, Y\}$, and $Q_2 = R(X)$, $lv(Q_2) = \{X\}$, both conjuncts are true as $\{X\} \cap$

$\{X, Y\} = \{X\} \neq \emptyset$ and $\{X\} \subset \{X, Y\}$. Given $Q_1 = R(X)$, LVE would count logvar Y, eliminate $S(X)$, and then start inducing a joint distribution in $\phi(\#_Y[Q(X, Y)], R(X))$. Logvar X appears twice, preventing a classical count conversion. Generalised counting is also not possible as the generalised version requires that no other counted logvar exists in the PRVs that are converted into a combined CRV. Here, Y is counted, inhibiting a count conversion and thus leading to a grounding of X. The same holds with $Q_2 = S(X)$. Therefore, a combination of $Q(X, Y)$ with any of the other two PRVs causes groundings.

For a grounding query within a model, groundings occur if a query term occurs with a specific constellation of logvars in the model as seen in Example 9.3.1.

Proposition 9.3.2. *If* $\exists Q \in \mathbf{Q}, \exists g \in G, Q \in rv(g), \exists A, B \in rv(g), \mathbf{X}_A \coloneqq lv(A) \cap lv(Q), \mathbf{X}_B \coloneqq lv(B) \cap lv(Q)$ *s.t.* $\mathbf{X}_A \neq \emptyset \wedge \mathbf{X}_B \neq \emptyset \wedge \mathbf{X}_A \not\subseteq \mathbf{X}_B \wedge \mathbf{X}_B \not\subseteq \mathbf{X}_A$*, then* Q *causes groundings.*

Groundings occur if a query term Q has more logvars than two non-query PRVs A, B that co-occur with Q and contain different logvars from Q. To eliminate A (B), a count conversion of the query logvars not in A (B) is necessary. But, B (A) contains at least one of those query logvars, meaning the double occurrence (in Q and B/A) prohibits the count conversion. For $\phi(Q(X, Y), R(X), S(Y))$ and $\{Q(X, Y)\}$, the condition in Proposition 9.3.2 is true as $\{X\}$ is not subset of $\{Y\}$ and vice versa.

If either condition in Propositions 9.3.1 and 9.3.2 is true for an input model and a query, we know the query induces groundings before LVE has even started. If the condition is not true for an input model, it may become true for intermediate factors during LVE, i.e., the query still induces groundings. However, the groundings occur later than during shattering on a grounded query, salvaging as many lifted computations as possible.

Inverting and generalising Proposition 9.3.1 characterises liftable queries. (Groundings may still occur due to the model itself.)

Proposition 9.3.3. *If* $\forall Q_1, Q_2 \in \mathbf{Q}$ *with* $\mathbf{L} = lv(Q_1) \cap lv(Q_2)$ *and* $\forall g \in G$ *with* $Q_1, Q_2 \in rv(g)$ *holds* $(\mathbf{L} = \emptyset) \vee (lv(Q_1) = lv(Q_2))$*, then* $\mathbf{Q}$ *does not cause groundings in itself.*

The logvars of query terms need to be either identical or disjoint. Given a parfactor $\phi(Q(X, Y), R(X), S(Y))$, the query terms $R(X), S(Y)$ with disjoint logvars do not cause groundings, allowing LVE to eliminate $Q(X, Y)$ and count both logvars. Given a parfactor $\phi(Q(X, Y), R(X), S(X))$, containing $S(X)$ instead of $S(Y)$, the query terms $R(X), S(X)$ do not cause groundings, either. One only has to bear in mind that even though queries over $Q(X, Y), R(X, Y)$ conform with the given condition, the existing count conversions do not handle counting multiple logvars at once.

The condition in Proposition 9.3.2 inverted postulates for all query terms $Q \in \mathbf{Q}$ that for all PRVs A, B co-occurring with Q, the logvars from Q in A, B need to be subsets of each other to not cause groundings in a model or intermediate factors. For the following proposition, we consider all PRVs in a model and not just PRVs co-occurring with a query term due to possible co-occurrences in an intermediate factor.

Proposition 9.3.4. *If* $\forall Q \in \mathbf{Q}, \forall A, B \in rv(G), \mathbf{X}_A \coloneqq lv(A) \cap lv(Q), \mathbf{X}_B \coloneqq lv(B) \cap lv(Q)$ *holds* $\mathbf{X}_A \subseteq \mathbf{X}_B \vee \mathbf{X}_B \subseteq \mathbf{X}_A$*, then* $\mathbf{Q}$ *does not cause groundings in* G*.*

The condition in Proposition 9.3.4 is true for query terms with no logvar as $\mathbf{X}_A = \mathbf{X}_B = \emptyset$. Query terms with one logvar also fulfil the condition as $\mathbf{X}_A$ and $\mathbf{X}_B$ are either empty or contain the one query logvar, which means either both sets are equal or an empty set is a subset of the other set. The following is a more elaborate example.

Example 9.3.2. Consider a model with a single parfactor $\phi(Q(X), R(X,Y), S(X,Y,Z))$ and a query term $\{S(X,Y,Z)\}$. PRVs $A = Q(X)$ and $B = R(X,Y)$ share logvars with $Q = S(X,Y,Z)$, with $\mathbf{X}_A = \{X\}$ and $\mathbf{X}_B = \{X,Y\}$. As $\{X\} \subset \{X,Y\}$, the condition in Proposition 9.3.4 holds. Assume that Z fulfils the preconditions of a count conversion w.r.t. $\{X,Y\}$. LVE counts Z, converting $S(X,Y,Z)$ into $\#_Z[S(X,Y,Z)]$, to first eliminate $R(X,Y)$ and then $Q(X)$ in $\phi'(Q(X), R(X,Y), \#_Z[S(X,Y,Z)])$, yielding $\phi''(\#_Z[S(X,Y,Z)])$. LVE counts X and Y and normalises the result, providing a compact answer without groundings. The example also works with additional logvars in A and B, e.g., $\hat{\phi}(Q(X,W), R(X,Y,W), S(X,Y,Z))$.

Proposition 9.3.4 covers all PRVs in G. But, one does not need to be as exhaustive. One only needs to consider the PRVs in the submodel for a query. With singleton queries, Proposition 9.3.4 regards a single parcluster $\mathbf{C}_i$ instead of $rv(G)$. Parclusters and separators may also help to assess possible groundings. Given a query over terms $Q(X,Y)$ and $R(X)$ and both query terms appearing in different parclusters, such a query might be liftable as a count conversion may be applicable for X and Y in one parcluster while X is countable in the other, which may allow LVE to avoid grounding.

As a side effect, a lifted run yields a compact result representation through CRVs. A blowup of the result occurs whenever LVE has to ground a logvar. Given a lifted run, no grounding is necessary while eliminating non-query terms, and the result is compactly represented through CRVs, which have a size polynomial in the domain size of the logvars in the query. We formulate the observation as a secondary result:

Corollary 9.3.1. *The result of a parameterised query with a lifted solution has a size polynomial in the domain sizes of the query logvars.*

After characterising grounding as well as liftable queries, we turn to discuss the effects of parameterised queries on complexity.

Complexity For single ground query terms, which requires a single parcluster for query answering with LJT and a bounded number of splits for LVE, the runtime complexity of LJT given a model G with an FO jtree J is $O(\log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ with $(w_g, w_{\#})$ being the lifted width of J, n the largest domain size in $lv(J)$, r the largest range size among the PRVs in J, $n_{\#}$ the largest domain size among the counted logvars, and $r_{\#}$ the largest range size among the PRVs in the CRVs. In Section 8.2, we have shown

that the combined complexity of answering a conjunctive query from $\mathcal{CQ}^{lift}$ with LJT is $O(n'_J \cdot \log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ where n'_J is the number of parclusters in the corresponding subtree of the submodel to answer a particular query. With parameterised queries, queries are conjunctions of parameterised query terms. Regarding conjunctions, the result from Section 8.2 still holds since the number of parclusters combined into a submodel influences the complexity of answering a single query as before. When answering a query, LVE for parameterised queries eliminates non-query terms followed by inducing a joint distribution, which means the application of count conversions as we only consider liftable queries. A lifted summing out and a count conversion have the same complexity (Taghipour, 2013), i.e., $O(\log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$. A PRV appearing in a query leads LVE to not eliminate such a term but to count-convert it, which exchanges one operation with another of the same complexity. Thus, the overall complexity as shown for liftable conjunctive queries still holds.

Comparing a conjunctive query over k interchangeable query terms to an equivalent parameterised query, one can see the positive effect of parameterised queries. The complexity of QA for the ground query is exponential in k as LVE grounds a logvar with k constants whereas the complexity of QA for the parameterised query lies in $O(\log_2(n) \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ as $n'_J = 1$, a fact which can transfer into a runtime being multiple orders of magnitude smaller. To illustrate the effect, consider the query term $\{Sick(X)_{|\top}\}$ as well as its grounding $\{Sick(alice), Sick(eve), Sick(bob)\}$ in G_{ex}.

Example 9.3.3. In G_{ex}, $k = n = 3$, $n_{\#} = 2$, and $r_{\#} = 2$. To answer the given query, LJT chooses parcluster $\mathbf{C}_3$, which means $n'_J = 1$, $w_g = 3$, and $w_{\#} = 0$. For the ground version, we have a worst-case scenario of

$$2^{(k-1)+w_g} = 2^{(3-1)+3},$$

which shows that k outweighs w_g for larger domains. The parameterised version bypasses k as an exponent, leading to a worst-case scenario of

$$\log_2(n) \cdot 2^{w_g} = \log_2 3 \cdot 2^3.$$

The actual largest size of a parfactor during LVE is 16 for the ground version and 8 for the parameterised version.

Intermezzo: Queries in PDBs Queries like $\{Q(X,Y)\}$ or $\{Q(X,Y), R(X), S(Y)\}$ on a parfactor $\phi(Q(X,Y), R(X), S(Y))$ cause LVE to ground. The latter query matches a non-hierarchical PDB query. Dalvi and Suciu (2012) show that such "forbidden queries" over PDBs are #P-hard. Overlapping logvars in a parameterised query or its affected parfactors make a query in our context hard in a similar vein.

PDBs enable us to answer queries such as $Treat(eve, m)$ with m as a free variable, using a set of rules and a PDB as a source for basic facts, creating facts like

$Treat(eve, injection)$ and $Treat(eve, tablet)$ with inferred probabilities. Rewriting rules into a parameterised model and entering facts as evidence, we use parameterised queries to answer corresponding queries, e.g., $\{Treat(eve, M)\}$. If facts lead to different probabilities for some m, the result mirrors the difference, distinguishing $Treat(eve, injection)$ and $Treat(eve, tablet)$. While the PDB scenario infers facts based on the entries in the PDB, parameterised models include domain information. Assuming $|\mathcal{D}(M)| = 5$ and PDB facts about *injection* and *tablet*, all five medicines influence the result.

To completely model basic facts in a PDB that have a probability < 1.0 as evidence, we need a straightforward extension of evidence parfactors, which usually assign a probability of 1 to an observed value and 0 to the remaining values, by assigning some probability distribution. Absorbing evidence would change potentials according to the given distribution. Gehrke *et al.* (2019d) provides a formal definition. Extending evidence with a random distribution is a reasonable step for other scenarios as well, considering noisy channels, measurement errors, or other influences that make an observation less reliable.

From uncertain evidence, we go back to the topic of this chapter and take a look at an empirical evaluation of parameterised queries, including certain evidence.

9.4 Empirical Evaluation

We pick up the previous evaluation regarding grounding queries and show how parameterising queries allows for speeding up runtimes. Additionally, we look at the influence of evidence. We have extended the LVE implementation by Taghipour to answer parameterised queries and adapted the LJT implementation to incorporate the extended LVE implementation. The FOKC implementation by Van den Broeck does not allow for parameterised queries. Therefore, FOKC is not part of this evaluation. The input model is identical to the one used in the previous evaluation, which has an FO jtree with $n_J = 10$ parclusters and a lifted width of $(3, 1)$.

Parameterised Query The evaluation in Section 8.3 regarding grounding conjunctive queries concerns runtime behaviour of a query $\mathbf{Q} = \bigcup_{x \in gr(X)} \{R(x)\}$ for some PRV $R(X)$ in the input model. This part of the evaluation introduces $R(X)$ as the parameterised equivalent to the grounding query. The parameterised query promises a runtime polynomial in the domain size in contrast to the runtime exponential in the domain size of the grounding query. We also compare the parameterised query against a singleton ground query $R(x), x \in gr(X)$ to see how much more effort a parameterised query really brings. Therefore, we look at runtimes w.r.t. increasing domain sizes, comparing runtimes of the different query types and shattering modes.

Figure 9.3 shows runtimes in milliseconds [ms] on a log scale for QA with LJT (hollow symbols) and LVE (filled symbols) for the grounding query with preemptive shattering (squares, "ground"), the grounding query with on-demand shattering (circles, "on-

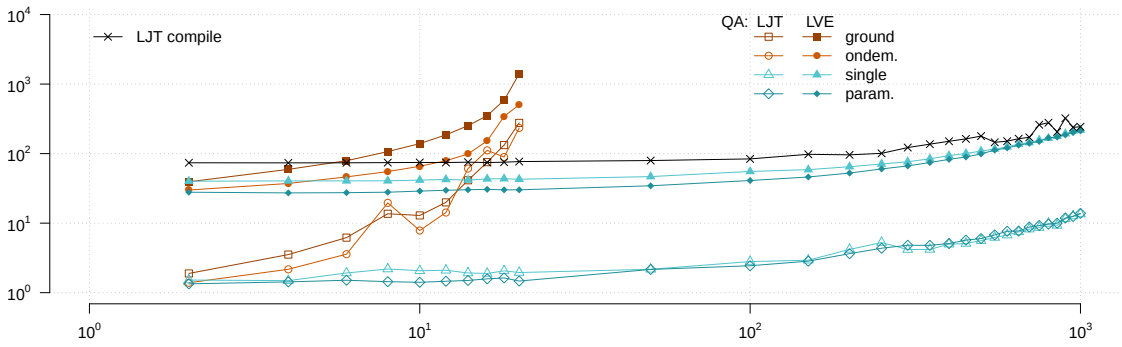


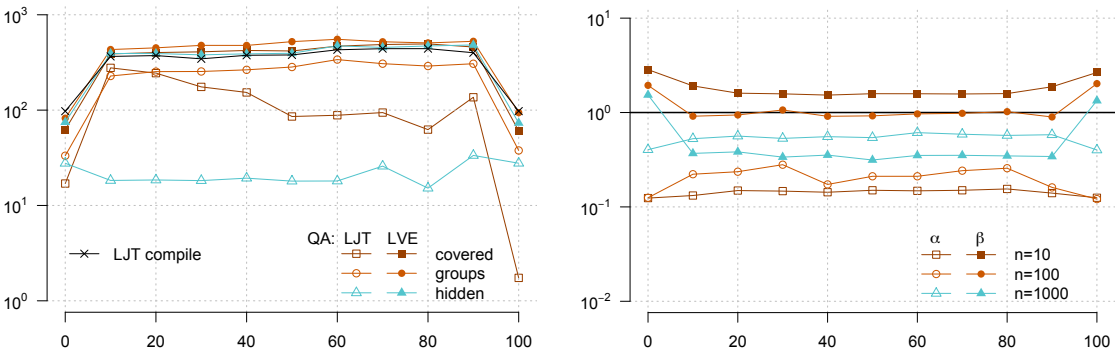
Figure 9.3: Runtimes [ms] for query answering for LJT and LVE as well as compile time for LJT w.r.t. domain size n ranging from 2 to 1000; subtree size $n'_J = 1$

dem."), the singleton query (triangles, "single"), and the parameterised query (diamonds, "param."). The x-axis plots the increasing domain size for model logvars, ranging from 2 to 1000, on a log scale. As in Fig. 8.3, the grounding queries lead LVE and LJT to have a sharp increase in runtime with rising domain sizes, while the singleton query exhibits a polynomial increase in runtimes. The parameterised query displays a similar behaviour, with runtimes being very close to the runtimes of the singleton query.

As expected, LJT runtimes are lower than LVE runtimes with compile time of LJT (black crosses) being slightly longer than for answering a single query with LVE. Runtimes for the parameterised query are at times lower than runtimes for the singleton query, even though the result of the parameterised query is larger. The reason lies in the parameterised query having a $\top$ constraint, which means that LVE does not need to split its input model on the query term. With a constraint that restricts a logvar to a subset of constants, the parameterised query requires a split as well. Overall, a parameterised singleton query allows for runtimes similar to ground singleton query.

Evidence As previously discussed, evidence can have an influence on the result representation. There are three possible cases: (i) Instances of evidence PRVs appear in query terms. (ii) Constants of evidence PRVs appear in query terms, referring to different PRVs. (iii) Neither instances nor constants of evidence PRVs appear in query terms. Based on these cases, we set up an evaluation. We consider evidence for a one-logvar PRV $R(X)$ that appears in one parcluster together with another PRV $S(X)$ with the same logvar X, varying the coverage of evidence from 0% to 100% for the instances of $R(X)$. We then pose three queries, one for $R(X)$, one for $S(X)$, and one for some PRV $Q(Y)$ in another parcluster without X in its logvars, referred to as the "covered" query, the "groups" query, and the "hidden" query respectively.

Figure 9.4a depicts runtimes in milliseconds [ms] on a log scale for QA with LJT (hollow symbols) and LVE (filled symbols), both answering the "covered" query (squares),



(a) Runtimes [ms] for query answering for LJT and LVE plus compile time for LJT; $n = 1000$

(b) α and β values for LJT compared to LVE using "groups" runtimes (worst query)

Figure 9.4: Interplay of parameterised queries and evidence with evidence coverage ranging from 0% to 100% for a 1-logvar PRV

the "groups" query (triangles), and the "hidden" query (diamonds) for a domain size of $n = 1000$. The x-axis shows the increasing evidence coverage. LVE runtimes for all three queries mirror each other over the different evidence sets. After introducing evidence, runtimes increase as evidence needs to be handled. With more evidence, runtimes go down again and reach a low with 100% evidence coverage as evidence completely eliminates one PRV. The runtimes are within the same order of magnitude. The "groups" query leads to largest runtimes, while in most cases the "hidden" query leads to the shortest runtimes. For the corner cases, the "covered" query leads to slightly faster runtimes.

For all three queries, LJT exhibits shorter runtimes than LVE as expected. The "covered" query and the "groups" query also show an increase and decrease in runtime as LVE. The LJT runtimes for the "groups" query mirror the LVE behaviour the closest. For the "covered" query with 100% evidence, LJT exhibits a sharp decrease in runtime as the submodel becomes very small after evidence absorption. The "hidden" query leads to slightly different runtime behaviour of LJT. The reason lies in evidence not only being hidden from the query but also the used parcluster, i.e., the local model and messages do not change structurally given different evidence coverage; only concrete potentials change. Thus, evidence coverage does not particularly influence runtimes for this query.

Figure 9.4b depicts α and β values for LJT compared to LVE for domain sizes $n \in \{10, 100, 1000\}$ using the runtimes of the "groups" query, which has higher α values than the other two queries. For all three domain sizes, α values are below 1, meaning LJT is able to trade off its static overhead over multiple queries. β values show that LJT needs one to three queries to actually trade off its overhead. With $n = 1000$, LJT achieves its tradeoff the fastest, where compiling a helper structure pays off with the first query.

In general, LJT and LVE for parameterised queries allow for efficient answering of queries with interchangeable query terms, with LJT being able to leverage its cluster

representation for faster query answering. Yet, evidence is easily manageable by both algorithms in combination with parameterised queries. LJT even benefits from its helper structure for certain queries where evidence is hidden from a parcluster and thus does not lead to larger runtimes.

9.5 Interim Conclusion: Parameterised Queries

In this chapter, we apply lifting to queries, parameterising query terms as we have parameterised randvars and consequently, factors. Parameterised queries allow for a compact representation of queries with interchangeable query terms by introducing logvars to queries. LVE for parameterised queries provides a lifted run if possible. It shatters a model on a parameterised query, avoiding the initial grounding w.r.t. constants appearing in a query. Then, it eliminates non-query terms, which allows for a form of on-demand shattering if groundings are unavoidable. After eliminating all non-query terms, LVE induces a joint distribution. Doing so, LVE uses count conversions to provide a compact result representation through CRVs. Normalisation incorporates CRVs in a result.

As mentioned during the evaluation, parameterised queries do not occur in this form in the implementation of FOKC. An interesting path regarding future work would be to look at other lifted inference algorithms to see whether one can implement parameterised queries in them. Additionally, one can further investigate the connection between parameterised queries and forbidden queries as well as the query classes Van den Broeck (2013) investigates w.r.t. completeness of FOKC.

In conclusion, we have introduced parameterised queries, which provide an efficient means to answering queries about distributions, e.g., over different fractions of people being sick given an epidemic. This chapter holds Contribution (4b), lifted QA by lifting of queries, and part of Contribution (4c), combined completeness and complexity results for complex queries. In our quest for an extended query language, we now leave queries for probability distributions behind and turn to queries for most probable assignments. Speaking in the terms of our running example, we are interested in the most likely subset of sick people and their treatments.

Chapter 10

Most Probable Assignments

So far, queries concern marginal and conditional probability distributions over sets of randvars, parameterised or instantiated. We call these types of queries probability queries. Another important inference task is abduction, in graphical models often formalised as computing the most probable assignment to some randvars in a model without evidence. Asking for the most probable assignment to *all* randvars without evidence is known as an MPE query (total abduction). Within the VE formalisation, switching from a probability query to an MPE query requires replacing the sum-out operations with arg max operations. The generalisation is a maximum a posteriori, MAP, query for the most probable assignment to a *subset* of model randvars (partial abduction). An MAP query requires summing out those randvars not in the query before maxing out the randvars in the query. With sum-out and max-out operations not being commutative, MAP queries can impose elimination orders that fast lead to prohibitively large intermediate results (Dechter, 1999). We call MPE and MAP queries assignment queries.[1]

Pearl (1988) introduces the idea of MPEs and a propagation algorithm for singly-connected networks. Dawid (1992) presents an algorithm to compute an MPE on jtrees. Dechter (1999) formalises computing MPEs as a form of VE, replacing sum-out with max-out operations to eliminate randvars. While these methods are based on the idea of VE, other methods for solving an MPE problem have been developed based on different formalisms. Park (2002) reduces the MPE task to the weighted MAX-SAT problem. Analogously to VE, Darwiche (2009) replaces sum operations with max operations in KC to compute an MPE query with a circuit. Kimmig *et al.* (2017) extend WMC to algebraic model counting and use KC to solve an algebraic model counting problem for an MPE query. ProbLog (De Raedt *et al.*, 2007), initially focused on answering probability queries by reducing the task to WMC, handles MPE tasks by reducing the problem to weighted MAX-SAT (Fierens *et al.*, 2015). Kimmig *et al.* (2011) extend ProbLog, introducing algebraic ProbLog for semi-rings, and define how to compute an MPE with it. Shterionov *et al.* (2015) extend ProbLog with a new encoding for solving the MPE task. Bach *et al.* (2017) present probabilistic soft logic to deal with continuous domains and MPE. A related problem is finding a most probable database or most probable hypotheses given a PDB. Ceylan *et al.* (2017) show the complexity of computing most

[1] In the literature, MPE is also known as MAP and MAP is known as partial/marginal MAP.

probable databases and hypotheses, jumping off the work by Gribkoff *et al.* (2014) on most probable databases. Lüdtke *et al.* (2018) study most probable state sequences based on sampling using the lifting idea for a sparse encoding.

While most of these frameworks allow for modelling relations, objects, and uncertainty, they do not focus on lifted inference. For solving MPE problems, the first lifted solution comes from de Salvo Braz *et al.* (2006), who adapt an earlier version of LVE, which does not lift all calculations that are possible to lift with generalised counting. For solving an MPE query, Apsel and Brafman (2012a) simplify a model by eliminating logvars under a uniform assignment condition, which we could incorporate as a preprocessing step into the upcoming inference algorithm. Sarkhel *et al.* (2014) define the MPE task for Markov logic networks that only contain formulas with atoms that do not share logvars, highly restricting the type of models one can express. Sharma *et al.* (2018) focus on MAP queries, introducing a new lifting rule, which uses specific occurrences of logvars to simplify a model. Apsel *et al.* (2012) and Mladenov *et al.* (2014) use lifting for linear programs to find an approximate solution for an MPE task. None of the above concentrate on providing a complete algorithm for solving the MPE problem exactly for probabilistic relational models in the light of the latest advances in LVE, which means that so far there is no lifted algorithm that is complete for the same models as LVE.

Therefore, we use the LVE operator suite by Taghipour *et al.* (2013c) to define an MPE-LVE algorithm for solving MPE problems, transferring the ideas of Dawid and Dechter to the lifting scenario and thus, fully leveraging relational aspects of a model. We first presented the LVE operators redefined for MPE in the following paper:

> Tanya Braun and Ralf Möller. Lifted Most Probable Explanation. In *Proceedings of the International Conference on Conceptual Structures*, pages 39–54. Springer, 2018

This chapter also includes the generalised counting operators (Taghipour and Davis, 2012) redefined for MPE as well as algorithms combining LVE, LJT, and MPE-LVE for answering MAP queries. To the best of our knowledge, the chapter provides the first thorough theoretical analysis of MPE and MAP in the context of lifted inference.

In this last chapter of Part II, we present lifted algorithms for solving MPE and MAP problems based on the LVE operator suite, covering Contributions (5a) and (5b). We first present MPE versions of LVE and LJT, including redefined LVE operators. Then, we consider MAP queries, which allow LJT to leverage its strength regarding multiple queries. Last, we introduce an algorithm that answers a set of queries of varying type, fusing all algorithms that have occurred up to this point into one large algorithm. Next, we dive into a theoretical analysis, which is Contribution (5c), for all algorithms presented in this chapter. We end this chapter with an empirical evaluation.

10.1 Lifted Algorithms for Most Probable Explanations

We adapt LVE and LJT for MPE queries by rewriting operators for MPE-LVE and by introducing MPE-LVE into LJT and adapting message passing for MPE-LJT. But first, we define preliminaries including MPE and MAP problems and extend the definition of a parfactor for storing assignments next to potentials.

Preliminaries We define MPE and MAP problems as follows and extend the definition of parfactors as a consequence of dealing with assignment queries.

Definition 10.1.1 (MPE/MAP query/problem)**.** Let G be a model with a full joint distribution P_G, $\{E_j = e_j\}_{j=1}^m$ a set of events (evidence), where $E_j \in rv(G)$ are grounded PRVs or propositional randvars, and $\mathbf{V}_{|C} = rv(G) \setminus \{E_j\}_{j=1}^m$ the set of randvars without evidence in G. We denote a set of events by $\mathbf{A} = \mathbf{a}$. An *MPE problem* refers to the problem of finding an assignment for $\mathbf{V}$ with the highest probability w.r.t. P_G, i.e.,

$$MPE_G = \arg\max_{\mathbf{v}} P((\mathbf{V} = \mathbf{v})_{|C} | \mathbf{E} = \mathbf{e}). \tag{10.1}$$

Given 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 $\mathbf{T}_{|C''} = \mathbf{V}_{|C} \setminus \mathbf{U}_{|C'}$:

$$MAP_G = \arg\max_{\mathbf{u}} P((\mathbf{U} = \mathbf{u})_{|C'} | \mathbf{E} = \mathbf{e}) = \arg\max_{\mathbf{u}} \sum_{\mathbf{t} \in \mathbf{T}_{|C''}} P((\mathbf{U} = \mathbf{u})_{|C'}, (\mathbf{T} = \mathbf{t})_{|C''} | \mathbf{E} = \mathbf{e}) \tag{10.2}$$

We refer to an *MPE query* by $\mathrm{MPE}(\mathbf{E} = \mathbf{e})$ and to an *MAP query* by $\mathrm{MAP}(\mathbf{U}_{|C'} | \mathbf{E} = \mathbf{e})$. If $\mathbf{U}_{|C'} = \mathbf{V}_{|C''}$, i.e., $\mathbf{T} = \emptyset$, then $\mathrm{MAP}(\mathbf{U}_{|C'} | \mathbf{E} = \mathbf{e}) = \mathrm{MPE}(\mathbf{E} = \mathbf{e})$.

Example 10.1.1 (MPE and MAP queries)**.** Consider G_{ex} and evidence $sick(eve)$. An MPE query $\mathrm{MPE}(sick(eve))$ asks for the most probable assignments to all randvars in $rv(G_{ex}) \setminus \{Sick(eve)\}$. Solving the MPE problem behind the MPE query requires computing assignments for $rv(G_{ex}) \setminus \{Sick(eve)\}$ that lead to maximum potentials in $P_{G_{ex}}$ given $sick(eve)$. An MAP query $\mathrm{MAP}(Sick(X)_{|\top} | sick(eve))$ asks for the most probable assignment to $\{Sick(X)\}_{|\top} \setminus \{Sick(eve)\}$, which means $\{Sick(X)\}_{|C}$, $C = ((X), \{(alice), (bob)\})$. Solving the MAP problem behind the MAP query requires eliminating $rv(G_{ex}) \setminus \{Sick(eve)\} \setminus \{Sick(X)\}_{|\top}$, solving $\mathrm{MPE}(Sick(X)_{|C})$ in the result.

To define the new operators in MPE-LVE, we need formal definitions of a count function and of count normalisation.

Definition 10.1.2 (Count, count-normalised)**.** Given a constraint $C = (\mathbf{X}, C_{\mathbf{X}})$, for any $\mathbf{Y} \subseteq \mathbf{X}$ and $\mathbf{Z} \subseteq \mathbf{X} \setminus \mathbf{Y}$, the function $\mathrm{COUNT}_{\mathbf{Y}|\mathbf{Z}} : C_{\mathbf{X}} \to \mathbb{N}$ is defined by

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

We define $\text{COUNT}_{\mathbf{Y}|\mathbf{Z}}(t) = 1$ for $\mathbf{Y} = \emptyset$. $\mathbf{Y}$ is called *count-normalised* w.r.t. $\mathbf{Z}$ in C iff $\exists n \in \mathbb{N} : \forall t \in C_{\mathbf{X}} : \text{COUNT}_{\mathbf{Y}|\mathbf{Z}}(t) = n$. We call such an n the conditional count of $\mathbf{Y}$ given $\mathbf{Z}$ in C, denoted by $\text{NCOUNT}_{\mathbf{Y}|\mathbf{Z}}(C)$.

Example 10.1.2. Consider constraint $C = ((X, M), \{(eve, tablet), (alice, injection), (alice, tablet), (bob, injection), (bob, tablet)\})$ from Example 3.2.3, which is the result from shattering parfactor g_3 on the query term $Treat(eve, injection)$. With $\mathbf{X} = \{X, M\}$, $\mathbf{Y} = \{M\}$, and $\mathbf{Z} = \{X\}$, the count function calculates the following for tuple $(eve, tablet)$: First, it projects $(eve, tablet)$ onto $\{X\}$, which leaves (eve). Then, it joins the tuples from the second component of C with $\{(eve)\}$ over $\{X\}$, yielding $\{(eve, tablet)\}$, and projects the tuples onto $\{M\}$, which results in a set with one element, namely $(tablet)$. Last, it returns the cardinality of the set, here 1. For $(alice, injection)$, the inner projection yields $\{(alice)\}$, the join results in $\{(alice, injection), (alice, tablet)\}$, and the outer projection leads to $\{(injection), (tablet)\}$, yielding a cardinality of 2. The same holds for the remaining tuples in C. Thus, there does not exist a unique n for all tuples in C, that is, M is not count-normalised w.r.t. X in this case.

After shattering G_{ex} on evidence $sick(eve)$ as in Example 3.2.4, parfactor g_3^r has a constraint $C' = ((X, M), \{(alice, injection), (alice, tablet), (bob, injection), (bob, tablet)\})$. Here, each tuple leads to a count of 2 given $\mathbf{X} = \{X, M\}$, $\mathbf{Y} = \{M\}$, and $\mathbf{Z} = \{X\}$ and thus, M is count-normalised w.r.t. X in C'. The conditional count of M given X in C' is 2 and has been used in Example 3.2.5 where LVE eliminates $Treat(X, M)$.

The main difference between LVE and MPE-LVE is the change from lifted summing out to lifted maxing out. Lifted maxing out picks a maximum potential for a representative and then stores the argument values that lead to this potential for all interchangeable randvars. Later this section, we consider preconditions to ensure that randvars are interchangeable for this operation.

In the propositional case, an input valuation leads to one particular assignment. The same holds in the lifting case if all instances of a logvar lead to one particular, i.e., uniform, assignment. Then, a factor simply has to store this uniform assignment value. But, with CRVs and count conversions, an input valuation may include a histogram, which may stand for different input valuations, e.g., $(true, [2, 1])$ stands for two times $(true, true)$ and one time $(true, false)$, which can map to different arg max assignments, e.g., $true$ and $false$. Thus, an input valuation may lead to multiple assignments. As the valuations behind $[2, 1]$ appear simultaneously, the arg max assignments also appear simultaneously, which one can record in another histogram, $[2, 1]$, storing the assignment information in a compact way. Such an encoding also has the advantage that we do not need to store individual mapping rules for various mappings behind a histogram.

When computing an MPE solution, the algorithm has to store not only assignments but has to continue to store potentials, even though the ultimate output is a string of assignments. To this end, we extend the notion of a parfactor to map arguments to potentials as well as range values for maxed out PRVs, which are stored in histograms.

Definition 10.1.3 (Parfactor)**.** Assume a sequence of maxed out PRVs $\mathcal{B} = (B_1, \ldots, B_l)$ with $lv(\mathcal{B}) = \mathbf{Y}$, in which each PRV $B \in \mathcal{B}$ has a range of possible histograms. Let $\mathcal{A} = (A_1, \ldots, A_k)$ be a sequence of P(C)RVs with $lv(\mathcal{A}) \subseteq \mathbf{X}$ and $C = (\mathcal{X} \circ \mathcal{Y}, C_{\mathcal{X} \circ \mathcal{Y}})$ be a constraint on $\mathbf{X}$ and $\mathbf{Y}$. Let $\pi_{\mathbf{X}}(C)$ denote the projection of the second component of C onto $\mathbf{X}$. A *parfactor* g is given by $\forall \mathbf{x} \in \pi_{\mathbf{X}}(C) : \phi(\mathcal{A})^{\mathcal{B}}_{|C}$, where ϕ is a function

$$\phi : \times_{i=1}^{k} \mathcal{R}(A_i) \mapsto (\mathbb{R}^+, \times_{j=1}^{l} \mathcal{R}(B_j))$$

which maps a particular input $\mathbf{a} \in \times_{i=1}^{k} \mathcal{R}(A_i)$ to a potential $p \in \mathbb{R}^+$ and a sequence of histograms $(h_1, \ldots, h_l)$, where $h_j \in \mathcal{R}(B_j)$ is a histogram. We refer to the potential by ϕ^P (potential) and to the sequence by ϕ^A (assignment). ϕ^A_B refers to the assignment of B in ϕ^A. For short, we write $\phi(\mathcal{A})^{\mathcal{B}}_{|C}$. We omit $|C$ if $C = \top$ and $\mathcal{B}$ if $\mathcal{B} = ()$.

The sequence of maxed out PRVs does not influence the terms $rv(g)$, $lv(g)$, and $gr(g)$. They are still based on the logvars $\mathbf{X}$, arguments $\mathcal{A}$, and constraint C of g. The histograms of maxed out PRVs depend on the logvars maxed out along with PRVs as well as count conversions applied. A histogram explicitly encodes how many instances a maxed out PRV stands for in relation to the remaining logvars. Let us look at the following example to further clarify the new parfactor definition and the use of histograms.

Example 10.1.3. Consider the PRVs $Epid$, $Sick(X)$, and $Treat(X, M)$ and constraint $C' = ((X, M), \{(alice, injection), (alice, tablet), (bob, injection), (bob, tablet)\})$ as given in the previous example. For a parfactor $\phi(Epid, Sick(X))^{\mathcal{B}}_{|C'}$ with $\mathcal{B} = (Treat(X, M))$, Table 10.1a depicts possible input-output pairs of ϕ. Each valuation of $Epid$ and $Sick(X)$ maps to a tuple (ϕ^P, ϕ^A) of a potential ϕ^P, here numbers between 1 and 16, as well as a sequence of histograms ϕ^A. In the histograms, the first position refers to the number of instances of $Treat(X, M)$ with the assignment $true$, the second to the number of instances with the assignment $false$. The maxed out PRV $Treat(X, M)$ has an additional logvar, M, compared to arguments $Epid$ and $Sick(X)$. In C', there exist two constants of M, namely, $injection$ and $tablet$, for each constant of X, *alice* and *bob*. Therefore, the counts in the histograms add up to 2. E.g., the valuation $(false, false)$ maps to a potential of 4 and a histogram $[2, 0]$, which means two instances of $Treat(X, M)$ have the value $true$ in relation to each X constant in C'. Even though the histograms in the example are peak-shaped, i.e., have only one range value with a non-zero count, through count conversions, other histogram shapes occur. At the end of an assignment calculation, each logvar will be accounted for and thereby, the histograms explicitly encode how many instances of a PRV have a specific value, e.g., as seen in Table 10.1b. At the beginning of an assignment calculation, the set of histograms is empty, e.g., as shown in Table 10.1c.

Making assignments part of the output of parfactors allows for automatically dropping assignments along with potentials that are not maximum. Additionally, histograms allow for directly reading off how many instances take which value at the end. Next, we define the new *max-out* operator to implement the maxing out operation in a given parfactor.

Table 10.1: Parfactors and maxed out PRVs

(a) Parfactor with a maxed-out PRV

$Epid$	$Sick(X)$	$\phi(Epid, Sick(X))^{(Treat(X,M))}$
$false$	$false$	$(4, ([0, 2]))$
$false$	$true$	$(9, ([2, 0]))$
$true$	$false$	$(1, ([2, 0]))$
$true$	$true$	$(16, ([2, 0]))$

(b) Parfactor without arguments left

()	$\phi()^{(Epid,Sick(X),Treat(X,M))}$
.	$(256, ([1, 0], [2, 0], [4, 0]))$

(c) Parfactor without any maxed-out PRVs

$Epid$	$Sick(X)$	$Treat(X, M)$	$\phi(Epid, Sick(X), Treat(X, M))$
$false$	$false$	$false$	$(2, ())$
$false$	$false$	$true$	$(1, ())$
$false$	$true$	$false$	$(2, ())$
$false$	$true$	$true$	$(3, ())$
$true$	$false$	$false$	$(0, ())$
$true$	$false$	$true$	$(1, ())$
$true$	$true$	$false$	$(2, ())$
$true$	$true$	$true$	$(4, ())$

Elimination by Maxing Out Operator 1 defines lifted maxing out, which takes the role of eliminating PRVs in MPE-LVE. The symbol $\circ$ denotes concatenating sequences. The inputs are a parfactor $g = \phi(\mathcal{A})^{\mathcal{B}}_{|C}$ and a PRV A_i to eliminate from g. The preconditions of *max-out* are identical to *sum-out*, ensuring that (i) all occurrences of A_i are maxed out, (ii) each instance of A_i occurs in exactly one separate ground factor, and (iii) there is a unique exponent for exponentiation. Together, the conditions ensure that *max-out* eliminates a set of instances where the ground calculations would be copies of each other.

The output of Operator 1 is a parfactor $\phi'(\mathcal{A}')^{\mathcal{B}'}_{|C}$. The new argument sequence $\mathcal{A}'$ is the old sequence $\mathcal{A}$ minus A_i. The sequence of maxed out PRVs $\mathcal{B}$ is extended by A_i. Constraint C is unchanged, which is different compared to lifted summing out where the operator projects the constraint onto the remaining logvars. The projection is reasonable for lifted summing out as the instances of an eliminated logvar no longer hold relevant information. But, since we are interested in assignments, the instances play a role for the final result. Thus, we keep the constraint as is to keep track of all instances covered by an assignment. The main part of Operator 1 is dedicated to determining the new mappings for ϕ'. For a valuation $\mathbf{a}' = (\dots, a_{i-1}, a_{i+1}, \dots)$ of $\mathcal{A}'$, ϕ' maps $\mathbf{a}'$ to a tuple of

Operator 1 Lifted Maxing-out

Operator MAX-OUT

Inputs:

(1) $g = \forall \mathbf{x} \in \pi_{\mathbf{X}}(C) : \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$

(2) $A_i \in \mathcal{A}$, to be maxed out from g

Preconditions:

(1) For all $g' \in G, g' \neq g : rv(g') \cap \{A_{i|C}\} = \emptyset$.

(2) A_i contains all the logvars $X \in lv(\mathcal{A})$ for which $\pi_X(C)$ is not singleton.

(3) $\mathbf{X}^{excl} = lv(A_i) \setminus lv(\mathcal{A} \backslash A_i)$ count-normalised w.r.t. $\mathbf{X}^{com} = lv(A_i) \cap lv(\mathcal{A} \backslash A_i)$ in $\pi_{\mathbf{X}}(C)$.

Output: $\forall \mathbf{x} \in \pi_{\mathbf{X} \backslash \mathbf{X}^{excl}}(C) : \phi'(\mathcal{A}')^{\mathcal{B}'}_{|C}$ such that

(1) $\mathcal{A}' = (A_1, \ldots, A_{i-1}) \circ (A_{i+1}, \ldots, A_n)$,

(2) $\mathcal{B}' = (A_i) \circ \mathcal{B}$, and

(3) for each valuation $\mathbf{a}' = (\ldots, a_{i-1}, a_{i+1}, \ldots)$ of $\mathcal{A}'$,

$$\phi'(\mathbf{a}') = \left(p^r_{max}, (r \cdot h_{A_i}) \circ \bigcirc_{h_B \in \phi^A(\ldots, a_{i-1}, a_{max}, a_{i+1}, \ldots)} r_B \cdot h_B \right),$$

- $p_{max} = \max_{a_i \in \mathcal{R}(A_i)} \phi^P(\ldots, a_{i-1}, a_i, a_{i+1}, \ldots)$,
- $a_{max} = \arg\max_{a_i \in \mathcal{R}(A_i)} \phi^P(\ldots, a_{i-1}, a_i, a_{i+1}, \ldots)$,
- $r = \text{NCOUNT}_{\mathbf{X}^{excl} | \mathbf{X}^{com}}(\pi_{\mathbf{X}}(C))$,
- $h_{A_i} = \begin{cases} a_{max} & \text{if } A_i \text{ (P)CRV,} \\ \{(a_i, n_i)\}_{i=1}^{|\mathcal{R}(A_i)|}, n_i = \begin{cases} 1 & \text{if } a_i = a_{max}, \\ 0 & \text{otherwise.} \end{cases} & \text{otherwise.} \end{cases}$
- $r_B = \text{NCOUNT}_{\mathbf{X}^{excl} \cap lv(B) | \mathbf{X}^{com}}(\pi_{\mathbf{X}}(C))$.

Postcondition: $MPE_{G \setminus \{g\} \cup \{\text{MAX-OUT}(g, A_i)\}} = \arg\max_{A_{i|C} = a_i} P_G$

a potential and an extended and adapted sequence of assignments, i.e.,

$$\left(p^r_{max}, (r \cdot h_{A_i}) \circ \bigcirc_{h_B \in \phi^A(\ldots, a_{i-1}, a_{max}, a_{i+1}, \ldots)} r_B \cdot h_B \right).$$

The new potential p_{max} is the maximum potential among the potentials to which $\mathbf{a}'$ together with an $a_i \in \mathcal{R}(A_i)$ maps. The exponent r denotes the conditional count of $\mathbf{X}^{excl}$ given $\mathbf{X}^{com}$ in C and accounts for multiple logvar instances eliminated w.r.t. the instances represented by the remaining logvars. r is greater than 1 if maxing out eliminates a logvar along with a PRV, i.e., $|\mathbf{X}^{excl}| > 0$. In the histogram h_{A_i}, the operator stores the arg max assignment a_{max} that leads to p_{max}. If A_i is a PCRV, a_{max} already is a histogram. Otherwise, Operator 1 forms a new histogram with tuples $(a_{max}, 1)$ and

$(a_i, 0)$ for $a_i \in \mathcal{R}(A_i), a_i \neq a_{max}$. The histogram h_{A_i} is multiplied by r. If maxing out A_i does not eliminate a logvar from g, i.e., $|\mathbf{X}^{excl}| = 0$ and $r = 1$, the histogram remains the same. If maxing out A_i eliminates logvars from g, the operator materialises the number of instances eliminated w.r.t. the instances represented by the remaining logvars, i.e., r, by multiplying h_{A_i} with r. In the same vain, if maxing out A_i eliminates a logvar that appears in a maxed out PRV B with histograms h_B, the operator needs to multiply h_B with a conditional count of $\mathbf{X}^{excl} \cap lv(B)$ given $\mathbf{X}^{com}$ in C. Adapting the count conversion operator to assignments demonstrates the necessity of histograms as well as the multiplication with r to keep the histograms consistent.

The postcondition in Operator 1 reflects the max-out operation. Before moving on to the other operators, let us have a look at an example, maxing out $Treat(X, M)$ in G_{ex}, analogous to Example 3.2.1, which illustrates LVE summing out $Treat(X, M)$ in G_{ex}.

Example 10.1.4 (Lifted maxing out)**.** Only $g_3 = \phi_3(Epid, Sick(X), Treat(X, M))$ contains $Treat(X, M)$, which fulfils the first precondition. $Treat(X, M)$ also contains all logvars in g_3, with $\mathbf{X}^{excl} = \{M\}$ and $\mathbf{X}^{com} = \{X\}$. The constraint in g_3 is $\top$, i.e., each combination of constants from $\mathcal{D}(X)$ and $\mathcal{D}(M)$ exist. For each constant of X, there appear $|\mathcal{D}(M)| = 2$ constants of M, meaning M is count-normalised w.r.t. X with a count of $r = 2$. Thus, all three preconditions are fulfilled. To form the output, MPE-LVE needs a concrete mapping for g_3. Assume a function specification as given in Table 10.1c. The output parfactor ϕ' has the arguments $Epid$ and $Sick(X)$ and a $\top$ constraint. For each remaining valuation of $Epid$ and $Sick(X)$, MPE-LVE needs to store a tuple of a max potential and the corresponding arg max assignment for $Treat(X, M)$. Consider the valuation $(false, false)$. MPE-LVE iterates over the range values of $Treat(X, M)$, i.e., $false$ and $true$, and picks the larger potential mapped to by $(false, false)$ together with $(false)$ and $(false, false)$ together with $(true)$. Looking into Table 10.1c, the valuation $(false, false, false)$ leads to a potential of 2 and the valuation $(false, false, true)$ leads to a potential of 1. With $(false, false, false)$ yielding the larger potential, $p_{max} = 2$ and $a_{max} = false$. MPE-LVE builds the new tuple by setting p^r_{max} as the first component ϕ^P. For the second component ϕ^A, MPE-LVE has to build a histogram h for a_{max}. With $Treat(X, M)$ being boolean, h has two tuples $(true, n_1)$ and $(false, n_2)$ where $n_1 = 0$ and $n_2 = 1$ as $a_{max} = false$. To incorporate that maxing out $Treat(X, M)$ eliminates M, MPE-LVE multiplies h with r, leading to a histogram $[0, 2]$ in shorthand notation. As $\phi^A(false, false, false)$ is empty, no further histograms need to adapt to M being eliminated. MPE-LVE repeats these steps for the other valuations $(false, true)$, $(true, false)$, and $(true, true)$, resulting in a parfactor as depicted in Table 10.1a.

Next, we adapt the other LVE operators to incorporate the reworked parfactors and treat assignments appropriately.

LVE Operators for MPE The remaining LVE operators are *absorb*, *multiply*, *count-convert*, *split*, *expand*, *count-normalise*, and *ground-logvar*. We need to ensure that each

operator incorporates the stored assignments ϕ^A if necessary. The operator *absorb* handles lifted absorption of evidence, which only happens at the beginning before any changes occur due to maxing out instead of summing out. Thus, $\mathcal{B} = ()$ and $\phi^A = ()$ throughout the application of *absorb*. The other LVE operators transform a part of a model to enable lifted eliminations. Most of the changes occur in *multiply* and *count-convert*.

The operator *multiply* implements multiplying two parfactors, which may establish the first precondition of *max-out*. The LVE operator assumes that the logvars in both parfactors do not share names and that an alignment then renames logvars in one of the parfactors to ensure that only those PRVs and logvars map that should map when merging both constraints during multiplication. The definition of an alignment, which is based on a substitution, follows.

Definition 10.1.4 (Alignment)**.** An *alignment* θ of parfactors $\phi_1(\mathcal{A}_1)_{|C_1}$ and $\phi_2(\mathcal{A}_2)_{|C_2}$ is a one-to-one substitution $\{\mathbf{X}_1 \to \mathbf{X}_2\}$ with $\mathbf{X}_1 \subseteq lv(\mathcal{A}_1)$ and $\mathbf{X}_2 \subseteq lv(\mathcal{A}_2)$ s.t.

$$(\pi_{\mathbf{X}_1}(C_1))\theta = \pi_{\mathbf{X}_2}(C_2).$$

Operator 2 shows how to multiply two parfactors. The inputs are two parfactors $\phi_1(\mathcal{A}_1)^{\mathcal{B}_1}_{|C_1}$ and $\phi_2(\mathcal{A}_2)^{\mathcal{B}_2}_{|C_2}$ to multiply and an alignment θ that aligns a subset of logvars $\mathbf{Z}_1$ from $lv(\mathcal{A}_1)$ with a subset of logvars $\mathbf{Z}_2$ from $lv(\mathcal{A}_2)$.

Operator 2 Lifted Multiplication

Operator MULTIPLY
Inputs:
(1) $g_1 = \forall \mathbf{x}_1 \in \pi_{\mathbf{X}_1}(C_1) : \phi_1(\mathcal{A}_1)^{\mathcal{B}_1}_{|C_1}, g_1 \in G$
(2) $g_2 = \forall \mathbf{x}_2 \in \pi_{\mathbf{X}_2}(C_2) : \phi_2(\mathcal{A}_2)^{\mathcal{B}_2}_{|C_2}, g_2 \in G$
(3) $\theta = \{\mathbf{Z}_1 \to \mathbf{Z}_2\}$, an alignment between g_1 and g_2
Preconditions:
(1) for $i = 1, 2 : \mathbf{Y}_i = lv(\mathcal{A}_i) \setminus \mathbf{Z}_i$ is count-normalised w.r.t. $\mathbf{Z}_i$ in $\pi_{\mathbf{X}_i}(C_i)$
(2) $rv(\mathcal{B}_2) \cap rv(\mathcal{B}_2) = \emptyset$
Output: $\forall \mathbf{x} \in \pi_{\mathbf{X}_1\theta \cup \mathbf{X}_2}(C) : \phi(\mathcal{A})^{\mathcal{B}}_{|C}$ such that
(1) $\mathcal{A} = \sigma_{rv(\mathcal{A}_1\theta)\setminus rv(\mathcal{A}_2)}(\mathcal{A}_1\theta) \circ \mathcal{A}_2$,
(2) $\mathcal{B} = \mathcal{B}_1\theta \circ \mathcal{B}_2$,
(3) $C = C_1\theta \bowtie_{\mathbf{Z}_2} C_2$, and
(4) for each valuation $\mathbf{a}$ of $\mathcal{A}$, with $\mathbf{a}_1 = \pi_{\mathcal{A}_1\theta}(\mathbf{a})$ and $\mathbf{a}_2 = \pi_{\mathcal{A}_2}(\mathbf{a})$,

$$\phi(\mathbf{a}) = \left(\phi_1^P(\mathbf{a}_1)^{\frac{1}{r_2}} \cdot \phi_2^P(\mathbf{a}_2)^{\frac{1}{r_1}}, \phi_1^A(\mathbf{a}_1) \circ \phi_2^A(\mathbf{a}_2)\right)$$

where $r_i = \text{NCOUNT}_{\mathbf{Y}_i|\mathbf{Z}_i}(\pi_{\mathbf{X}_i}(C_i))$.
Postcondition: $G \equiv G \setminus \{g_1, g_2\} \cup \{\text{MULTIPLY}(g_1, g_2, \theta)\}$

The operator has two preconditions: (i) The logvars outside the alignment in each parfactor are count-normalised w.r.t. the logvars in the alignment. (ii) The maxed out PRVs do not intersect. The first condition also appears in lifted multiplication of LVE. It ensures a unique conditional count for the exponents in the output. The second condition is specific to MPE-LVE, ensuring that the parfactors do not share maxed out PRVs, which is true for an algorithm run starting with empty assignment sequences. A maxed out PRV occurs after applying the *max-out* operator to a PRV A. One of its preconditions ensures that it can only max out A if A occurs only in a given parfactor. Thus, maxed out PRVs from different parfactors are distinct. Otherwise, the parfactors would have been multiplied before maxing out A.

The output of Operator 2 is a parfactor $\phi(\mathcal{A})^{\mathcal{B}}_{|C}$. The new argument sequence $\mathcal{A}$ merges the old sequences $\mathcal{A}_1$ and $\mathcal{A}_2$, keeping only one version of shared PRVs. The new sequence of maxed out PRVs $\mathcal{B}$ is a concatenation of $\mathcal{B}_1\theta$ and $\mathcal{B}_2$, which is enabled by the second precondition. The new constraint is a join of $C_1\theta$ and C_2 with the common logvars $\mathbf{Z}_2$. For a valuation $\mathbf{a}$ of $\mathcal{A}$, ϕ maps $\mathbf{a}$ to a potential and a sequence of assignments, i.e.,

$$\left(\phi_1^P(\mathbf{a}_1)^{\frac{1}{r_2}} \cdot \phi_2^P(\mathbf{a}_2)^{\frac{1}{r_1}}, \phi_1^A(\mathbf{a}_1) \circ \phi_2^A(\mathbf{a}_2)\right).$$

The assembly of the potential coincides with the LVE version of the operator. The new input $\mathbf{a}$ is projected onto the original argument sequences to get the corresponding inputs $\mathbf{a}_1$ and $\mathbf{a}_2$. The potentials that $\mathbf{a}_1$ and $\mathbf{a}_2$ each map to are scaled accordingly and then multiplied. The scaling is necessary if parfactors represent a different number of ground factors. See Taghipour *et al.* (2013c) for details and examples. The second precondition allows for concatenating the assignments that $\mathbf{a}_1$ and $\mathbf{a}_2$ map to. The postcondition records that applying Operator 2 with inputs g_1, g_2, and θ in G, which removes g_1 and g_2 from G and adds the output parfactor to G, results in a model equivalent to G.

Operator 3, *count-convert*, implements counting a logvar X in a parfactor, which excludes X from the set of regular logvars and thus, may enable another PRV to be eliminated by *max-out*. The inputs are a parfactor $g = \phi(\mathcal{A})^{\mathcal{B}}_{|C}$ and a logvar X in g. The operator has four preconditions: (i) Only one input $A_i \in \mathcal{A}$ contains X. (ii) X is count-normalised w.r.t. $\mathbf{X} \setminus \{X\}$. (iii) No inequality constraint exists between X and any other counted logvar in g. (iv) $A_{i|C}$ does not overlap with a PRV in any other parfactor in G. The first three conditions are identical to count conversion in LVE. The forth condition is new. When counting X, the operator combines histograms of maxed out PRVs, after which it may no longer be possible to trace which valuations map to a particular assignment. Thus, if, e.g., grounding a counted logvar becomes necessary, which uses the *expand* operator, it may not be possible to reconstruct which instance leads to which assignment. The same holds for splitting a CRV through the *expand* operator as a more general case. The forth precondition ensures that the PRV that becomes the (P)CRV does not occur in another part in the model and thus does not need to be split or grounded because of the other occurrences. The operator *multiply* introduced above could establish this

Operator 3 Count Conversion

Operator COUNT-CONVERT

Inputs:

(1) $g = \forall \mathbf{x} \in \pi_{\mathbf{X}}(C) : \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$

(2) $X \in lv(\mathcal{A})$, to count in g

Preconditions:

(1) There is exactly one atom $A_i \in \mathcal{A}$ with $X \in lv(A_i)$.

(2) X is count-normalised w.r.t. $\mathbf{Z} = lv(\mathcal{A}) \setminus \{X\}$ in $\pi_{\mathbf{X}}(C)$.

(3) For all counted logvars $X^{\#}$ in g: $\pi_{X,X^{\#}}(C) = \pi_X(\pi_{\mathbf{X}}(C)) \times \pi_{X^{\#}}(\pi_{\mathbf{X}}(C))$.

(4) For all $g' \in G, g' \neq g : rv(A_{i|C} \cap rv(g) = \emptyset$.

Output: $\forall \mathbf{x}' \in \pi_{\mathbf{X} \setminus \{X\}}(C) : \phi'(\mathcal{A}')^{\mathcal{B}}_{|C}$ such that

(1) $\mathcal{A}' = (A_1, \ldots, A_{i-1}) \circ A'_i \circ (A_{i+1}, \ldots, A_n)$, $A'_i = \#_X[A_i]$, and

(2) for each valuation $\mathbf{a}'$ to $\mathcal{A}'$ with $a'_i = h$,

$$\phi'(\ldots, a_{i-1}, h, a_{i+1}, \ldots) = \left(\prod_{a_i \in \mathcal{R}(A_i)} \phi^P(\ldots, a_{i-1}, a_i, a_{i+1}, \ldots)^{h(a_i)}, \bigcirc_{B \in \mathcal{B}} \frac{1}{r_B} \cdot \sum_{a_i \in \mathcal{R}(A_i)} h(a_i) \cdot \phi^A_B(\ldots, a_{i-1}, a_i, a_{i+1}, \ldots) \right)$$

- $h = \{(a_i, n_i)\}_{i=1}^m$, $m = |\mathcal{R}(A_i)|$, $a_i \in \mathcal{R}(A_i)$, $n_i \in \mathbb{N}$, and $\sum_{a_i \in \mathcal{R}(A_i)} h(a_i) = r$,
- $r = \text{NCOUNT}_{X|\mathbf{Z}}(\pi_{\mathbf{X}}(C))$,
- $r_B = \begin{cases} 1 & \text{if } X \in lv(B) \\ r & \text{otherwise} \end{cases}$

Postcondition: $G \equiv G \setminus \{g\} \cup \{\text{COUNT-CONVERT}(g, X)\}$

precondition for a PRV. Combining all occurrences of A_i in g does not limit MPE-LVE because eliminating A_i requires all occurrences in one parfactor anyway. Doing so before count conversion even prevents counting a logvar several times in different parfactors.

The output of Operator 3 is a parfactor $\phi'(\mathcal{A}')^{\mathcal{B}}_{|C}$ where C and $\mathcal{B}$ are identical to the input parfactor. For $\mathcal{A}'$, the operator replaces A_i with a (P)CRV $\#_X[A_i]$. The range of $\#_X[A_i]$ is a set of histograms, each histogram $h = \{(a_i, n_i)\}_{i=1}^m$ fulfilling that $\sum_{i=1}^m n_i = r = \text{NCOUNT}_{X|\mathbf{Z}}(\pi_{\mathbf{X}}(C))$, $\mathbf{Z} = lv(\mathcal{A}) \setminus \{X\}$. For a valuation $\mathbf{a}' = (\ldots, a_{i-1}, h, a_{i+1}, \ldots)$ of $\mathcal{A}'$, ϕ maps $\mathbf{a}'$ to a tuple of a potential and a sequence of assignments, i.e.,

$$\left(\prod_{a_i \in \mathcal{R}(A_i)} \phi^P(\ldots, a_{i-1}, a_i, a_{i+1}, \ldots)^{h(a_i)}, \bigcirc_{B \in \mathcal{B}} \frac{1}{r_B} \cdot \sum_{a_i \in \mathcal{R}(A_i)} h(a_i) \cdot \phi^A_B(\ldots, a_{i-1}, a_i, a_{i+1}, \ldots) \right)$$

where the calculation of the new potential is identical to count conversion in LVE. The

operator multiplies all potentials that the different $a_i \in \mathcal{R}(A_i)$ in $(\dots, a_{i-1}, a_i, a_{i+1}, \dots)$ map to, raising each potential to the power of $h(a_i) = n_i$ as n_i reveals how many times $\phi(\dots, a_{i-1}, a_i, a_{i+1}, \dots)$ occurs.

As a histogram such as $[1, 1]$ combines instances with different range values, the original ϕ may map to two different assignments. Therefore, the operator combines histograms for the maxed out PRVs in $\mathcal{B}$ by summing over the range of A_i and adding corresponding histograms for a maxed out PRV B. If $X \in lv(B)$, B has been eliminated from g at one point and as such contains all logvars that are not yet eliminated including X. The operator makes the number of instances of X, of which there are r in relation to the remaining logvars, explicit in B: It adds $h(a_i)$ times the histograms mapped to by $\phi(\dots, a_{i-1}, a_i, a_{i+1}, \dots)$ for each a_i. The sum over the range values combined with the different $h(a_i)$ counts leads to r histograms being combined and materialises X in B. If $X \notin lv(B)$, the summation makes the counts inconsistent. As mentioned above, the histograms show how many instances of a maxed out PRV B have a particular value in relation to the remaining logvars. Adding the histograms duplicates the number of instances of B by r, though B does not stand for r instances in relation to X. If $X \notin lv(B)$, then a parfactor has been multiplied into g that has B as a maxed out PRV and the shared PRVs during multiplication did not include A_i. Thus, the histograms that the different $a_i \in \mathcal{R}(A_i)$ map to are identical. The summation yields $r \cdot h$, which, multiplied with $\frac{1}{r}$, leads to h again.

The postcondition asserts that applying Operator 3 with inputs g and X in G, which removes g from G and adds the output parfactor to G, results in an equivalent model compared to G. Next, we look at an example to illustrate the *count-convert* operator.

Example 10.1.5 (Count conversion)**.** Consider $g_1 = \phi_1(Epid, Nat(D), Man(W))^{\mathcal{B}_1}$, $\mathcal{B}_1 = ()$. Assume another parfactor $g_r = \phi_r(Nat(D), Man(W), R(D, W))^{\mathcal{B}}$, $\mathcal{B} = ()$, which contains $Nat(D)$, $Man(W)$, and a new PRV $R(D, W)$ as arguments. The new PRV carries no purpose other than to illustrate the effect of counting a logvar that appears in a maxed out PRV within the running example. In this example, we will compute a solution to the MPE problem for a model of g_1 and g. Assume a function specification as given in Tables 10.3a and 10.3b. The first operator to apply is *max-out* to eliminate $R(D, W)$ from g. $R(D, W)$ fulfils all three preconditions in g. The result is a new parfactor $g'_r = \phi'_r(Nat(D), Man(W))^{\mathcal{B}}$, $\mathcal{B} = (R(D, W))$ as shown in Table 10.3c. g is compiled as described in Example 10.1.4 by choosing the maximum potential and its corresponding assignment for each remaining valuation. As no logvar is eliminated, the resulting histograms are multiplied by $\frac{1}{r}, r = 1$.

In g_1 and g'_r, no other PRV is eliminable as no PRV contains all logvars and additionally, $Nat(D)$ as well as $Man(W)$ do not appear in a single parfactor. A count conversion requires a PRV to occur in a single parfactor. The next operation is a multiplication of g'_r and g_1. Both parfactors represent the same number of ground factors, meaning, the r_i counts in the output are both 1 and the exponent is 1. The shared PRVs are $Nat(D)$ and

Table 10.3: Parfactor specifications and intermediate results for Example 10.1.5

(a) $g_1 = \phi_1(Epid, Nat(D), Man(W))$

$Epid$	$Nat(D)$	$Man(W)$	ϕ
$false$	$false$	$false$	$(4, ())$
$false$	$false$	$true$	$(3, ())$
$false$	$true$	$false$	$(2, ())$
$false$	$true$	$true$	$(1, ())$
$true$	$false$	$false$	$(1, ())$
$true$	$false$	$true$	$(2, ())$
$true$	$true$	$false$	$(3, ())$
$true$	$true$	$true$	$(4, ())$

(b) $g_r = \phi_r(Nat(D), Man(W), R(D, W))$

$Nat(D)$	$Man(W)$	$R(D, W)$	ϕ
$false$	$false$	$false$	$(1, ())$
$false$	$false$	$true$	$(4, ())$
$false$	$true$	$false$	$(3, ())$
$false$	$true$	$true$	$(2, ())$
$true$	$false$	$false$	$(4, ())$
$true$	$false$	$true$	$(3, ())$
$true$	$true$	$false$	$(2, ())$
$true$	$true$	$true$	$(1, ())$

(c) $g'_r = \phi'_r(Nat(D), Man(W))^{(R(D,W))}$

$Nat(D)$	$Man(W)$	ϕ'_r
$false$	$false$	$(4, ([1, 0]))$
$false$	$true$	$(3, ([0, 1]))$
$true$	$false$	$(4, ([0, 1]))$
$true$	$true$	$(2, ([0, 1]))$

(d) $g'_{1r} = \phi'_{1r}(Epid, Nat(D), Man(W))^{(R(D,W))}$

$Epid$	$Nat(D)$	$Man(W)$	ϕ'_{1r}
$false$	$false$	$false$	$(4 \cdot 4, ([1, 0]))$
$false$	$false$	$true$	$(3 \cdot 3, ([1, 0]))$
$false$	$true$	$false$	$(2 \cdot 4, ([0, 1]))$
$false$	$true$	$true$	$(1 \cdot 2, ([0, 1]))$
$true$	$false$	$false$	$(1 \cdot 4, ([0, 1]))$
$true$	$false$	$true$	$(2 \cdot 3, ([0, 1]))$
$true$	$true$	$false$	$(3 \cdot 4, ([0, 1]))$
$true$	$true$	$true$	$(4 \cdot 2, ([0, 1]))$

$Man(W)$. As we do not have standardised apart logvars, the alignment maps D to D and W to W. The new parfactor $g'_{1r} = \phi'_{1r}(Epid, Nat(D), Man(W))^{\mathcal{B}}$, $\mathcal{B} = (R(D, W))$ is shown in Table 10.3d. The new argument sequence is equal to the one from g_1 as g'_r does not carry any additional PRVs. To compile a tuple (ϕ^P, ϕ^A) for a particular valuation, e.g., $\mathbf{a} = (false, false, false)$, *multiply* looks up the tuple for $(false, false, false)$ in g_1 and the tuple for $(false, false)$, the last two entries in $\mathbf{a}$, in g'_r. The tuple from g_1 is $(4, ())$ and from g'_r is $4, ([1, 0])$. Thus, the tuple for $\mathbf{a}$ in g'_{1r} has the potential $4 \cdot 4$ and the assignment sequence $() \circ ([1, 0])$. For valuation $\mathbf{a} = (false, false, true)$, the operator looks up the tuple for $(false, false, true)$ in g_1 and the tuple for $(false, true)$ in g'_r. With $(3, ())$ and $(3, ([0, 1]))$, the new tuple for $\mathbf{a}$ to map to consists of $3 \cdot 3$ and $() \circ ([0, 1])$. The operator assembles the tuples for the remaining valuations in the same way.

Given g'_{1r}, a count conversion of D or W is possible (intermediate result of size 12). The operator *count-convert* counts, e.g., D in g'_{1r} with $r = 2$, yielding a parfactor $g'' = \phi''(Epid, \#_D[Nat(D)], Man(W))^{\mathcal{B}}_{|\top}$, $\mathcal{B} = (R(D, W))$ as shown in Table 10.5a. The new CRV $\#_D[Nat(D)]$ has histograms of the form $[0, 2]$, $[1, 1]$, and $[2, 0]$, with the first position

Table 10.5: Intermediate results for Example 10.1.5 continued

(a) $\phi''(Epid, \#_D[Nat(D)], Man(W))^{(R(D,W))}$

$Epid$	$\#_D[Nat(D)]$	$Man(W)$	ϕ''
$false$	$[0,2]$	$false$	$(8^0 \cdot 16^2, (0 \cdot [0,1] + 2 \cdot [1,0])) = (256, ([2,0]))$
$false$	$[0,2]$	$true$	$(2^0 \cdot 9^2, \ (0 \cdot [0,1] + 2 \cdot [1,0])) = (81, \ ([2,0]))$
$false$	$[1,1]$	$false$	$(8^1 \cdot 16^1, (1 \cdot [1,0] + 1 \cdot [0,1])) = (128, ([1,1]))$
$false$	$[1,1]$	$true$	$(2^1 \cdot 9^1, \ (1 \cdot [0,1] + 1 \cdot [1,0])) = (18, \ ([1,1]))$
$false$	$[2,0]$	$false$	$(8^2 \cdot 16^0, (2 \cdot [0,1] + 0 \cdot [1,0])) = (64, \ ([0,2]))$
$false$	$[2,0]$	$true$	$(2^2 \cdot 9^0, \ (2 \cdot [0,1] + 0 \cdot [1,0])) = (4, \ ([0,2]))$
$true$	$[0,2]$	$false$	$(12^0 \cdot 4^2, (0 \cdot [0,1] + 2 \cdot [0,1])) = (16, \ ([0,2]))$
$true$	$[0,2]$	$true$	$(8^0 \cdot 6^2, \ (0 \cdot [0,1] + 2 \cdot [0,1])) = (36, \ ([0,2]))$
$true$	$[1,1]$	$false$	$(12^1 \cdot 4^1, (1 \cdot [0,1] + 1 \cdot [0,1])) = (48, \ ([0,2]))$
$true$	$[1,1]$	$true$	$(8^1 \cdot 8^1, \ (1 \cdot [0,1] + 1 \cdot [0,1])) = (64, \ ([0,2]))$
$true$	$[2,0]$	$false$	$(12^2 \cdot 4^0, (2 \cdot [0,1] + 0 \cdot [0,1])) = (144, ([0,2]))$
$true$	$[2,0]$	$true$	$(8^2 \cdot 6^0, \ (2 \cdot [0,1] + 0 \cdot [0,1])) = (64, \ ([0,2]))$

(b) $\phi'''(Epid, \#_D[Nat(D)])^{(Man(W),R(D,W))}$

$Epid$	$\#_D[Nat(D)]$	ϕ'''
$false$	$[0,2]$	$(256^2, (2 \cdot [0,1], 2 \cdot [2,0]))$
$false$	$[1,1]$	$(128^2, (2 \cdot [0,1], 2 \cdot [1,1]))$
$false$	$[2,0]$	$(64^2, \ (2 \cdot [0,1], 2 \cdot [0,2]))$
$true$	$[0,2]$	$(36^2, \ (2 \cdot [1,0], 2 \cdot [0,2]))$
$true$	$[1,1]$	$(64^2, \ (2 \cdot [1,0], 2 \cdot [0,2]))$
$true$	$[2,0]$	$(144^2, (2 \cdot [1,0], 2 \cdot [0,2]))$

(c) $\phi(Epid)^{(\#_D[Nat(D)],Man(W),R(D,W))}$

$Epid$	ϕ
$false$	(65536, ([0,2],[0,2],[4,0]))
$true$	(20736, ([2,0],[2,0],[0,4]))

referring to $Nat(D) = true$ and the second position to $Nat(D) = false$. Let us look at the valuation $(false, [0,2], false)$ as an example for how the output is assembled. To get the tuple for this valuation, the operator inserts the range values of $Nat(D)$ into the position of $[0,2]$, leading to the valuations $(false, true, false)$ and $(false, false, false)$ for g'_{1r}. In g'_{1r}, the operator looks up the tuples for both valuation, i.e., $(8, ([0,1]))$ and $(16, ([1,0]))$. To assemble the new potential, the operator multiplies 8^0 and 16^2, each raised to the power of the respective count in $[0,2]$. To assemble the assignments, the operator adds the histograms multiplied with the histogram counts, i.e., $0 \cdot [0,1] + 2 \cdot [1,0]$. The rationale behind both assemblies is that given a histogram of $[0,2]$, the operator has to incorporate two times the valuation of $false$ for $Nat(D)$ and the valuation of $true$ not at all. Consider the valuation $(false, [1,1], false)$. Here, the tuples to combine from g'_{1r} are again $(8, ([0,1]))$ and $(16, ([1,0]))$. But, the histogram counts are now different, which leads to $8^1 \cdot 16^1$ for the new potential and $1 \cdot [0,1] + 1 \cdot [1,0]$ for the new assignment.

Valuation $(false, [1,1], false)$ reveals why histograms encode assignments. Behind $(false, [1,1], false)$, there are two valuations, ($false$,$true$,$false$) and ($false$,$false$,$false$) for the two $Nat(D)$ instances. The first valuation maps to $[0,1]$, i.e., $R(D,W) = false$, the second maps to $[1,0]$, i.e., $R(D,W) = true$. Thus, the operator stores a histogram $[1,1]$ to encode that both assignments occur given that $Epid = false$, $Man(W) = false$ and one instance of $Nat(D)$ has the value $true$ and the other the value $false$.

In g'', $Man(W)$ is eliminable. Maxing out $Man(W)$ eliminates W, which means the histograms in the result are multiplied by $r = 2$. The result $\phi'''(Epid, \#_D[Nat(D)])^{\mathcal{B}}_{|\top}$, $\mathcal{B} = (Man(W), R(D,W))$ is depicted in Table 10.5b. Next, $\#_D[Nat(D)]$ is eliminable, yielding a parfactor with $Epid$ as argument (Table 10.5c). For $\#_D[Nat(D)]$, each a_{max} already is a histogram. Eliminating $Epid$ leads to a final result of an MPE for a model of g_1 and g, which is the line marked in gray in Table 10.5c. The assignment sequence $([0,1],[0,2],[0,1],[2,0])$ for $(Epid, \#_D[Nat(D)], Man(W), R(D,W))$ contains only peak-shaped histograms, that is each PRV has a uniform assignment for all its instances, i.e.,

$$Epid = false \wedge \forall D, W : Nat(D) = false \wedge Man(W) = false \wedge R(D,W) = true$$

As mentioned before, there exist three generalised counting operators, *count-convert*, *merge*, and *merge-count* (Taghipour and Davis, 2012), which (i) count-convert a logvar that appears in more than one PRV, (ii) merge CRVs with counted logvars of the same domain into one CRV, and (iii) merge-count a PRV into an existing CRV with an inequality constraint between their logvars. The first operator is a generalisation of the LVE count conversion as introduced in Section 3.2, which in its basic workflow works as count conversion, forming histograms for a set of PRVs instead of one PRV. The second operator combines histograms of two CRVs accordingly without any manipulation of potentials, while the third operator merges a given histogram of a CRV with range values of the PRV to merge into the CRV. The first and third operators follow the same idea as Operator 3 when incorporating assignments, while the second operator has no influence on potentials and assignments alike. See Appendix A.4 for full specifications.

The operators *split*, *expand*, *count-normalise*, and *ground-logvar* proceed as before, duplicating a parfactor and partitioning constraints. The operators *split* and *expand*, which implements splitting for (P)CRVs, establish non-overlapping PRVs, e.g., to ensure the first precondition of *max-out*. When splitting a parfactor on a PRV that refers to a subset of the constants in the parfactor, *split* partitions a constraint into two parts, one for the subset of constants and one for the remaining part. Splitting involves logvars, which may occur in a maxed out PRV B. However, the split has no consequence for B and its histograms as a histogram specifies counts for each instance of the remaining logvars, not for the overall number of instances, which changes with *split*.

(P)CRVs are split by applying *expand* to preserve the count normalisation of the counted logvar. Expanding a (P)CRV $\#_X[R(\mathbf{X})]_{|C}$ on a PRV $R(\mathbf{X})_{|C'}$ or (P)CRV $\#_X[R(\mathbf{X})]_{|C'}$ entails in part splitting histograms accordingly. The result contains one

parfactor in which X is replaced by two new logvars, one for the X instances in C' and one for the remaining X instances and $\#_X[R(\mathbf{X})]_{|C}$ is replaced by two new (P)CRVs. For MPE, *expand* has a new precondition. If X appears in a maxed out PRV B, the histograms of B have already incorporated X when X was counted. After applying a sequence of LVE operators, it may no longer be possible to reconstruct how $R(\mathbf{X})$ influences the assignments of B and split the histograms of B accordingly. Therefore, X must not appear in any maxed out PRV. The precondition has also led to the new precondition of *count-convert*. As a (P)CRV can no longer be expanded if a maxed out PRV contains its counted logvar, the new precondition ensures that all instances of the underlying PRV are combined in a single parfactor. If a PRV no longer appears in any other parfactor, it cannot cause an expansion of the (P)CRV.

The operator *count-normalise* duplicates a parfactor and partitions constraints to count-normalise a set of logvars w.r.t. another set of logvars, e.g., to ensure the third precondition of *max-out*. The operator *ground-logvar* implements grounding a (non-counted) logvar X in the arguments of a parfactor through duplication and partitioning as well (one partition for each instance of X). The same argument as for *split* applies for both *count-normalise* and *ground-logvar* w.r.t. the histograms remaining consistent. See Appendix A.2 for full specifications. Next, we set up MPE-LVE.

MPE-LVE The MPE version of LVE computes a most likely assignment for each model PRV without evidence. The algorithm has the same workflow as the original LVE but applies the operators specified in the previous paragraphs. Algorithm 6 shows MPE-LVE with model G and evidence $\mathbf{E}$ as inputs to answer MPE($\mathbf{E}$).

At the beginning of an algorithm run, each parfactor in G maps to an empty sequence of histograms. Having absorbed $\mathbf{E}$ in G, Alg. 6 maxes out all PRVs in G, applying transforming operators if necessary. The heuristics for choosing the next operator is still based on the expected size of the intermediate result. Eliminating all PRVs in G leads to a set of parfactors with no arguments, which are multiplied into one parfactor $\phi()^{\mathcal{B}}_{|C}$

Algorithm 6 MPE-LVE

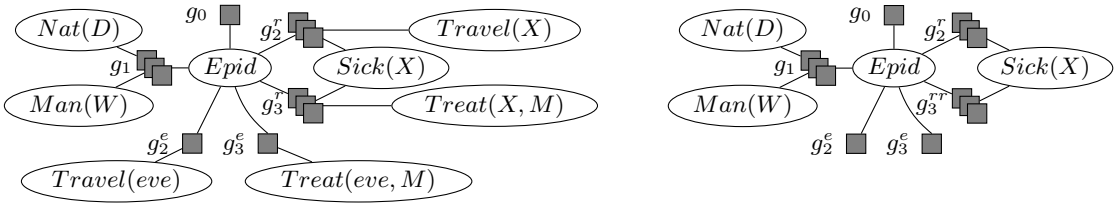
```
procedure MPE-LVE(Model G, Evidence E)
    G ← Absorb E in G                          ▷ Shatters G on E if necessary
    while G contains not maxed out PRVs do
        if there exists a PRV A eliminable then
            G ← Max out A in G
        else
            G ← Apply transforming operator applicable in G
    G ← Multiply remaining parfactors in G
    return G                                   ▷ Contains one parfactor
```

to combine the assignments in one parfactor and get the (unnormalised) potential of the MPE assignment. $\phi^A()$ holds how many instances have a specific value for each PRV in $\mathcal{B} = rv(G) \setminus rv(\mathbf{E})$. One could output G without multiplying parfactors if only interested in the assignments. Next, we look at an MPE query MPE($sick(eve)$) for G_{ex}.

Example 10.1.6 (MPE-LVE)**.** Answering MPE($sick(eve)$) starts with absorbing evidence $sick(eve)$ in G_{ex}, which proceeds as with LVE, leading to splits in g_2 and g_3. Figure 10.1a shows G_{ex} as a parfactor graph after absorbing evidence. Proceeding with increasing size of intermediate results, MPE-LVE maxes out

- $Travel(eve)$ in g_2^e, resulting in $g_2^e = \phi_2'(Epid)^{(Travel(eve))}$,
- $Treat(eve, M)$ in g_3^e, resulting in $g_3^e = \phi_3'(Epid)^{(Treat(eve,M))}$,
- $Travel(X)$ in g_2^r, resulting in $g_2^r = \phi_2''(Epid, Sick(X))^{(Travel(X))}_{|C_2^r}$, and
- $Treat(X, M)$ in g_3^r, resulting in $g_3^r = \phi_3''(Epid, Sick(X))^{(Treat(X,M))}_{|C_3'}$.

Figure 10.1b shows the remaining model after performing the four max-out operations. As with LVE and Example 3.2.5, no further maxing out is possible. Next to grounding each logvar (8 to 16 mappings), MPE-LVE can count either D or W (each 12 mappings), and multiply g_2^r and g_3^r (4 mappings). Thus, MPE-LVE multiplies g_2^r and g_3^r into $g_{23} = \phi_{23}(Epid, Sick(X))^{\mathcal{B}}_{C_3'}$, $\mathcal{B} = (Travel(X), Treat(X, M))$, concatenating the assignments for $Travel(X)$ and $Treat(X, M)$. LVE-MPE can now eliminate $Sick(X)$, resulting into $g_{23}' = \phi_{23}'(Epid)^{\mathcal{B}}_{C_3'}$, $\mathcal{B} = (Sick(X), Travel(X), Treat(X, M))$, adding histograms for $Sick(X)$. Maxing out $Sick(X)$ eliminates X from g_{23}, i.e., $r = 2$, leading to a multiplication of all histograms of $(Sick(X), Travel(X), Treat(X, M))$ with 2, as each PRV contains X. Next, MPE-LVE randomly chooses to count D, resulting in $g_1^\# = \phi_1^\#(Epid, \#_D[Nat(D)], Man(W))^{()}$. As no maxed out PRV exists in g_1, the count conversion coincides with a count conversion in LVE. In $g_1^\#$, $Man(W)$ contains all (non-counted) logvars and appears only in this parfactor, and W is count-normalised ($r = 2$). So, MPE-LVE eliminates $Man(W)$, followed by eliminating $\#_D[Nat(D)]$,



(a) After absorbing evidence

(b) After four max-out operations

Figure 10.1: Parfactor graph of G_{ex} for Example 10.1.6

resulting in $g'_1 = \phi'_1(Epid)^{\mathcal{B}}$, $\mathcal{B} = (\#_D[Nat(D)], Man(W))$. The remaining randvar is *Epid*. LVE multiplies all remaining parfactors (g^e_2, g^e_3, g'_{23}, g'_1, g_0) into one parfactor $g = \phi(Epid)^{\mathcal{B}}_{C'_3 \times C_1}$, $\mathcal{B} = (Travel(eve), Treat(eve, M), Sick(X), Travel(X), Treat(X, M), \#_D[Nat(D)], Man(W))$ to max out *Epid*. The result is an assignment, which could have the following form after rewriting peak-shaped histograms, given concrete specifications for the potential functions:

$$\begin{aligned} &Epid = false \wedge\ Travel(eve) = false \wedge \forall M : (Treat(eve, M) = true) \\ \wedge\ &\forall X : (Sick(X) = false \wedge Travel(X) = true \wedge \forall M : (Treat(X, M) = false)) \\ \wedge\ &\forall D, W : (Nat(D) = false \wedge Man(W) = false) \end{aligned}$$

The above example has uniform assignments for all PRVs. Assignments where histograms have counts in at least two positions may occur if a (P)CRV is part of the model specification where a histogram that is not uniform is chosen as for a maximum assignment. Another case of histograms without uniform assignments occurs with parfactors where a constraint encodes $X \neq Y$. The following example shows such an MPE.

Example 10.1.7. De Salvo Braz *et al.* (2006) present an example about business companies B being in a partnership around a certain product P and whether those companies are retailers. There are 15 companies and an un-disclosed number of products in their example. We choose 3 companies and 1 product for this example to keep the numbers small. The parfactors are

$$\phi(Prtnrs(P, B_1, B_2))^{()}_{|C} \qquad \phi(Prtnrs(P, B_1, B_2), Rtl(B_1), Rtl(B_2))^{()}_{|C}$$

in which the PRVs are boolean and the constraints encode $B_1 \neq B_2$. Multiplying both parfactors and then maxing out the PRV $Prtnrs(P, B_1, B_2)$ yields a parfactor $\phi(Rtl(B_1), Rtl(B_2))^{\mathcal{B}}_{|C}$, $\mathcal{B} = (Prtnrs(P, B_1, B_2))$. Let the specification in Table 10.7a be the result. Counting B_1 yields $\phi(\#_{B_1}[Rtl(B_1)], Rtl(B_2))^{\mathcal{B}}_{|C}$, the count conversion proceeding as with LVE. The result is depicted in Table 10.7b. But, the result does not allow to eliminate $Retail(B_2)$ as it overlaps with $Retail(B_1)$ in the CRV. Following the operator given in Appendix A.4, merge-counting procures a joint CRV for both entries, shown in Table 10.7c. The assignment for $\#_{B_1}[Rtl(B_1)]$ with maximum potential is $[1, 2]$ (and $[2, 1]$) with an assignment of $[4, 2]$ (and $[2, 4]$) for $Prtnrs(P, B_1, B_2)$. The histograms are not peak-shaped. In other words, the result says that there exists one retail company and two non-retail companies (two retail and one non-retail) and of the six possible partnerships, four exist and two do not (two and four).

MPE-LVE allows for computing a lifted solution to an MPE problem, leveraging the relational structure of the underlying model. Next, we present how to compute a solution to an MPE query with MPE-LJT, leveraging the cluster representation of FO jtrees.

Table 10.7: Example parfactor for merge-counting

(a) $\phi(Rtl(B_1), Rtl(B_2))|_C^{(Prtnrs(P,B_1,B_2))}$

$Rtl(B_1)$	$Rtl(B_2)$	ϕ
$false$	$false$	$(1, ([1,0]))$
$false$	$true$	$(2, ([1,0]))$
$true$	$false$	$(3, ([0,1]))$
$true$	$true$	$(1, ([0,1]))$

(b) $\phi(\#_{B_1}[Rtl(B_1)], Rtl(B_2))|_C^{(Prtnrs(P,B_1,B_2))}$

$\#_{B_1}[Rtl(B_1)]$	$Rtl(B_2)$	ϕ
$[0,2]$	$false$	$(1^2, ([2,0]))$
$[0,2]$	$true$	$(2^2, ([2,0]))$
$[1,1]$	$false$	$(3 \cdot 1, ([1,1]))$
$[1,1]$	$true$	$(1 \cdot 2, ([1,1]))$
$[2,0]$	$false$	$(3^2, ([0,2]))$
$[2,0]$	$true$	$(1^2, ([0,2]))$

(c) $\phi(\#_{B_1}[Rtl(B_1)])|_C^{(Prtnrs(P,B_1,B_2))}$

$\#_{B_1}[Rtl(B_1)]$	ϕ
$[0,3]$	$(1^3, (3 \cdot [2,0])) = (1, ([6,0]))$
$[1,2]$	$(4 \cdot 3^2, (1 \cdot [2,0] + 2 \cdot [1,1])) = (36, ([4,2]))$
$[2,1]$	$(2^2 \cdot 9, (2 \cdot [1,1] + 1 \cdot [0,2])) = (36, ([2,4]))$
$[3,0]$	$(1^3, (3 \cdot [0,2])) = (1, ([0,6]))$

MPE-LJT Using LJT to compute a solution to an MPE problem has the advantage of a reduced space to search for the next PRV to max out. We adapt LJT to compute an MPE by calculating messages using MPE-LVE. Algorithm 7 shows MPE-LJT with model G and evidence $\mathbf{E}$ as input to answer MPE($\mathbf{E}$). MPE-LJT constructs an FO jtree J for G, enters evidence $\mathbf{E}$ into J, and passes messages in J, which only needs an inward pass. At the innermost node, MPE-LJT maxes out the remaining PRVs and returns a most likely assignment for $rv(G) \setminus rv(\mathbf{E})$.

The first two steps are identical to LJT, while message passing changes. Messages carry over the assignments of maxed out PRVs to some innermost parcluster, at which

Algorithm 7 MPE-LJT

procedure MPE-LJT(Model G, Evidence $\mathbf{E}$)
  Construct an FO jtree $J = (V, E)$ for G
  Enter $\mathbf{E}$ into J
  Pass messages on J based on Condition (1) ▷ MPE-LVE as subroutine
  Get $\mathbf{C}_i \in V$ where Condition (2) triggers
  $G' \leftarrow G_i \cup \bigcup_{j \in nbs(i)} m_{ji}$
  return MPE-LVE(G', $\emptyset$)

the remaining PRVs are maxed out. A message m_{ij} from parcluster $\mathbf{C}_i$ with local model G_i to parcluster $\mathbf{C}_j$ is still a set of parfactors, each with a subset of $\mathbf{S}_{ij}$ as arguments. MPE-LJT computes m_{ij} by passing to MPE-LVE a query over $\mathbf{S}_{ij}$ and a model $G' = G_i \cup \bigcup_{k \in nbs(i), k \neq j} m_{ki}$, i.e., MPE-LVE($G'$, $\mathbf{S}_{ij}$, $\emptyset$). MPE-LVE then eliminates $\mathbf{C}_i \setminus \mathbf{S}_{ij}$ from G', skipping again the last step before returning the result. As G' may contain messages, the parfactors in G' may contain already maxed out PRVs in contrast to the example run of MPE-LVE before. Since maxed out PRVs in messages do not interfere with the elimination order of the next parcluster due the FO jtree properties, the maxed out PRVs do not pose any challenges to a lifted computation. The original LJT message passing has had two conditions based on which messages flow at a parcluster $\mathbf{C}_i$:

(1) $\mathbf{C}_i$ has received messages from all neighbours but $\mathbf{C}_j$: $\mathbf{C}_i$ sends a message to $\mathbf{C}_j$.

(2) $\mathbf{C}_i$ has received a message from its remaining neighbour $\mathbf{C}_j$: $\mathbf{C}_i$ sends messages to all other neighbours.

In MPE-LJT, messages flow based on Condition (1). When Condition (2) first triggers at a parcluster $\mathbf{C}_i$, it indicates to MPE-LJT to use $\mathbf{C}_i$ to finish answering an MPE query, maxing out the PRVs in $\mathbf{C}_i$ from its local model and received messages. Combined with the assignments carried over through the messages, maxing out the remaining PRVs leads to the complete MPE w.r.t. G and $\mathbf{E}$. For the last step of maxing out, MPE-LJT uses MPE-LVE as specified in Alg. 6. Let us consider MPE($sick(eve)$) for G_{ex} again.

Example 10.1.8 (MPE-LJT)**.** Computing an MPE for G_{ex} starts with constructing an FO jtree J as seen in Fig. 4.1b with parclusters $\mathbf{C}_1$, $\mathbf{C}_2$, and $\mathbf{C}_3$ and local models $G_1 = \{g_0, g_1\}$, $G_2 = \{g_2\}$, and $G_3 = \{g_3\}$.

Entering $sick(eve)$ as evidence into J causes changes in G_2 and G_3, leading to $G_2 = \{g_2^e, g_2^r\}$ and $G_3 = \{g_3^e, g_3^r\}$, depicted in Fig. 4.3, where the constraints in g_2^r and g_3^r restrict X to *alice* and *bob*. Message passing commences. $\mathbf{C}_1$ and $\mathbf{C}_3$ prepare a message for $\mathbf{C}_2$. At $\mathbf{C}_1$, logvar D is counted to allow for maxing out $Man(W)$ before maxing out $\#_D[Nat(D)]$. Message m_{12} consists of

$$g_0 = \phi_0(Epid) \qquad \text{and}$$
$$g_1'' = \phi_1''(Epid)^{\mathcal{B}_1}, \mathcal{B}_1 = (\#_D[Nat(D)], Man(W)).$$

At $\mathbf{C}_3$, $Treat(eve, M)$ is maxed out from g_3^e and $Treat(X, M)$ from g_3^r. Message m_{32} contains the results of these two max-out operations,

$$g_3^{e\prime} = \phi_3'(Epid)^{\mathcal{B}_3^e}, \qquad \mathcal{B}_3^e = (Treat(eve, M)) \text{ and}$$
$$g_3^{r\prime} = \phi_3'(Epid, Sick(X))^{\mathcal{B}_3^r}_{|C_3}, \mathcal{B}_3^r = (Treat(X, M))$$

$\mathbf{C}_2$ receives both messages, which triggers Condition (2), and LJT uses MPE-LVE to max out the PRVs in $\mathbf{C}_2$ to complete the MPE solution. Maxing out $Travel(eve)$ in g_2^e

and $Travel(X)$ in g_2^r yields

$$g_2^{e\prime} = \phi_2'(Epid)^{\mathcal{B}_2^e}, \qquad \mathcal{B}_2^e = (Travel(eve)) \text{ and}$$
$$g_2^{r\prime} = \phi_2'(Epid, Sick(X))^{\mathcal{B}_2^r}_{|C_2}, \mathcal{B}_2^r = (Travel(X)).$$

MPE-LVE multiplies $g_2^{r\prime}$ and $g_3^{r\prime}$ from m_{32} to max out $Sick(X)$. The result is a parfactor

$$g_{23}' = \phi_{23}'(Epid)^{\mathcal{B}_{23}}_{|C_3}, \mathcal{B}_{23} = (Sick(X), Travel(X), Treat(X, M)).$$

The last randvar to max out is *Epid*, for which MPE-LVE multiplies the remaining parfactors g_0, g_1'', $g_2^{e\prime}$, g_{23}', and $g_3^{e\prime}$ into one, combining all assignments in one parfactor. After maxing out *Epid*, MPE-LVE outputs the resulting parfactor containing the assignments for all model PRVs, which coincides with the assignment MPE-LVE produces on G_{ex}.

With MPE-LJT, we have an inference algorithm for solving an MPE problem that leverages the relational structures of a model through lifting and speeds up its inference by using its FO jtree, which exploits the sparseness of a model. Next, we take a closer look at queries for more general MAP assignments, where we further exploit the cluster representation of LJT to determine liftable MAP queries.

10.2 Lifted Algorithms for MAP Assignments

MPE-LVE and MPE-LJT provide efficient means to compute solutions to MPE queries, which ask for a maximum a posteriori assignment to all model PRVs without evidence assigned. An MAP query MAP($\mathbf{U}|\mathbf{E} = \mathbf{e}$) asks for a maximum a posteriori assignment to $\mathbf{U}$, a subset of model PRVs for which there is no evidence assigned. An algorithm has to sum out non-query terms and then max out the query terms $\mathbf{U}$. In the following paragraphs, we present MAP-LVE and MAP-LJT for such MAP queries.

MAP-LVE LVE for MAP queries works according to the semantics of MAP queries. Algorithm 8 shows MAP-LVE with model G, query terms $\mathbf{U}$, and evidence $\mathbf{E}$ as inputs to answer MAP($\mathbf{U}|\mathbf{E}$). MAP-LVE begins like LVE: It shatters G on $\mathbf{U}$, absorbs $\mathbf{E}$ in G, and then eliminates all non-query terms $rv(G) \setminus \mathbf{U}$ using the original LVE operators (lines 4–8). After having eliminated all non-query terms, MAP-LVE eliminates all query terms in $\mathbf{U}$ using the MPE-LVE operators (lines 9–13).

In the style of parameterised queries, the query terms $\mathbf{U}$ can be parameterised to denote a whole set of instances of a PRV. With arbitrary subsets $\mathbf{U}$, the computation may lead to prohibitively large intermediate results as well as groundings due to logvars in $\mathbf{U}$, both problems stemming from the enforced elimination order of first eliminating $rv(G) \setminus \mathbf{U}$ and then $\mathbf{U}$. Consider the MAP query MAP($\{Travel(X), Treat(X, M)\}_{|\top}|sick(eve)$) with query terms $\{Travel(X), Treat(X, M)\}_{|\top}$ and evidence $sick(eve)$ for G_{ex}.

Algorithm 8 MAP-LVE

```
1: procedure MAP-LVE(Model G, Query terms U, Evidence E)
2:     G ← Shatter G on U
3:     G ← Absorb E in G                          ▷ Shatters G on E if necessary
4:     while G contains non-query PRV do                                  ▷ LVE like
5:         if there exists a PRV A eliminable then
6:             G ← Sum out A in G
7:         else
8:             G ← Apply transforming operator applicable in G
9:     while G contains query PRV do                                  ▷ MPE-LVE like
10:        if there exists a PRV A eliminable then
11:            G ← Max out A in G
12:        else
13:            G ← Apply transforming operator applicable in G
14:    G ← Multiply remaining parfactors in G
15:    return G                                          ▷ Contains one parfactor
```

Example 10.2.1 (MAP-LVE)**.** Given MAP($Travel(X), Treat(X, M)|sick(eve)$), shattering G_{ex} on $\{Travel(X), Treat(X, M)\}_{|\top}$ does not change G_{ex}. Absorbing evidence results in g_2^e, g_2^r, g_3^e, and g_3^r, with the new G_{ex} as shown in Fig. 10.1a. MAP-LVE has to eliminate the PRVs $Nat(D)$, $Man(W)$, $Sick(X)$, and $Epid$ with LVE.

Eliminating $Nat(D)$ and $Man(W)$ follows the same steps as before, counting logvar D, summing out $Man(W)$, followed by summing out $\#_D[Nat(D)]$, yielding a parfactor $\phi_1''(Epid)$. To sum out $Sick(X)$, MAP-LVE has to multiply g_2^r and g_3^r into a parfactor $g_{23} = \phi_{23}(Epid, Sick(X), Travel(X), Treat(X, M))$, which has $2^4 = 16$ possible valuations, which is already larger than the largest parfactor during answering an MPE query with MPE-LVE. In g_{23}, $Sick(X)$ does not contain all logvars. Thus, MAP-LVE counts M in g_{23}, leading to a parfactor $g'_{23} = \phi'_{23}(Epid, Sick(X), Travel(X), \#_M[Treat(X, M)])$ with $2^3 \cdot 3 = 24$ possible valuations. From g'_{23}, MAP-LVE eliminates $Sick(X)$. The remaining randvar to sum out is $Epid$, which requires MAP-LVE to multiply all parfactors into one and count M of $Treat(eve, M)$. The result is a parfactor with arguments $Epid$, $Travel(eve)$, $\#_M[Treat(eve, M)]$, $Travel(X)$, and $\#_M[Treat(X, M)]$) with $2^3 \cdot 3^2 = 72$ possible valuations. Eliminating $Epid$ yields the following parfactor, which no longer contains non-query terms and represents the starting point for max-out operations:

$$\phi'(Travel(eve), \#_M[Treat(eve, M)], Travel(X), \#_M[Treat(X, M)]) \tag{10.3}$$

MAP-LVE maxes out $\#_M[Treat(X, M)]$, which leads to a smaller intermediate result than maxing out $Travel(X)$. Afterwards, MAP-LVE maxes out $Travel(X)$, which eliminates X as well, followed by maxing out $\#_M[Treat(eve, M)]$ and $Travel(eve)$, the last

Algorithm 9 MAP-LJT

procedure MAP-LJT(Model G, Query terms $\{\mathbf{U}_k\}_{k=1}^m$, Evidence $\mathbf{E}$)
  Construct an FO jtree $J = (V, E)$ for G
  Enter $\mathbf{E}$ into J
  Pass messages on J ▷ LVE as subroutine
  for each $\mathbf{U}_k \in \{\mathbf{U}_k\}_{k=1}^m$ **do**
    Find a subtree J' s.t. $\mathbf{U}_k \subseteq rv(J')$
    Extract a submodel G' from J'
    MAP-LVE(G', $\mathbf{U}_k$, $\emptyset$) ▷ Output or store result

two PRVs no longer containing uncounted logvars. The result are assignments for PRVs $Travel(eve)$, $\#_M[Treat(eve, M)]$, $Travel(X)$, and $\#_M[Treat(X, M)]$, which fall under $\{Travel(X), Treat(X, M)\}_{|\top}$.

The result exhibits how evidence affects the result representation given an MAP query similar to parameterised queries. The given example also shows how intermediate results can become very large. Assuming *Epid* appears in $\mathbf{U}$ as well, the above example would not have required multiplying all parfactor into one parfactor so early, which would also avoid the second count conversion of M. An FO jtree allows for an easy check of liftable MAP queries that do not produce larger intermediate results than the MPE versions.

MAP-LJT Similar to MAP-LVE starting as LVE, MAP-LJT begins like LJT for parameterised (conjunctive) queries, passing messages using LVE and then, extracting a submodel G' that covers the query terms. For MAP queries though, MAP-LJT passes on G' and the query terms to MAP-LVE instead of LVE. MAP-LVE sums out the remaining non-query terms and finishes with maxing out the query terms. In true LJT fashion, Alg. 9 shows MAP-LJT with a model G, a *set of MAP query terms* $\{\mathbf{U}_k\}_{k=1}^m$, and evidence $\mathbf{E}$ as input to answer a set of MAP queries MAP($\mathbf{U}_k|\mathbf{E}$). Let us consider MAP($\{Travel(X), Treat(X, M)\}_{|\top}|sick(eve)$) in G_{ex} again.

Example 10.2.2 (MAP-LJT)**.** For an MAP query with $\{Travel(X), Treat(X, M)\}_{|\top}$ as query terms and $sick(eve)$) as evidence in G_{ex}, MAP-LVE constructs an FO jtree and enters evidence as in Example 10.1.8. Messages flow from parclusters $\mathbf{C}_1$ and $\mathbf{C}_3$ to parcluster $\mathbf{C}_2$ and back as shown in Example 4.3.4. Afterwards, MAP-LJT finds a subtree for $\{Travel(X), Treat(X, M)\}_{|\top}$, which consists of $\mathbf{C}_2$ and $\mathbf{C}_3$ and extracts a submodel G' of local models G_2 and G_3 as well as message m_{12} from outside the subtree, which coincides with Example 8.1.2 about answering $P(Travel(eve), Treat(eve, injection)|sick(eve))$. To answer the MAP query, MAP-LJT now has to eliminate $Sick(X)$ and *Epid* by summing out and $Travel(X)$ and $Treat(X, M)$ by maxing out, for which it uses MAP-LVE with $\{Travel(X), Treat(X, M)\}_{|\top}$ as the query terms and G' as the model. Evidence

Algorithm 10 J-MPE-LJT with an FO jtree as input for MPE-LJT

procedure J-MPE-LJT(FO jtree J)
  Pass messages on J based on Condition (1) ▷ MPE-LVE as subroutine
  Get $\mathbf{C}_i \in V$ where Condition (2) triggers
  $G' \leftarrow G_i \cup \bigcup_{j \in nbs(i)} m_{ji}$
  return MPE-LVE(G', $\emptyset$)

is empty. MAP-LVE counts M in the product of g_2^r and g_3^r and eliminates $Sick(X)$ from the result as seen in Example 10.2.1. To eliminate $Epid$, MAP-LVE multiplies all remaining parfactors into one, counts the second M logvar, and sums out $Epid$, leading again to a parfactor as in Expression (10.3) from the MAP-LVE example calculation with arguments $Travel(eve)$, $\#_M[Treat(eve, M)]$, $Travel(X)$, and $\#_M[Treat(X, M)]$. From this parfactor, MAP-LVE maxes out the remaining PRVs and returns the result.

The example still leads to the same large intermediate parfactor as before. But, MAP-LJT allows for reusing messages, avoiding repeated eliminations. In the example, MAP-LJT avoids eliminating $Nat(D)$ and $Man(W)$ again.

Bounded MAP Queries FO jtrees allow for identifying *bounded* MAP queries, i.e., MAP queries without a blowup of intermediate parfactor sizes. Such MAP queries cover complete separators or subtrees of an FO jtree, where outside PRVs are eliminable during message passing with LVE and inner PRVs are maxed out with MPE-LVE or MPE-LJT. Consider a query MAP($\{Treat(X, M), Sick(X), Epid\}_{|\top}|sick(eve)$), which asks for an assignment to all PRVs in parcluster $\mathbf{C}_3$. LJT can use MPE-LVE with local model G_3 and message m_{23} as input to efficiently answer the MAP query. With an MAP query over a subtree, MAP-LJT can use MPE-LJT to further leverage the FO jtree.

Algorithm 10 formalises MPE-LJT for a subtree of an FO jtree, named J-MPE-LJT. J-MPE-LJT takes an FO jtree J as input, which is assumed to be a subtree of a larger FO jtree $\hat{J}$. J-MPE-LJT answers an MPE query MPE($rv(J)$), which is an MAP query MAP($rv(J)$) in $\hat{J}$. Evidence is not part of the algorithm as evidence is handled in $\hat{J}$. On J, J-MPE-LJT proceeds as MPE-LJT, sending messages inward and completing the assignment computation with MPE-LVE at a central parcluster. The upcoming section presents a combined LJT version for probability as well as assignment queries that reuses an FO jtree and messages as much as possible for a set of queries of varying types.

10.3 A Variety of Queries

For a set of probability and assignment queries, LJT shows its particular strength as it is able to reuse an FO jtree and messages between various queries of different types. Whereas the LVE versions compute answers to corresponding queries starting from the

Algorithm 11 Com-LJT

```
procedure COM-LJT(Model G, Query terms and types {(Q_k, t_k)}_{k=1}^m, Evidence E)
    Construct an FO jtree J for G
    Enter E into J
    Pass messages on J                                          ▷ LVE as subroutine
    for each (Q_k, t_k) ∈ {(Q_k, t_k)}_{k=1}^m do
        if t_k = MPE then                                       ▷ Q_k = ∅
            J-MPE-LJT(J)                                        ▷ Output or store result
        else
            Find a subtree J' s.t. Q_k ⊆ rv(J')
            if t_k = MAP ∧ Q_k = rv(J') then
                J-MPE-LJT(J')                                   ▷ Output or store result
            else
                Extract a submodel G' from J'
                if t_k = MAP then                               ▷ Q_k ⊂ rv(J')
                    MAP-LVE(G', Q_k, ∅)                         ▷ Output or store result
                else
                    LVE(G', Q_k, ∅)                             ▷ Output or store result
```

original model, we can set up an LJT algorithm that combines the different LVE and LJT versions in one framework, called Com-LJT. Com-LJT constructs an FO jtree for the input model, enters evidence, and passes messages with LVE. Afterwards, Com-LJT can answer all probability and MAP queries. If an MAP query is over complete subtrees, Com-LJT can use J-MPE-LJT. If an MPE query occurs, Com-LJT does a message pass with MPE-LVE, which is identical to passing a complete FO jtree to J-MPE-LJT.

Algorithm 11 shows the combined algorithm with a model G, a set of query terms paired with its query type $\{(\mathbf{Q}_k, t_k)\}_{k=1}^m$, $t_k \in \{\text{MPE}, \text{MAP}, P\}$, and evidence $\mathbf{E}$ as input. If $t_k = P$, Com-LJT answers $P(\mathbf{Q}_k|\mathbf{E})$. If $t_k = \text{MAP}$, Com-LJT answers $\text{MAP}(\mathbf{Q}_k|\mathbf{E})$. If $t_k = \text{MPE}$, which means $\mathbf{Q}_k = \emptyset$, Com-LJT answers $\text{MPE}(\mathbf{E})$. Com-LJT constructs an FO jtree J for G, enters evidence, and passes messages like any other LJT version. Then, Com-LJT goes through the set of queries. If $t_k = \text{MPE}$, Com-LJT uses J-MPE-LJT with J as input, which saves the effort of building an FO jtree anew. J-MPE-LJT computes a new set of messages to answer the MPE query. However, Com-LJT keeps the original messages in J in its query loop to use them for other queries. Caching the messages takes up more memory but means faster answering of the next query. If $t_k \neq \text{MPE}$, the query is either an MAP query or a probability query with a set of query terms $\mathbf{Q}_k$. For both query types, Com-LJT finds a subtree J' covering $\mathbf{Q}_k$. If $t_k = \text{MAP}$ and the query terms cover a subtree J' of J, i.e., $rv(J') = \mathbf{Q}_k$, Com-LJT uses J-MPE-LJT with J' as input, which reuses the messages from outside J'. The messages contain the result of summing out non-query PRVs. J-MPE-LJT maxes out the query PRVs to solve the

posed MAP problem. If $rv(J') \subset \mathbf{Q}_k$ or $t_k = P$, Com-LJT extracts a corresponding submodel G' from J' and, depending on the query type, uses either MAP-LVE or LVE to answer the query, reusing the messages from outside J'. To illustrate Com-LJT, consider the following example with a set of queries for G_{ex}.

Example 10.3.1 (Com-LJT)**.** Assume the following queries with evidence $sick(eve)$:

(1) $(\{Treat(X, M), Sick(X), Epid\}_{|\top}, \text{MAP})$

(2) $(\{Sick(X), Epid\}_{|\top}, P)$

(3) $(\emptyset, \text{MPE})$

(4) $(\{Treat(eve, injection)\}, P)$

(5) $(\{Treat(eve, injection), Travel(eve)\}, P)$

Com-LJT constructs an FO jtree for G_{ex} and enters evidence as in the previous examples. Messages flow from parclusters $\mathbf{C}_1$ and $\mathbf{C}_3$ to parcluster $\mathbf{C}_2$ and back as shown in Example 4.3.4. The FO jtree is now prepared to answer any query with evidence $sick(eve)$. For pair (1) with type MAP, Com-LJT finds a subtree J' which consists of $\mathbf{C}_3$. As the query terms and the PRVs in $\mathbf{C}_3$ are the same, Com-LJT uses J-MPE-LJT on the subtree. As J' consists of only one parcluster, message passing on J' terminates immediately and MPE-LVE maxes out the PRVs $Treat(X, M)$, $Treat(eve, M)$, $Sick(X)$, and $Epid$, X restricted to *alice* and *bob*.

Pair (2) with type P represents a parameterised conjunctive query. Com-LJT uses $\mathbf{C}_2$ to answer the query, providing LVE with the local model and messages of $\mathbf{C}_2$ as a model and $\{Sick(X), Epid\}_{|\top}$ as the query terms. LVE sums out $Treat(X, M)$ and $Treat(eve, M)$, counts X, and normalises the result for the terms $\#_X[Sick(X)]$ and $Epid$. For pair (3) with type MPE, Com-LJT takes J and tasks J-MPE-LJT with calculating an MPE solution, which proceeds with message passing as in Example 10.1.8. After having procured the MPE solution, Com-LJT proceeds with pair (4) with type P, which references a simple probability query with a single query term. The subtree consists of $\mathbf{C}_3$. Com-LJT provides LVE with the local model and messages of $\mathbf{C}_3$ as a model and $\{Treat(eve, injection)\}$ as the query term, which proceeds as in Example 4.3.5. For pair (5), Com-LJT compiles a subtree of $\mathbf{C}_2$ and $\mathbf{C}_3$, extracts a submodel of local models and outside message m_{12}, and uses LVE to compute an answer as in Example 8.1.2.

The example shows how Com-LJT constructs an FO jtree once and reuses it for a variety of queries. Additionally, the messages calculated with LVE are usable for probability as well as MAP queries, avoiding repeated computations of identical sum-out operations. In summary, Com-LJT combines lifting and clustering of relational models into one algorithm for exact, efficient repeated inference for a query language that allows for assignment and probability queries alike. Next, we look at correctness and complexity results to further grasp lifted inference w.r.t. assignment queries.

10.4 Theoretical Discussion

We consider soundness, completeness, and complexity of the various algorithms proposed in this section, analysing liftable MAP queries. Based on liftable inputs, we conduct a complexity analysis of the MPE and MAP versions of LVE and LJT.

Soundness We start with soundness for MPE-LVE and continue with MPE-LJT, followed by the MAP versions, before ending with Com-LJT. The proofs rely on the soundness of the LVE operator suite (Taghipour *et al.*, 2013c) and LJT (shown in Chapter 5).

Theorem 10.4.1. *Let G be a model and $\mathbf{E}$ be evidence. Then, MPE-LVE is sound, i.e., computes an MPE for $rv(G)\backslash rv(\mathbf{E})$ equivalent to an MPE computed for $gr(rv(G)\backslash rv(\mathbf{E}))$.*

Proof. For a set of interchangeable randvars, the arg max assignment is identical given each possible valuation of the remaining randvars. As instances of a PRV are interchangeable, assigning the same value for its instances, as in the *max-out* operator, is correct. Storing the assignments in histograms based on counts is a different representation of the same information. Thus, the postcondition in *max-out* holds. Let us consider the other operators. As we do not change computations w.r.t. potentials, these parts of an operator are still sound. We need to consider assignments stored in each parfactor. The operator *absorb* is identical to the LVE operator of the same name applied to parfactors without maxed out PRVs, which eliminates $\mathbf{E}$ from G through appropriate absorption. The transforming operators *split*, *expand*, *count-convert*, and *ground* do not manipulate assignments given their preconditions and thus, work as before with the postcondition holding. The operators *multiply* and *count-convert* manipulate assignments to varying degree. Operator *multiply* concatenates assignments of disjoint sets of maxed out PRVs, thus, keeping the assignments consistent. As argued for *count-convert*, summing over corresponding histograms combines assignments that the instances of a PRV map to, which are materialised by counting a CRV. Therefore, the transformed model is equivalent to the model before applying a count conversion. Generalised counting operators keep assignments consistent with the same argument. In summary, all MPE-LVE operators are sound with the individual postconditions holding.

Based on this suite of sound MPE-LVE operators, MPE-LVE translates into the application of a sequence of sound MPE-LVE operators to G and $\mathbf{E}$ to max out $rv(G) \setminus rv(\mathbf{E})$. The final parfactor contains assignments for $rv(G) \setminus rv(\mathbf{E})$, which come from applying *max-out*. The postcondition of *max-out* ensures that each assignment is equivalent to an assignment for the grounded instances. As replacing $\sum$ with arg max yields a correct MPE solution in the ground case (Dechter, 1999) and MPE-LVE computes an MPE solution equivalent to one computed on a ground level, MPE-LVE is sound. □

Theorem 10.4.2. *Let G be a model and $\mathbf{E}$ be evidence. MPE-LJT is sound, i.e., computes a correct answer to the query* MPE($\mathbf{E}$).

Proof. The first two steps of MPE-LJT coincide with LJT. Thus, MPE-LJT constructs a valid FO jtree J for G. A valid FO jtree allows for local computations (Shenoy and Shafer, 1990). Evidence entering and message passing consist of applying sound MPE-LVE operators, leading to correct results in local models and messages. The message passing scheme combines all model parts at some parcluster of J, where the final application of MPE-LVE operators eliminates the remaining PRVs. The result is then a parfactor, which holds correct assignments for $rv(G) \setminus rv(\mathbf{E})$. □

Next, we consider the MAP versions of LVE and LJT, which apply both LVE and MPE-LVE operators to answer MAP queries, before analysing J-MPE-LJT for subtrees of an FO jtree and Com-LJT, which combines algorithms for different types of queries.

Theorem 10.4.3. *Let G be a model, $\mathbf{U}_{|C}$ be query terms, and $\mathbf{E}$ be evidence. Then, MAP-LVE is sound, i.e., computes a most probable assignment for $\mathbf{U}_{|C}$ that is equivalent to a most probable assignment computed for $gr(\mathbf{U}_{|C})$.*

Proof. In the ground case, computing a most probable assignment for a subset of model randvars $\mathbf{U}_{|C}$ requires summing out the randvars not in $\mathbf{U}_{|C}$ before maxing out $\mathbf{U}_{|C}$. MAP-LVE implements the computation by using LVE operators to sum out $\mathbf{T} = rv(G) \setminus \mathbf{U}_{|C} \setminus rv(\mathbf{E})$ and then using MPE-LVE operators to max out $\mathbf{U}_{|C}$. LVE computes a result equivalent to computing a result on the ground level using its LVE operators. Thus, given the LVE operators are sound, the result of summing out $\mathbf{T}$ in G is equivalent to the result of summing out on the ground level. Next, MAP-LVE uses MPE-LVE operators to max out $\mathbf{U}_{|C}$. MPE-LVE computes a result equivalent to computing a result on the ground level using its MPE-LVE operators. Again, given the MPE-LVE operators are sound, MAP-LVE computes a result equivalent to computing a result on the ground level when maxing out $\mathbf{U}_{|C}$ in G'. The final parfactor contains a most probable assignment for $\mathbf{U}_{|C}$ that is equivalent to one computed on the ground level for $gr(\mathbf{U}_{|C})$. □

Theorem 10.4.4. *Let G be a model, $\{\mathbf{U}_{|C,k}\}_{k=1}^{m}$ a set of query terms, and $\mathbf{E}$ evidence. Then, MAP-LJT is sound, i.e., computes a correct answer to each query* $\mathrm{MAP}(\mathbf{U}_{|C,k}|\mathbf{E})$.

Proof. Given a sound LJT, MAP-LJT constructs a valid FO jtree J for G, allowing for local computations (Shenoy and Shafer, 1990) during evidence entering and message passing. Evidence entering and message passing consist of applying sound LVE operators, leading to correct results in local models and messages. Extracting the submodel G' for query terms $\mathbf{U}_{|C,k}$ from the local models and outside messages, LJT combines all necessary parfactors without duplicates (cf. the proof of Thm. 8.2.1 regarding soundness of LJT for conjunctive queries). MAP-LJT then presents MAP-LVE with G' as model and $\mathbf{U}_{|C}$ as query terms, which yields a correct result as shown in the proof of Thm. 10.4.3. Therefore, MAP-LJT computes a correct answer to each given MAP query. □

Theorem 10.4.5. *Let G be a model with an FO jtree J with evidence $\mathbf{E}$ entered and messages passed and $\mathbf{U}_{|C}$ be query terms with $\mathbf{U}_{|C} = rv(J')$ for a subtree J' of J. Then, J-MPE-LJT is sound, i.e., computes a correct answer to the query* $\mathrm{MAP}(\mathbf{U}_{|C}|\mathbf{E})$.

Proof. Given J is a valid FO jtree and LVE Operators are sound, entering evidence and passing messages leads to correct representations of all model parfactors as well as evidence in the messages and local models of each parcluster in J. Computing an MAP query $\mathrm{MAP}(\mathbf{U}_{|C}|\mathbf{E})$ where $\mathbf{U}_{|C} = rv(J')$ for some subtree J' of J means summing out $\mathbf{T} = rv(G) \setminus \mathbf{U}_{|C} \setminus rv(\mathbf{E})$, which all occur in parclusters outside of J', while all PRVs in $\mathbf{U}_{|C} \setminus rv(\mathbf{E})$ need to be maxed out. Thus, J-MPE-LJT is able to compute $\mathrm{MAP}(\mathbf{U}_{|C}|\mathbf{E})$ by using J' only. Using the messages from outside J', $\mathbf{T}$ has already been summed out. On J', J-MPE-LJT follows MPE-LJT and sends messages using MPE-LVE operators to compute a most probable assignment for all PRVs in J'. The remaining steps of the proof follow the same argument as for the soundness of MPE-LJT. J-MPE-LJT passes messages on a subtree of a valid FO jtree, which allows for local computations (Shenoy and Shafer, 1990). Given the MPE-LVE operators are sound, J-MPE-LJT computes correct messages by applying sound operators. At some parcluster in J' with MPE messages from each neighbour, J-MPE-LJT maxes out the remaining PRVs, which yields a correct assignment for $\mathbf{U}_{|C}$. □

Theorem 10.4.6. *Let G be a model, $\{(\mathbf{Q}_{|C,k}, t_k)\}_{k=1}^{m}$ a set of queries, and $\mathbf{E}$ evidence. Then, Com-LJT is sound, i.e., computes a correct answer for each $(\mathbf{Q}_{|C,k}, t_k)$.*

Proof. Next to LVE and LJT, we assume MPE-LVE, MAP-LVE, and J-MPE-LJT to be sound based on Thms. 10.4.1, 10.4.3 and 10.4.5. As construction, evidence entering, message passing coincide with LJT and LJT is sound given LVE is sound, Com-LJT constructs a valid FO jtree J, the basis for local computations (Shenoy and Shafer, 1990), and enters evidence and passes messages, leading to correct local models and messages. Com-LJT can process MPE queries ($t_k = \mathrm{MPE}$), MAP queries ($t_k = \mathrm{MAP}$), and probability queries ($t_k = P$). Given an MPE query, Com-LJT uses J-MPE-LJT on J, which produces a correct assignment for $rv(J) \setminus rv(\mathbf{E}) = rv(G) \setminus rv(\mathbf{E})$. Given an MAP query over $\mathbf{Q}_{|C,k}$ where there exists a subtree J' such that $rv(J') = \mathbf{Q}_{|C,k}$, Com-LJT uses J-MPE-LJT on J', which produces a correct assignment for $rv(J') \setminus rv(\mathbf{E}) = \mathbf{Q}_{|C,k}$. Given an MAP query over $\mathbf{Q}_{|C,k}$ where the subtree J' contains more PRVs than in $\mathbf{Q}_{|C,k}$, Com-LJT uses MAP-LVE to compute a correct assignment for $\mathbf{Q}_{|C,k} \subset rv(J')$. Given a probability query with query terms $\mathbf{Q}_{|C,k}$, Com-LJT uses LVE for parameterised conjunctive queries to compute a correct result. In conclusion, Com-LJT uses sound algorithms to provide correct results to each query. □

Completeness This analysis has two parts, one for MPE queries and one for MAP queries. We start with MPE queries, which do not concern specific query terms, only models and evidence. The results for MPE-LVE and MPE-LJT coincide with the results

of LVE and LJT for query class $\mathcal{Q}$ of single ground query terms and model classes $\mathcal{M}^{2lv}$ of two-logvar models and $\mathcal{M}^{prv1}$ of models with one-logvar PRVs (given liftable evidence).

Theorem 10.4.7. *MPE-LVE and MPE-LJT are complete for the model classes $\mathcal{M}^{2lv}$ and $\mathcal{M}^{prv1}$.*

Proof. The completeness results of LVE and LJT hold for the query class $\mathcal{Q}$, which also includes computing a result for an empty query. An empty query requires no shattering on query terms and an elimination of all model PRVs. Answering an empty query corresponds to explicitly computing the normalisation constant Z of a model. LVE and LJT are able to compute a domain-lifted solution to such an empty query for any model of the two classes. Computing an answer to an MPE query also requires eliminating all model PRVs and as such is an analogous task to the empty query for LVE and LJT. As both the MPE versions of LVE and LJT follow the same workflow as before, we have to show that the changes made to the operators keep the algorithms domain-lifted. The steps performed in the operators *max-out* and *sum-out* are analogous to each other, sifting through the input parfactor and performing an arithmetic operation on potentials (addition/maximisation). Thus, this change does not lead to a change in complexity. At the same time of performing a maximisation, the *max-out* operator also constructs a histogram corresponding to the arg max assignment. Constructing such a histogram can be thought of as performing a count conversion for the PRV that is maxed out. The complexity of a count conversion is polynomial in the domain sizes of the logvars involved (Taghipour, 2013). Thus, the overall complexity of *max-out* remains polynomial in the domain sizes of the logvars involved.

The operator *multiply* sifts through its input parfactors as before. Additionally, it concatenates histogram sequences, which is a constant operation w.r.t. domain sizes. The operator *count-convert* performs for each potential calculation an analogous histogram calculation, summing and multiplying counts instead of multiplying and exponentiating potentials. The size of a histogram, i.e., the number of tuples in it, depends on the range size of the underlying PRV. Thus, the additional work load is bounded by the same complexity as the original count conversion. The remaining operators do not have a change in operation, with histograms being copied along, which is an operation independent of domain sizes. The remaining part to show is that the preconditions of the operators do not lead to operations depending exponentially on domain sizes, i.e., do not lead to groundings. The operators *count-convert* and *expand* have additional preconditions, which as argued during their introduction do not limit MPE-LVE in its operations. Thus, answering an MPE query with MPE-LVE corresponds to answering an empty query with LVE with a runtime complexity polynomial in domain sizes and the arguments for completeness of LVE transfer to MPE-LVE w.r.t. the two classes.

The results for MPE-LVE extend to MPE-LJT: As fusion ensures that message passing does not induce any additional groundings, the completeness results of MPE-LVE hold also for MPE-LJT (compare Chapter 5). □

The complexity of the MPE-LVE operators compared to the LVE operators remains the same. A more detailed complexity analysis follows in the complexity paragraph. The second part of this completeness analysis concerns MAP queries. Due to the non-commutativity of summing out and maxing out, answering MAP queries can quickly lead to prohibitively large intermediate results. In a paradox turn, an algorithm can still be domain-lifted and complete as long as runtimes are polynomial in the domain sizes of the model logvars. But, logvars in query terms can cause groundings, making MAP-LVE and MAP-LJT not complete for two-logvar models.

Theorem 10.4.8. *MAP-LVE and MAP-LJT are not complete for the model class $\mathcal{M}^{2lv}$ and the class of all possible MAP queries $\mathcal{MAP}$.*

Proof. Assume both algorithms are complete for all possible two-logvar models. Consider a parfactor $\phi(Q(X,Y), R(X), S(Y))$ and an MAP query $\mathrm{MAP}(\{Q(X,Y)\})$ without evidence. MAP-LVE has to sum out $R(X)$ and $S(Y)$ before maxing out $Q(X,Y)$. Neither summing out $R(X)$ nor $S(Y)$ is possible as neither contains both logvars. As both logvars are not countable, MAP-LVE has to ground one of them, which means the algorithm run is no longer polynomial in the domain size of the grounded logvar. MAP-LJT exhibits the same behaviour using MAP-LVE as a subroutine. Therefore, MAP-LVE and MAP-LJT are not complete w.r.t. the class of two-logvar models. □

An MAP query poses a problem to MAP-LVE and MAP-LJT that is similar to the problem that parameterised conjunctive queries pose to LVE and LJT w.r.t. constraints and logvars. Therefore, there exists a query class $\mathcal{MAP}^{lift}$ that corresponds to $\mathcal{PCQ}^{lift}$.

Definition 10.4.1. Query class $\mathcal{MAP}^{lift}$ refers to MAP query terms with one logvar per PRV and one set of constants per logvar.

Theorem 10.4.9. *MAP-LVE and MAP-LJT are complete for the query class $\mathcal{MAP}^{lift}$ and the model classes $\mathcal{M}^{2lv}$ and $\mathcal{M}^{prv1}$.*

Proof. To show that LVE is complete for the query class $\mathcal{MAP}^{lift}$, we need to show that LVE is domain-lifted for all queries in $\mathcal{MAP}^{lift}$. The proof combines arguments from Thms. 8.2.3 and 9.3.3 about completeness of liftable (parameterised) conjunctive queries.

Given a query $\mathbf{Q} \in \mathcal{MAP}^{lift}$, LVE shatters the input model on $\mathbf{Q}$. The splitting procedure defined by Taghipour *et al.* (2013c) splits any parfactor into two parts at most given a PRV. With one set of constants per logvar in $\mathbf{Q}$, a parfactor is split into two parfactors at most for each query term covered by its arguments. In a parfactor that covers some query terms $\mathbf{Q}' \subseteq \mathbf{Q}$, query terms in $\mathbf{Q}'$ with the same logvar or constant lead to one split. Query terms in $\mathbf{Q}'$ with different logvars lead to a split for each logvar, resulting in 2^l splits at most, where l is the number of logvars in $\mathbf{Q}'$. With a bounded number of splits only depending on the number of logvars (not on the domains), a liftable MAP query does not lead to groundings during shattering.

For models in $\mathcal{M}^{2lv}$, LVE eliminates all non-query terms with two logvars as $\mathcal{MAP}^{lift}$ only contains query terms with one logvar. Eliminating two-logvar PRVs either uses lifted summing out or group inversion (Taghipour *et al.*, 2013d). Afterwards, a model contains only one-logvar PRVs without any counted logvars and ground randvars, which is a model of $\mathcal{M}^{prv1}$. Following the argument from the proof of Thm. 9.3.3, MAP-LVE is able to count all remaining logvars and eliminate all non-query terms. The remaining CRVs refer to query terms, which MAP-LVE maxes out, yielding an assignment for $\mathbf{Q}$. Thus, MAP-LVE is complete for $\mathcal{M}^{2lv}$ and $\mathcal{M}^{prv1}$. As MAP-LJT uses MAP-LVE to answer a particular query, completeness for MAP-LJT immediately follows. □

Similar to parameterised queries, more liftable queries exist in addition to the queries in $\mathcal{MAP}^{lift}$. E.g., if query terms contain fewer logvars than the non-query terms, no groundings occur, which compares to Proposition 9.3.4. If there is a query term $Q(X, Y)$, which appears in a parfactor together with $R(X)$ and $S(Y)$, then groundings occur (see Proposition 9.3.2). But, in contrast to parameterised queries, the logvars of MAP query terms may be subsets of each other as MAP-LVE does not need to induce a joint distribution. In summary, liftable MAP queries exist and under certain conditions, one can inspect a model to determine a query to be liftable or grounding. As intermediate parfactors still may get prohibitively large, the upcoming complexity analysis characterises queries with the same tree width as probability and MPE queries.

Complexity We analyse the runtime complexity of the MPE and MAP versions of LVE and LJT. We assume a model and evidence that permit a lifted solution.

The complexity results of MPE-LVE are based on two observations, (i) the MPE-LVE operators stay within the runtime complexity of their LVE counterparts and (ii) an MPE query corresponds to an empty probability query, cf. the proof of Thm. 10.4.7 for both. As mentioned before, the runtime complexity of LVE and of message passing in LJT are identical given a a model G with an FO jtree J, i.e., $O(n_J \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ where n_J is the number of parclusters in J, $(w_g, w_{\#})$ is the lifted width of J, n is the largest domain size among $lv(G)$, r the largest range size among the PRVs in J, $n_{\#}$ is the largest domain size of the counted logvars, and $r_{\#}$ is the largest range size among the PRVs in the CRVs. Given the two observations from above, we can transfer the runtime complexity of LVE to MPE-LVE and formulate the following theorem.

Theorem 10.4.10. *The runtime complexity of MPE-LVE is equal to the runtime complexity of LVE for single ground query terms, lying in*

$$O(n_J \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{10.4}$$

Computing an answer to an empty query with LVE and an answer to an MPE query requires the same number of eliminations, carried out by operators of the same complexity. Thus, the complexity of MPE-LVE and LVE coincide.

For MPE-LJT, we need to consider its individual steps. The runtime complexity of construction is negligible. The runtime complexity of evidence entering is characterised by Expression (10.4). Message passing has only an inward pass, meaning, $n_J - 1$ message calculations, each in parcluster complexity. After the inward pass, MPE-LVE eliminates the PRVs at some final parcluster, which also lies in the parcluster complexity. Thus, Expression (10.4) also characterises the overall runtime complexity of MPE-LJT.

Theorem 10.4.11. *The runtime complexity of MPE-LJT is equal to the runtime complexity of MPE-LVE and therefore, LVE, i.e.,* $O(n_J \cdot \log_2 n \cdot r^{w_g} \cdot n_\#^{w_\# \cdot r_\#})$ *(10.4).*

The effort for evidence entering and message passing directly coincides with the operations MPE-LVE performs. Only construction produces overhead within MPE-LJT. But, given a model and a specific set of evidence, only one MPE query is possible. Thus, MPE-LJT cannot answer multiple queries and trade off its overhead for construction. A speed-up for answering an MPE query is still possible due to a reduced search space for possible operations in parclusters. Given a string of evidence sets, MPE-LJT might offset its construction effort over answering MPE queries for each evidence set.

We now move to analysing the runtime complexity of the MAP algorithms. MAP queries are similar to parameterised conjunctive queries, requiring a subtree for the query terms. LJT has a query answering complexity for liftable parameterised conjunctive queries of $O(n'_J \cdot \log_2 n \cdot r^{w_g} \cdot n_\#^{w_\# \cdot r_\#})$ where n'_J stands for the number of parclusters in the subtree J' required for answering a parameterised conjunctive query. But since summing out and maxing out are not commutative, the runtime complexity even of liftable MAP queries does not adhere to $(w_g, w_\#)$. In a worst-case scenario, MAP-LVE has to multiply all parfactors in J' into one parfactor over all PRVs in J' to sum out non-query terms, which would lead n'_J to become a part of r^{w_g}, i.e., $r^{n'_J \cdot w_g}$. Given an FO jtree J, we are able to determine queries that are liftable and have intermediate results bounded by the lifted width of J. For LVE, an FO dtree and its lifted width take over the role of an FO jtree.

Proposition 10.4.1. *An MAP query over query terms* $\mathbf{Q}$ *for a model* G *with FO jtree* J *permits a domain-lifted run iff* $\mathbf{Q} = rv(J')$ *for a subtree* J' *of* J*. We call such a query a bounded MAP query.*

Such a bounded MAP query permits characterising the complexity of MAP-LVE and MAP-LJT in terms of the lifted width of the corresponding FO jtree. Given an FO jtree J, an MAP query over all PRVs of a subtree of J guarantees that all PRVs that do no occur in the subtree are eliminable by summing out. The non-query PRVs appear in the remaining parclusters of J given the subtree. Assume that J' has n'_J parclusters, which means $n_J - n'_J$ parclusters remain outside of J'. Eliminating the non-query PRVs then has a complexity of

$$(n_J - n'_J) \cdot O(\log_2 n \cdot r^{w_g} \cdot n_\#^{w_\# \cdot r_\#}), \tag{10.5}$$

which is $(n_J - n'_J)$ times the parcluster complexity. Within the subtree, MPE-LVE operators are applied at each parcluster of the subtree, which leads to a complexity of

$$n'_J \cdot O(\log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}). \tag{10.6}$$

Expressions (10.5) and (10.6) together lead to the following runtime complexity of MAP-LVE given a liftable bounded MAP query.

Theorem 10.4.12. *For a bounded MAP query, MAP-LVE has a runtime complexity of*

$$O(n_J \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}) \tag{10.7}$$

which coincides with the complexity of MPE-LVE and LVE.

In contrast to MPE-LJT, MAP-LJT is able to answer a set of queries given an evidence set. MAP-LJT performs a full message pass, which has a complexity as in Expression (10.4). Given an MAP query and a corresponding subtree, the query answering step basically follows J-MPE-LJT, maxing out the query terms in the subtree. The complexity of answering an MAP query falls under $O(n'_J \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}})$ again, which leads to the following overall complexity of MAP-LVE.

Theorem 10.4.13. *Given an FO jtree J with n_J parclusters for a model G and a set of m liftable bounded MAP queries, which require a subtree with a maximum of n'_J parclusters, the complexity of MAP-LJT is*

$$O((n_J + m \cdot n'_J) \cdot \log_2 n \cdot r^{w_g} \cdot n_{\#}^{w_{\#} \cdot r_{\#}}), \tag{10.8}$$

The runtime complexity of MAP-LJT coincides with the complexity of LJT for parameterised (conjunctive) queries. MAP-LJT is able to offset its overhead of construction and message passing by answering multiple queries, which are expected to require small subtrees of a given FO jtree. Com-LJT also falls under the complexity given in Expression (10.8), in which $n'_J = n_J$ for MPE queries.

The next section presents an empirical evaluation to show that it is practically achievable to implement the given algorithms leading to runtimes for MPE queries and bounded MAP queries that lie close to the runtimes of probability queries.

10.5 Empirical Evaluation

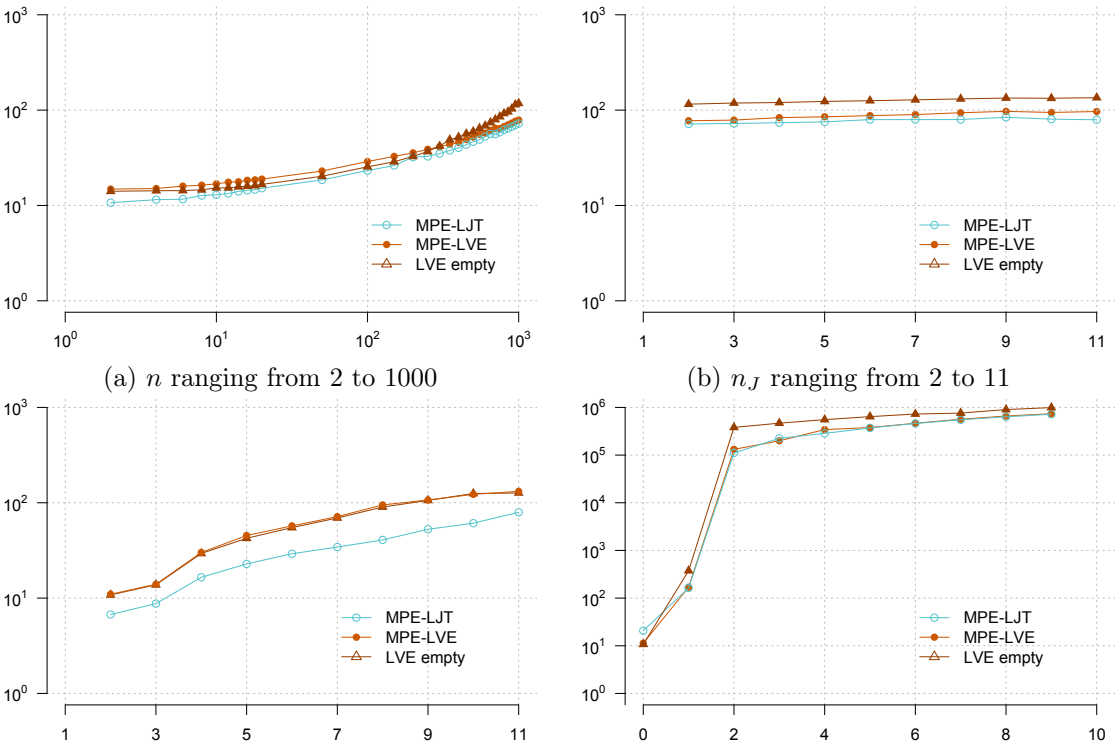
This section presents an empirical evaluation of the MPE and MAP versions of LJT and LVE. FOKC is not a part of this evaluation as its implementation (available at `https://dtai.cs.kuleuven.be/software/wfomc`) does not allow for assignment queries. For the MPE and MAP versions, we have implemented prototypes based on the LJT prototype of the previous evaluations and the LVE implementation by Taghipour (available at

`https://dtai.cs.kuleuven.be/software/gcfove`). We test the implementations w.r.t. the parameters that influence the complexity of LVE and LJT, namely, (i) largest domain size n, (ii) number of parclusters n_J, and (iii) lifted width $(w_g, w_\#)$ in which w_g is the ground width and $w_\#$ the counting width. The basic input model is G_{ex} with boolean ranges, $n = 1000$, which we use for each logvar, $n_J = 3$, $w_g = 3$, and $w_\# = 1$. Evidence is empty as to not add noise w.r.t. the parameters. When varying one parameter, the remaining parameters appear fixed. Additionally, for varying n_J, each new parcluster has a lifted width of $(3, 0)$. When varying w_g, each of the $n_J = 3$ parclusters has a ground width of w_g. When varying $w_\#$, each of the $n_J = 3$ parclusters has a counting width of $w_\#$. Doing so ensures that each message is affected and that each possible query leads to a parcluster of worst case size. The overall number of parfactors $|G|$ varies between 3 and 70 with $|gr(G)|$ ranging from 11 to 24,000,000,001. We present the empirical evaluation in two parts, (i) runtimes for MPE calculations and (ii) runtimes for MAP calculations.

MPE Evaluation This first part of the evaluation investigates the tradeoff still possible with MPE-LJT compared to MPE-LVE as well as the effect of maxing out and managing assignments on runtimes. Handling assignments adds work for the MPE versions, but, on average, determining a maximum value takes less time than computing a sum. As a baseline, we use LVE answering an empty query $P(.)$, which means computing the normalisation constant Z. An empty query complements an MPE query as for each, an algorithm eliminates all model PRVs without splitting off query terms.

Figure 10.2 shows runtimes in milliseconds [ms] for MPE-LJT (hollow circle) and MPE-LVE (filled circle) as well as LVE (filled triangle) on a log-scaled y-axis. In each subfigure, one of the parameters domain size n, number of parclusters n_J, ground width w_g, and counting width $w_\#$ varies. In Fig. 10.2a, n varies from 2 to 1000 on a log-scaled x-axis. The runtimes of all three algorithms increase with rising domain sizes as expected from the complexity analysis. Though, only marginally, MPE-LJT has the shortest runtimes over all n. With larger domain sizes, LVE answering an empty query is the slowest program, though, again only slightly. In Fig. 10.2b, n_J varies between 2 and 11. All versions exhibit a linear increase in runtimes. The MPE versions have shorter runtimes from the beginning. But, only a fraction of time separates MPE-LJT and MPE-LVE. MPE-LVE runtimes increase slightly more than MPE-LJT runtimes. For $n_J = 2$, the LJT version takes 0.93 of the runtime of MPE-LVE. For $n_J = 11$, this factor is 0.82. In Fig. 10.2c, w_g varies from 2 to 11. Runtimes of LVE and MPE-LVE coincide over all models while MPE-LJT is faster by a factor between 0.43 and 0.63. In Fig. 10.2d, $w_\#$ lies between 0 and 9. Here, runtimes of MPE-LVE and MPE-LJT more or less coincide. LVE takes slightly longer to answer an empty query. Runtimes surge when reaching a counting width of 2 after which the increase flattens out.

Overall, the MPE versions exhibit behaviour as expected from the complexity analysis. Computing answers to MPE queries does not have a negative impact on runtimes, more



(a) n ranging from 2 to 1000

(b) n_J ranging from 2 to 11

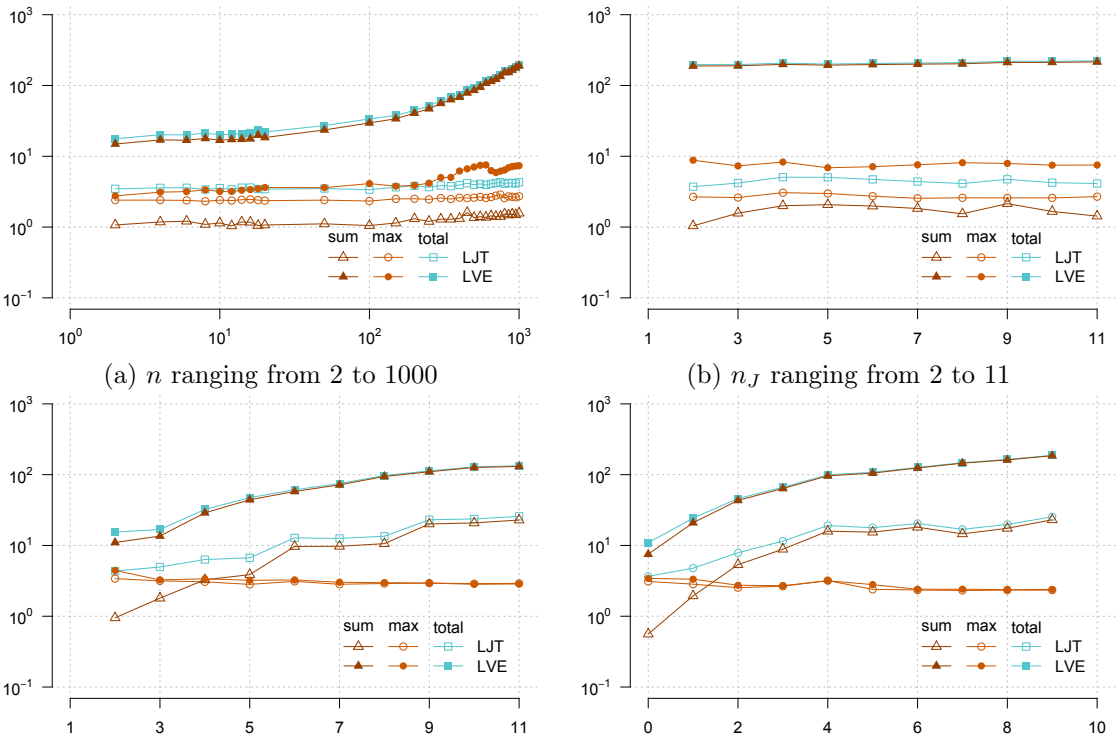
(c) w_g ranging from 2 to 11

(d) $w_\#$ ranging from 0 to 9

Figure 10.2: Runtimes [ms] for MPE-LJT and MPE-LVE with LVE answering an empty query as a baseline; if not stated, domain size $n = 1000$, number of parclusters $n_J = 3$, ground width $w_g = 3$, counting width $w_\# = 1$

to the contrary. As MPE-LJT always has to perform an MPE message pass to compute an answer, MPE-LJT runtimes do not exhibit a strong lead compared to MPE-LVE. Still, MPE-LJT has a reduced runtime compared to MPE-LVE using the underlying helper structure. Evidence handling, though not part of this evaluation, is not affected by assignment handling as assignments are empty at that point in the algorithm. The effort for entering evidence in MPE-LJT and handling evidence in MPE-LVE is identical in terms of applications of the absorption operator. MPE-LJT and MPE-LVE then handle the resulting model after absorption, which follows the same behaviour investigated here.

MAP Evaluation This second part of the evaluation takes a look at how runtimes behave w.r.t. domain size, number of parclusters, and lifted width given a bounded MAP query. Additionally, we compare runtimes for an MAP query causing a grounding, an MAP query requiring an extra count conversion, and a bounded MAP query. Each query consists of two (parameterised) query terms with a $\top$ constraint, contained within one



(a) n ranging from 2 to 1000

(b) n_J ranging from 2 to 11

(c) w_g ranging from 2 to 11

(d) $w_\#$ ranging from 0 to 9, $n = 10$

Figure 10.3: Runtimes [ms] for MAP-LJT and MAP-LVE with LJT and LVE answering an empty query as a baseline; if not stated, domain size $n = 1000$, number of parclusters $n_J = 3$, ground width $w_g = 3$, counting width $w_\# = 1$

parcluster. We do not consider subtree size for MAP queries as we expect a behaviour of MAP-LJT similar to LJT for conjunctive queries as investigated in Section 8.3.

Figure 10.3 shows runtimes in milliseconds [ms] for MAP-LJT (hollow symbols) and MAP-LVE (filled symbols) on a log-scaled y-axis for an MAP query over a separator. In each subfigure, one of the parameters n, n_J, w_g, and $w_\#$ varies. Three runtimes are given per program, total runtime (squares), runtime for summing out ("sum", triangles), and runtime for maxing out ("max", circles). Given that MAP-LJT answers the query on a smaller submodel than MAP-LVE, we expect the MAP-LJT runtimes to be shorter than the MAP-LVE runtimes for summing out and, consequently, in total.

In Fig. 10.3a, n varies from 2 to 1000 on a log-scaled x-axis. Total runtimes of MAP-LVE exhibit a sharper increase than those of MAP-LJT as the input model for answering the query is larger and the effects of larger domain sizes amplify because of it. In Fig. 10.3b, n_J varies between 2 and 11. Runtimes of MAP-LJT are almost constant as the MAP query can be answered on one parcluster. MAP-LVE runtimes exhibit a

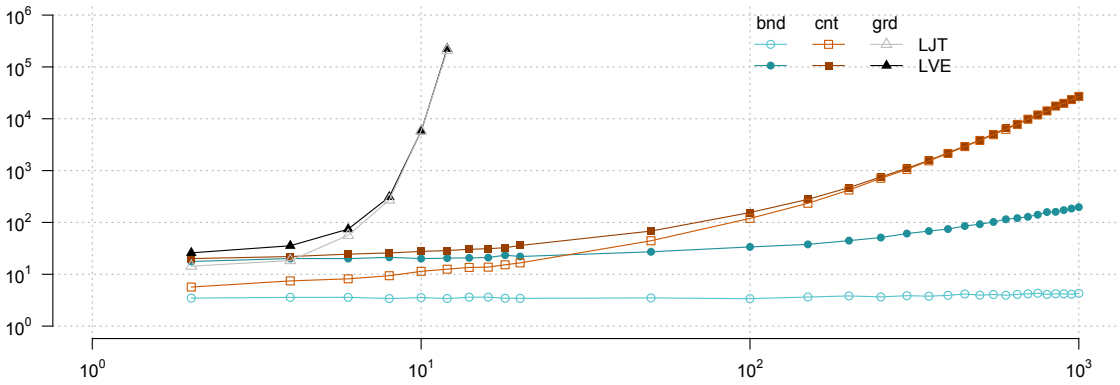


Figure 10.4: Runtimes [ms] for answering a bounded query ("bnd"), a liftable query with enlarged parfactor size ("cnt"), and a grounding query ("grd") for MAP-LJT and MAP-LVE w.r.t. domain size n ranging from 2 to 1000; number of parclusters $n_J = 3$, ground width $w_g = 3$, counting width $w_\# = 1$

small linear increase that is barely noticeable in the log-scaled plot. In both plots, the "sum" runtimes make up a large part of the total runtimes of MAP-LVE as the sum-out part outweighs the max-out part. With MAP-LJT, the "max" part makes up the larger portion of the total runtimes as only one non-query term needs to be summed out before maxing out begins. Overall, MAP-LJT has shorter runtimes, being more than one order of magnitude faster than MAP-LVE, for all n_j tested and $n \geq 100$.

In Fig. 10.3c, w_g varies from 2 to 11 and in Fig. 10.3d, $w_\#$ lies between 0 and 9. For the $w_\#$ runtimes, the logvars are set up with a domain size of $n = 10$ as a domain size of $n = 1000$ leads to memory errors from a counting width of 5 onwards. For both settings, MAP-LVE and MAP-LJT exhibit a similar increase in runtimes, with MAP-LJT needing around 0.2 times the runtime of MAP-LVE. Varying the lifted width still shows that the "sum" part of MAP-LVE dominates the total runtimes of MAP-LVE. With MAP-LJT, the "max" part makes up a larger portion of the total runtime only in the beginning. With a rising lifted width, the query covers a shrinking portion of the parcluster, which means, MAP-LJT has to sum out more terms. Thus, the "sum" part becomes the main influence on the overall runtime, similar to MAP-LVE. With varying lifted width, the runtimes of the "max" part of both programs coincide, which is expected from a theoretical point of view as both algorithms have to perform the same maxing-out operations (the difference lies in the number of summing-out operations due to the different model sizes during query answering). Given a query over a whole parcluster, the summing-out runtimes of MAP-LJT drop to 0 and the maxing-out time makes up the total runtime with a behaviour that is similar to the one investigated during the MPE evaluation.

Figure 10.4 shows the effect of a query on runtimes by comparing total runtimes of three different queries, namely, a bounded query ("bnd", circles) as before, a liftable query

with enlarged parfactor size ("cnt", squares), and a grounding query ("grd", triangles). The figure shows runtimes in milliseconds [ms] for MAP-LJT (hollow symbols) and MAP-LVE (filled symbols) on a log-scaled y-axis. The x-axis shows an increasing domain size n ranging from 2 to 1000 to illustrate the exponential effect of groundings. Given a grounding query, runtimes of both programs surge. Starting with $n = 14$, the grounding query leads both programs to have memory errors. The counting query, which has the effect of requiring two count conversion to be able to sum out a non-query term with fewer logvars, leads both programs to exhibit an increase in runtimes that is not as sharp as the one for the grounding query but still noticeable. MAP-LVE with a bounded query performs one count conversion due to the model, while MAP-LJT with a bounded query performs zero count conversions as the PRVs requiring a count conversion are independent from the parcluster used for query answering. Together, they illustrate how count conversions have a greater impact on runtimes than lifted eliminations.

Overall, both MAP versions exhibit expected behaviour when varying parameters that influence the runtime complexity. With bounded MAP queries, MAP-LJT has decidedly shorter runtimes than MAP-LVE, enabling MAP-LJT to trade off its static overhead. Given smaller models, MAP-LJT needs 3 to 6 bounded queries to trade-off its overhead. With larger models, MAP-LJT needs only 2 queries. Considering Com-LJT as a framework for answering a variety of queries efficiently, liftable (assignment or probability) queries contribute to trading off static overhead and answering a set of queries fast.

10.6 Interim Conclusion: Most Probable Assignments

Queries for most probable assignments represent a new task to LVE and LJT compared to the queries for probabilities or probability distributions in the previous chapters. In this chapter, we introduce MPE and MAP queries as assignment queries. Conceptually, answering a query for a most probable explanation means replacing summing out with maxing out. But, given a parameterised model and a set of LVE operators, the implementation of the replacement is not straight-forward. First, adapting LVE and LJT to answer assignment queries requires a way to store and retrieve assignments that lead to a maximum potential. To this end, we present an extended notion of parfactors that map its arguments to potentials as well as assignments for a sequence of maxed out PRVs. Second, answering assignment queries requires a new operator for maxing out a set of interchangeable randvars as well as adapted operators to handle assignments appropriately. Therefore, we present a complete specification of all MPE-LVE operators including generalised counting operators.

Based on the redefined MPE-LVE operators, MPE-LVE and MPE-LJT are able to efficiently answer MPE queries in probabilistic relational models, leveraging their individual strengths. With MAP-LVE and MAP-LJT, we also present two algorithms to answer MAP queries in probabilistic relational models. In combination with an FO jtree, it

is even possible to characterise MAP queries that allow both algorithms to provide an answer without requiring any groundings or causing larger intermediate results than a comparatively simple MPE query would, an observation we pour into J-MPE-LJT. To the best of our knowledge, the theoretical analysis provides the first analysis of MPE and MAP queries in the lifted inference field in such detail.

An interesting avenue to take with the MAP algorithms specified here is looking at the new lifted inference rule introduced by Sharma *et al.* (2018) and investigate whether it is transferable to parameterised models. The same holds for the optimisation of Apsel and Brafman (2012a) regarding look-ups for uniform assignments. Additionally, one can turn to the lifting rule of domain recursion. Domain recursion allows for lifted runs of models representing transitivity as shown by Kazemi *et al.* (2017). The question is how domain recursion can be used to lift such models also for MPE queries.

With LVE and LJT versions for assignment queries, we have filled a gap in lifted inference algorithms for assignment queries based on VE, allowing for handling assignment queries in a lifted way. By setting up Com-LJT, we have shown how probability, MPE, and MAP queries can exploit an underlying FO jtree to varying degrees. Coming to the end of this chapter and the second part of this dissertation, we have covered Contribution (5a), redefined LVE operators for solving MPE queries, Contribution (5b), LVE and LJT versions for MAP queries, and Contribution (5c), completeness and complexity results for LVE and LJT versions answering MPE and MAP queries. In aiming for a richer query language, we have designed lifted algorithms for queries for marginal distributions as well as MAP assignments allowing for a set of parameterised query terms. The upcoming part focusses on adapting LJT to handling incremental changes in a model as well as allowing for different QA algorithms as a subroutine.

Part III

Further Algorithm Extensions

Chapter 11

Adaptive Inference

A common task in many applications is repeated inference on variations of a model. Variations range from a new set of observed events to updating a probability distribution given observations or adapting a model structure while optimising a model representation. Applications include risk analysis where an MPE is of interest with changing sets of events coming in regularly (Muñoz-González *et al.*, 2017). For learning a model structure given observed data, one approach, called structural expectation-maximisation, alternates between minimally changing a model structure and updating distributions in a model to optimise the representation of the observed data. The approach involves changing a model w.r.t. structure and distributions as well as repeated inference when computing the probability of the observed data in the altered model (Friedman, 1998).

In a naive way, one incorporates the changes in a model or evidence and performs inference. Adaptive inference, however, aims at performing inference more efficiently when changes in a model or evidence occur. Research exists for adaptive inference on propositional models (Delcher *et al.*, 1995; Acar *et al.*, 2008a,b). But, modelling realistic scenarios yields large probabilistic relational models, requiring exact and efficient reasoning about sets of individuals. But, to the best of our knowledge, research for adaptive inference on relational models is limited. Now, changes can also affect the sets of individuals over which one reasons or the sets on which one conditions on. How to handle such incremental changes correctly and efficiently is not obvious. Nath and Domingos (2010b) and Ahmadi *et al.* (2011) provide an approximate algorithm based on lifted belief propagation, LBP, for lifted, adaptive inference for changing evidence. Nath and Domingos (2010b) reuse results from previous algorithm runs and propagate messages only in affected regions. Ahmadi *et al.* (2011) consider Gaussian belief propagation for continuous variables. But, a comprehensive algorithm for exact adaptive inference in probabilistic relational models is still missing.

This chapter focuses on exact inference for multiple queries and presents an efficient algorithm for adaptive inference based on LJT, called aLJT, handling changes in model and evidence. The following paper presented aLJT

> Tanya Braun and Ralf Möller. Adaptive Inference on Probabilistic Relational Models. In *Proceedings of AI 2018: Advances in Artificial Intelligence*, pages 487–500. Springer, 2018

This paper includes two main contributions, (i) procedures for adapting an FO jtree to incremental changes for its underlying model and (ii) an algorithm, aLJT, preserving as much work as possible under changes in a model. LJT for adaptive inference handles changes ranging from new evidence to extending a model with new factors containing new randvars. The algorithm quickly reaches the point of answering queries again after changes, which is especially important for time-critical or online query answering.

In this chapter, we present both the procedures for FO jtrees as well as aLJT, which make up Contribution (6a) and (6b) of this dissertation. We start with discussing the potential of adaptive inference for LJT. We continue with the two main contributions regarding adjusting an FO jtree and adaptive LJT. A theoretical analysis follows, concentrating on soundness, which compared to the paper mentioned above, is more fleshed out. An empirical evaluation highlights how the adaptive nature of aLJT allows for faster QA given consecutive changes to a model.

11.1 The Potential of Adaptive Inference and LJT

The LJT use case is answering a set of queries $\{\mathbf{Q}_i\}_{i=1}^m$ given a model G and evidence $\mathbf{E}$. LJT assumes a constant G for which it builds an FO jtree J, reusing J for varying evidence and queries. However, G may change, and changes do not necessarily mean a completely new evidence set or model. Given incremental changes, LJT can partially preserve J, local models, or messages. Let us consider what changes in any of the inputs mean for LJT and how LJT could exploit that certain parts remain unchanged under incremental changes.

Queries Changing queries are the main feature of LJT which only requires a model and evidence for setting up an FO jtree including messages. In an online QA scenario, LJT can answer any query coming in once message passing is complete. Given a new query, LJT continues answering queries, using J with its current local models and received messages. To speed up runtimes for queries, one needs to handle internal computations differently, e.g., through caching, as discussed earlier as a possible course for future work.

Evidence Currently, if evidence changes, LJT resets the local models in J to their original form, deleting messages and evidence. Then, it enters evidence. But, evidence may change only partially, making full evidence entering unnecessary. Only if evidence changes affect a parcluster $\mathbf{C}_i$, LJT has to redo evidence entering for $\mathbf{C}_i$.

Model If changes in a model occur, LJT currently construct an FO jtree for the changed model. Changes in a model belong to two categories, namely, changes that affect the model structure through arguments in parfactors and changes that do not. In case of changes in *potentials*, *domains* or *constraints*, and *ranges*, the model structure remains the

same, which entails that the parclusters remain the same. LJT could restart with evidence entering to ensure that it handles evidence correctly after updating the parfactors and constraints. New potentials in a parfactor or modified ranges usually concern only a part of a model, leaving parts of the FO jtree untouched. Incremental changes affecting the *model structure* occur in the form of deleting, adding, or replacing a parfactor. As the model is identical in large parts, we assume that it is efficient to keep J and adapt it accordingly, saving the effort of reconstructing unchanged parts of the model.

After changes in evidence and model, full message passing may be unnecessary since not all local models may have changed. Assume that evidence has changed for one peripheral parcluster, yielding a changed local model G_i. LJT only needs to distribute the changed G_i to the other parclusters. The same holds for model changes that affect only a handful of parclusters. These deliberations guide us in setting up adaptive steps for LJT to handle incremental changes. But before presenting aLJT, we look at how to adapt an FO jtree to changes, which becomes a subroutine of aLJT.

11.2 Adapting an FO Jtree to Model Changes

Model changes may yield a structure change in a model, which may cause a structure change in an FO jtree $J = (V, E)$. All actions towards adapting J need to ensure that J continues to be a minimal FO jtree and local models still partition G. This section looks at adding, deleting, or replacing a parfactor in a model G with an FO jtree J.

Addition Adding a parfactor g^+ to G requires adding g^+ to a local model for the local models in J to partition $G \cup \{g^+\}$. If the arguments in g^+ appear in a parcluster $\mathbf{C}_i$, we add g^+ to G_i. But, if g^+ contains new PRVs or if the known PRVs in g^+ do not appear in a single parcluster, there is no parcluster $\mathbf{C}_i$ s.t. $rv(g^+) \subseteq \mathbf{C}_i$. Thus, we adjust J until the known PRVs appear in a single parcluster, and handle the new PRVs accordingly.

Algorithm 12 shows pseudocode for adding g^+ to $J = (V, E)$. The instructions for marking parclusters become relevant for aLJT. We assume that g^+ contains at least one PRV from G to yield a single FO jtree and not a forest. Otherwise, in a nutshell, one has to search for argument PRVs in the forest and combine FO jtrees if necessary. Let $\mathbf{A}^{old}$ refer to the known PRVs and $\mathbf{A}^{new}$ to the newly introduced PRVs. We first consider the whole ADD procedure before delving into adjusting an FO jtree. After adjusting J, there is a parcluster $\mathbf{C}_i$ s.t. $\mathbf{A}^{old} \subseteq \mathbf{C}_i$. If g^+ includes only $\mathbf{A}^{old}$, procedure ADD adds g^+ to the local model G_i at $\mathbf{C}_i$. If g^+ contains new PRVs, ADD distinguishes between $\mathbf{A}^{old} \subset \mathbf{C}_i$ and $\mathbf{A}^{old} = \mathbf{C}_i$. In the former case, there are PRVs in $\mathbf{C}_i$ that do not appear in $rv(g^+)$ and vice versa. ADD adds a new node $\mathbf{C}_k \leftarrow rv(g^+)$ with $G_k \leftarrow \{g^+\}$ as a neighbour to i to keep parclusters small. In the latter case, $\mathbf{C}_i$ is a strict subset of the PRVs in g^+. ADD adds the new PRVs to $\mathbf{C}_i$ and g^+ to G_i for a minimal J. Now, the local models partition G'. The following example illustrates a simple addition.

Algorithm 12 Adding a parfactor g^+ to an FO jtree $J = (V, E)$

procedure ADD(FO jtree J, parfactor g^+)
  $\mathbf{A}^{old} \leftarrow rv(g^+) \cap rv(V)$, $\mathbf{A}^{new} \leftarrow rv(g^+) \setminus rv(V)$
  ADJUST(J, $\mathbf{A}^{old}$)
  Get $\mathbf{C}_i \in V$ where $\mathbf{A}^{old} \subseteq \mathbf{C}_i$
  if $\mathbf{A}^{new} = \emptyset$ **then**
    $G_i \leftarrow G_i \cup \{g^+\}$
    Mark $\mathbf{C}_i$
  else if $\mathbf{A}^{old} = \mathbf{C}_i$ **then**
    $\mathbf{C}_i \leftarrow \mathbf{C}_i \cup rv(g^+)$, $G_i \leftarrow G_i \cup \{g^+\}$
    Mark $\mathbf{C}_i$
  else
    Create $\mathbf{C}_k \leftarrow rv(g^+)$, $G_k \leftarrow \{g^+\}$
    Add $\mathbf{C}_k$ to V and $\{i, k\}$ to E
    Mark $\mathbf{C}_k$

procedure ADJUST(FO jtree J, PRVs $\mathbf{A}$)
  Extract a set of nodes N s.t. $\mathbf{A} \subseteq rv(N)$
  while $|N| > 1$ **do**
    Take two parclusters $\mathbf{C}_i, \mathbf{C}_j \in N$
    $P \leftarrow$ path from $\mathbf{C}_i$ to $\mathbf{C}_j$ including $\mathbf{C}_i, \mathbf{C}_j$
    Mark parclusters on P
    while $length(P) > 1$ **do**
      Merge $P[0]$ and $P[length(P) - 1]$ in J ▷ Definition 4.3.2 plus mark
      if $\exists \mathbf{C}_k, \mathbf{C}_l$ on $P : \mathbf{S}_{kl} \subseteq \mathbf{S}_{P[0]P[1]} \wedge \mathbf{S}_{kl} \subseteq \mathbf{S}_{P[length(P)-2]P[length(P)-1]}$ **then**
        Remove $\{k, l\}$ from E
        break
      $P \leftarrow P[1 \ldots length(P) - 2]$
    Update N regarding merged parclusters

Example 11.2.1 (Simple Addition)**.** Consider the FO jtree for G_{ex} as in Fig. 4.1. We add to G_{ex} the parfactor $g_4 = \phi_4(Epid, Sick(X), Work(X))$, where PRV $Work(X)$ holds if a person X works. For g_4, the known PRVs are $Epid, Sick(X)$ which appear in $\mathbf{C}_2$ and $\mathbf{C}_3$. Assume Alg. 12 chooses $\mathbf{C}_3$. Parfactor g_4 contains a new PRV, $Work(X)$, and $\mathbf{C}_3$ contains a PRV not in g_4, $Treat(X, M)$. Thus, Alg. 12 adds a parcluster $\mathbf{C}_4 = \{Epid, Sick(X), Work(X)\}$, $G_4 = \{g_4\}$. Figure 11.1 shows the result.

Procedure ADJUST in Alg. 12 arranges that $\mathbf{A}^{old} \subseteq \mathbf{C}_i$ for some $\mathbf{C}_i \in V$. ADJUST finds a set of parclusters N that cover $\mathbf{A}^{old}$ and merges N into a single parcluster to fulfil $\mathbf{A}^{old} \subseteq \mathbf{C}_i$. To merge N, it successively merges two parclusters $\mathbf{C}_i, \mathbf{C}_j \in N$. A merge is a union of parclusters, local models, and neighbours as given in Definition 4.3.2. Since

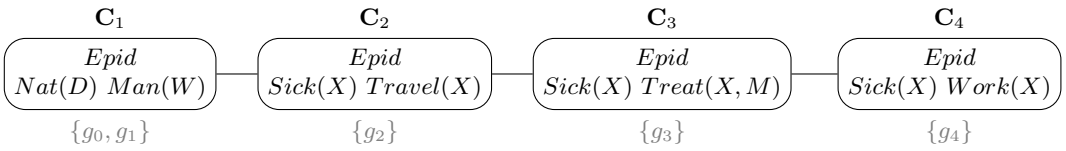
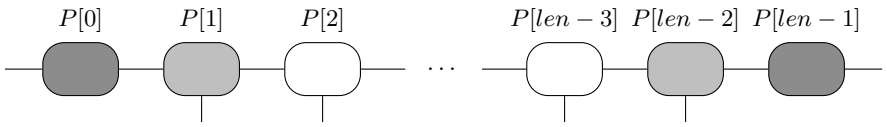
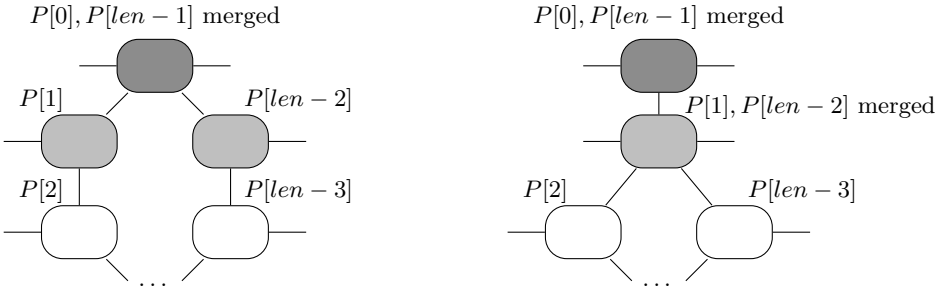


Figure 11.1: FO jtree after adding a parfactor



(a) Before merging $P[0]$ and $P[len-1]$



(b) After first merging

(c) After second merging

Figure 11.2: A path of parclusters; $len = length(P)$

J is acyclic, there exists a unique path P from $\mathbf{C}_i$ to $\mathbf{C}_j$ including $\mathbf{C}_i$ and $\mathbf{C}_j$, which forms a cycle if there lies more than one parcluster between $\mathbf{C}_i$ and $\mathbf{C}_j$, i.e., $|P| > 3$. Figure 11.2a depicts such a path of parclusters with $P[0] = \mathbf{C}_i$ and $P[len-1] = \mathbf{C}_j$. Merging $\mathbf{C}_i$ and $\mathbf{C}_j$ leads to a cycle, a scenario which is depicted in Fig. 11.2b.

There are two ways to resolve the cycle. The first and easy, though more unlikely, way is to find an appropriate edge to delete on the cycle. Deleting an edge would break the cycle, but it would require the running intersection property not be violated. Thus, an edge $\{k, l\}$ on the path has to fulfil a condition given the separator $\mathbf{S}_{kl}$ of that edge:

$$\mathbf{S}_{kl} \subseteq \mathbf{S}_{P[0]P[1]} \wedge \mathbf{S}_{kl} \subseteq \mathbf{S}_{P[length(P)-2]P[length(P)-1]}, \tag{11.1}$$

i.e., information on $\mathbf{S}_{kl}$ reaches $\mathbf{C}_k$ from $P[0]$ and $\mathbf{C}_l$ from $P[length(P)-1]$. If $\mathbf{S}_{kl}$ exists, ADJUST deletes the edge $\{k, l\}$ to break the cycle, which keeps the parclusters on P small. The second way to resolve the cycle is to merge parclusters of the path, always merging the parclusters at the ends, i.e., $P[0]$ and $P[length(P)-1]$, $P[1]$ and $P[length(P)-2]$, and so on until the cycle is resolved. Figures 11.2b and 11.2c show the parclusters after

the first and second merging. After each merging of parclusters at path ends, we again have the situation as in Fig. 11.2b. After the second merging as in Fig. 11.2c, the newly merged light grey parcluster becomes the dark grey parcluster in Fig. 11.2b. In this smaller cycle, Expression (11.1) might be true for some $\mathbf{S}_{kl}$. Thus, the pseudocode of ADJUST reflects this situation occurring recursively by merging path ends, removing the end parclusters from P, and checking Expression (11.1), until the cycle is resolved either by deleting an edge or by merging all parclusters on the path from opposite ends. To illustrate adjusting an FO jtree, we look at a slightly extended example FO jtree.

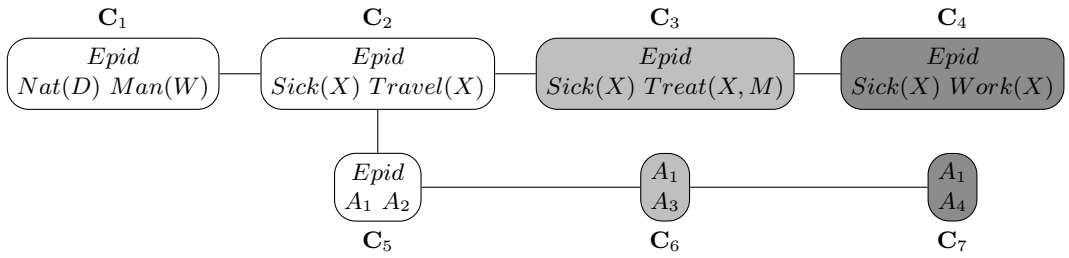
Example 11.2.2 (Adjusting)**.** Let the adapted FO jtree in Fig. 11.1 have three more parclusters with PRVs A_1, A_2, A_3, and A_4, shown in Fig. 11.3a. We add a parfactor $g' = \phi'(A_4, Work(X))$. A_4 appears in parcluster $\mathbf{C}_7$, while $Work(X)$ appears in parcluster $\mathbf{C}_4$, i.e., $N = \{\mathbf{C}_7, \mathbf{C}_4\}$, marked in dark grey in Fig. 11.3a. ADJUST merges $\mathbf{C}_4$ and $\mathbf{C}_7$ into $\mathbf{C}'_4$, which causes a cycle, shown in Fig. 11.3b. The unique path between the former $\mathbf{C}_7$ and $\mathbf{C}_4$ goes over the parclusters with indices $4, 3, 2, 5, 6, 7$. No separator on the path appears in $\mathbf{C}_4$ and $\mathbf{C}_7$, that is Expression (11.1) is not fulfilled, to easily break the cycle. ADJUST continues merging parclusters on a path that now covers the indices $3, 2, 5, 6$. ADJUST merges the path ends $\mathbf{C}_3$ and $\mathbf{C}_6$ into $\mathbf{C}'_3$, the result shown in Fig. 11.3c. At this point, separator $\mathbf{S}_{25} = \{Epid\}$ of edge $\{2, 5\}$ (thick edge in Fig. 11.3c) appears in the former $\mathbf{C}_3$ and $\mathbf{C}_6$. Therefore, ADJUST deletes edge $\{2, 5\}$, yielding a valid FO jtree with five parclusters as seen in Fig. 11.3d. At $\mathbf{C}'_4$, Alg. 12 can add g' to the local model.

Instead of deleting the edge, ADJUST could have merged $\mathbf{C}_2$ and $\mathbf{C}_5$ into one parcluster $\{Epid, Sick(X), Travel(X), A_1, A_2\}$, which would have lead to a valid FO jtree as well but would have one larger parcluster with five PRVs instead of two parclusters with three PRVs each, which means fewer calculations during query answering.

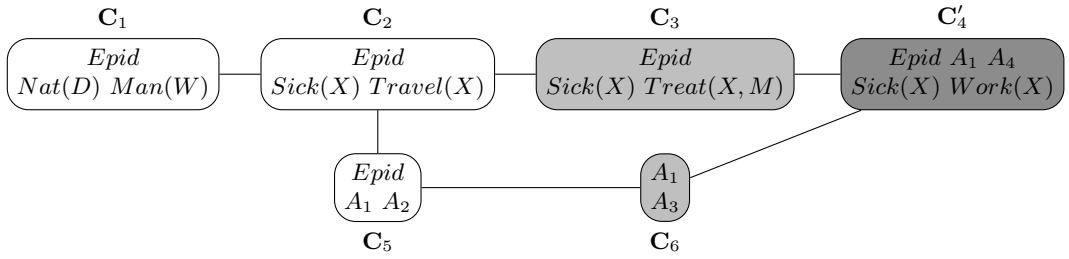
Deletion Deleting a parfactor g^- from G requires removing g^- from the local model G_i in which g^- appears. Afterwards, the local models partition $G \setminus \{g^-\}$. Deleting g^- might remove a PRV A from G_i if only g^- contained A in G_i. If A no longer appears in G_i, it may no longer need to be in $\mathbf{C}_i$ and could be removed from $\mathbf{C}_i$, but only if removing A from $\mathbf{C}_i$ does not violate the running intersection property. Removing A from $\mathbf{C}_i$ is permissible if no two separators contain A, i.e.,

$$\forall j, k \in nbs(i) : A \notin \mathbf{S}_{ij} \wedge A \notin \mathbf{S}_{ik}. \tag{11.2}$$

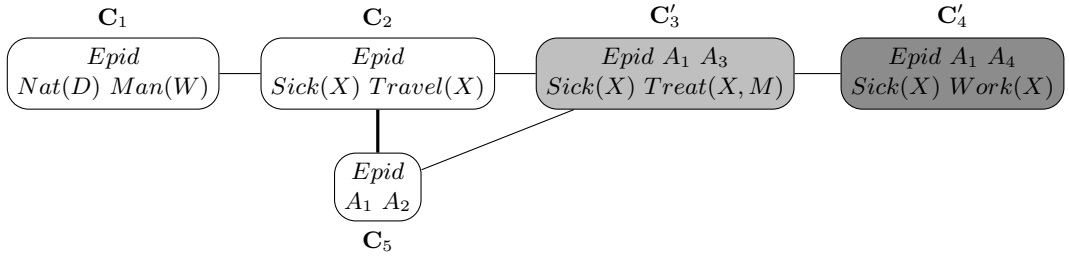
Procedure DELETE in Alg. 13 describes deleting g^- from $J = (V, E)$. After removing g^- from G_i, DELETE minimises $\mathbf{C}_i$. Procedure MINIMISE goes through the set of PRVs that no longer appear and checks for each PRV A if Expression (11.2) holds. If so, MINIMISE deletes A from $\mathbf{C}_i$, which minimises the messages that arrive from neighbours that contain A, which also reduces the number of calculations during query answering. If deleting PRVs from $\mathbf{C}_i$ leads to $\mathbf{C}_i \subseteq \mathbf{C}_j$ for a neighbour $\mathbf{C}_j$, MINIMISE merges $\mathbf{C}_i$ and $\mathbf{C}_j$ to keep J minimal. Let us look at deleting a parfactor.



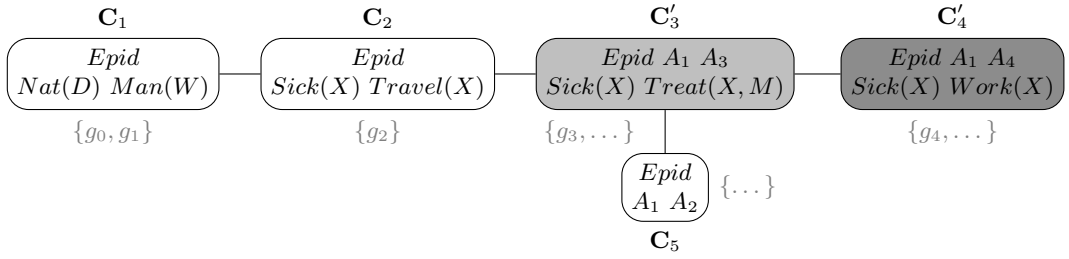
(a) Before adjusting



(b) After first merging



(c) After second merging



(d) After breaking cycle

Figure 11.3: Extended FO jtree during adjusting

Algorithm 13 Deleting a parfactor g^- from an FO jtree $J = (V, E)$

procedure DELETE(FO jtree J, parfactor g^-)
  Get $\mathbf{C}_i \in V$ s.t. $g^- \in G_i$
  $G_i \leftarrow G_i \setminus \{g^-\}$
  MINIMISE(J, $\mathbf{C}_i$, $rv(g^-) \setminus rv(G_i)$)
  Mark $\mathbf{C}_i$

procedure MINIMISE(FO jtree J, node $\mathbf{C}_i$, PRVs $\mathbf{A}$)
  for PRV $A \in \mathbf{A}$ **do**
    if $\forall j, k \in nbs(i) : A \notin \mathbf{S}_{ij} \wedge A \notin \mathbf{S}_{ik}$ **then**
      $\mathbf{C}_i \leftarrow \mathbf{C}_i \setminus \{A\}$
      Mark $\mathbf{C}_i$
  if $\mathbf{C}_i$ marked $\wedge\ \exists j \in nbs(i)$ s.t. $\mathbf{C}_i \subseteq \mathbf{C}_j$ **then**
    Merge $\mathbf{C}_i$ and $\mathbf{C}_j$ in J ▷ Definition 4.3.2 plus mark

Example 11.2.3. Consider the FO jtree in Fig. 11.3d. At parcluster $\mathbf{C}_1$, the local model is $G_1 = \{g_0, g_1\}$. We look at two scenarios, first deleting g_0 and second deleting g_1. Deleting g_0 would lead to a local model $\{g_1\}$, which still contains all PRVs of the parcluster, i.e., deleting g_0 has no other consequence for the FO jtree. Deleting g_1 would lead to a local model $\{g_0\}$, which contains only the PRV *Epid*. The two PRVs $Nat(D)$ and $Man(W)$ would no longer appear in G_1, which would mean they may be removed from $\mathbf{C}_1$ if the removal does not violate the running intersection property. As both PRVs do not appear in the only neighbour of $\mathbf{C}_1$, $Nat(D)$ and $Man(W)$ could both be removed from $\mathbf{C}_1$. After removing the PRVs, $\mathbf{C}_1$ would be a subset of $\mathbf{C}_2$. Therefore, MINIMISE could merge $\mathbf{C}_1$ and $\mathbf{C}_2$, leading to a parcluster $\mathbf{C}_2'$ with a local model of $\{g_0, g_2\}$.

Replacement Replacing a parfactor g^- with a parfactor g^+ in G boils down to adding g^+ and then deleting g^-, shown in Alg. 14. If both parfactors contain the same arguments, i.e., $rv(g^-) = rv(g^+)$, adding g^+ and deleting g^- only exchanges g^- with g^+ in the local model that contains g^-. If the new parfactor contains more PRVs than the old one, i.e., $rv(g^-) \subseteq rv(g^+)$, adding g^+ changes J in the sense that the parcluster of g^- is extended or a new parcluster added for $rv(g^+)$. If the parcluster of g^- is extended for g^+, deleting g^- does not change J any further as $rv(g^-)$ still appears in the parcluster through g^+. If a new parcluster is added for g^+, deleting g^- may lead to changes for the parcluster of g^-. First deleting g^- may lead to removing PRVs and superfluously merging parclusters before adding g^+. If the new parfactor contains fewer PRVs than the old parfactor, i.e., $rv(g^+) \subseteq rv(g^-)$, adding g^+ before deleting g^- uses that there exists a parcluster $\mathbf{C}_i$ with $rv(g^+) \subseteq \mathbf{C}_i$ as $rv(g^-) \subseteq \mathbf{C}_i$. If the arguments of g^- and g^+ overlap otherwise, i.e., $rv(g^-) \cap rv(g^+) \neq \emptyset$, first adding g^+ and then deleting g^- avoids unnecessarily deleting PRVs and merging parclusters for the overlap. If both parfactors do not share any PRVs,

Algorithm 14 Replacing a parfactor g^- with a parfactor g^+ in an FO jtree $J = (V, E)$

procedure REPLACE(FO jtree J, parfactor g^-, parfactor g^+)
  ADD(J, g^+)
  DELETE(J, g^-)

i.e., $rv(g^-) \cap rv(g^+) = \emptyset$, replacing g^- with g^+ naturally decomposes into adding g^+ and deleting g^-. To illustrate replacing parfactors, we look at the following example.

Example 11.2.4. Consider the FO jtree in Fig. 11.3d with its five parclusters again (without Example 11.2.3). Let us replace $g_2 = \phi_2(Epid, Travel(X), Sick(X))$ with a parfactor $g_2' = \phi_2'(Travel(X), Sick(X))$. The new parfactor contains fewer PRVs than the old parfactor. Algorithmically, the replacement involves adding g_2' and deleting g_2. As $rv(g_2') \subseteq \mathbf{C}_2$, one adds g_2' to $\mathbf{C}_2$. Then, REPLACE deletes g_2 at $\mathbf{C}_2$. After deleting g_2 from G_2, *Epid* no longer appears in G_2. But, *Epid* appears in both its separators and as such, has to remain in $\mathbf{C}_2$ to connect the appearance of *Epid* from $\mathbf{C}_1$ to $\mathbf{C}_3$.

In contrast to the above scenario of $\mathbf{C}_2$ remaining the same, assume that we replace g_2 with $g_2' = \phi_2'(Epid, Travel(X))$. Then, after g_2 is deleted from G_2, MINIMISE would remove $Sick(X)$ from $\mathbf{C}_2$ as $Sick(X)$ appears only in one separator. If $g_2' = \phi_2'(Epid, Sick(X))$, MINIMISE would remove $Travel(X)$ after DELETE has deleted g_2. Then, MINIMISE would merge $\mathbf{C}_2$ and $\mathbf{C}_3$ since $\mathbf{C}_2$ is a subset of $\mathbf{C}_3$.

This example concludes this section on adapting an FO jtree to changes in the form of adding, deleting, or replacing parfactors. The next section details adaptive steps for LJT to perform adaptive inference given incremental changes in the inputs of LJT.

11.3 LJT for Adaptive Inference

The extended algorithm aLJT performs adaptive inference more efficiently than by restarting from scratch. Algorithm aLJT again consists of the steps construction, evidence entering, and message passing before it answers queries. But, each step proceeds in an adaptive manner w.r.t. changes in input model G or in evidence $\mathbf{E}$ given an FO jtree J. If no FO jtree is present, aLJT reverts to LJT, requiring a model G to build J.

Algorithm 15 shows a description of aLJT for J, referring to the changes in G and $\mathbf{E}$ by Δ_G and $\Delta_{\mathbf{E}}$. Line 2 contains the adaptive construction step, which adapts J to Δ_G according to Algs. 12 to 14. To track changes, aLJT marks a parcluster $\mathbf{C}_i$ if a local model G_i changes s.t. the messages become invalid. Based on the marks and $\Delta_{\mathbf{E}}$, aLJT performs an adaptive evidence entering and message passing, answering queries as before. Lines 3 to 6 show adaptive evidence entering and lines 7 to 12 adaptive message passing. Lines 13 to 16 contain the steps to answer a query $\mathbf{Q}_i$ from a set of queries $\{\mathbf{Q}_i\}_{i=1}^m$, as in LJT. Next, we look at the adaptive steps, followed by an example.

Algorithm 15 LJT for adaptive inference

```
1: procedure aLJT(FO jtree J, Queries {Q_k}_{k=1}^m, changes Δ_G, changes Δ_E)
2:     Adapt J to Δ_G according to Algs. 12 to 14          ▷ marks parclusters
3:     for each parcluster C_i in J do
4:         if C_i marked or affected by Δ_E then
5:             Handle evidence at C_i
6:             Mark C_i
7:     while ∃ C_i ready to send message m_ij to C_j in J do
8:         if C_i marked, has marked message, or C_j is new then
9:             Send newly computed m_ij
10:        else
11:            Send empty message          ▷ Indicates for receiver "no changes"
12:        Mark m_ij at C_j as un/changed
13:    for each Q_k ∈ {Q_k}_{k=1}^m do
14:        Find a subtree J' s.t. Q_k ⊆ rv(J')
15:        Extract a submodel G' from J'
16:        LVE(G', Q_k, ∅)                          ▷ Output or store result
```

Construction aLJT handles changes Δ_G as in Algs. 12 to 14. Δ_G refers to parfactors to add, delete, or replace. When adding a parfactor, ADD marks the parcluster $\mathbf{C}_i$ to which the new parfactor is added. If adjusting J for known PRVs, all parclusters are marked on the path between two parclusters $\mathbf{C}_i, \mathbf{C}_j \in N$ that are merged. The messages for the parclusters on this path become invalid as neighbours disappear and appear as well as local models change with merging. Messages from parclusters that do not lie on the path remain valid at the parclusters on the path. When deleting a parfactor from the local model of $\mathbf{C}_i$, DELETE marks $\mathbf{C}_i$ and every parcluster. aLJT replaces a parfactor by adding and deleting, which includes marks.

Example 11.3.1 (Marking)**.** In Example 11.2.1, we add $\phi_4(Epid, Sick(X), Work(X))$ to G_{ex}, which leads to an additional parcluster $\mathbf{C}_4$ with local model $G_4 = \{g_4\}$ (see Fig. 11.1). ADD marks $\mathbf{C}_4$ as it is new.

For changes in potentials, ranges, or constraints, aLJT uses replacing a parfactor where the arguments of the old and new parfactor are identical. If the domain for a logvar X changes, aLJT marks a parcluster $\mathbf{C}_i$ if $X \in lv(\mathbf{C}_i)$ and the constraint w.r.t. X is $\top$. After incorporating all changes, aLJT has marked parclusters in J accordingly.

Evidence Entering Adaptive entering deals with changes $\Delta_\mathbf{E}$ in evidence and with evidence entering at parclusters marked during construction. In the first case, aLJT enters evidence at all parclusters $\mathbf{C}_i$ that are affected by $\Delta_\mathbf{E}$. $\Delta_\mathbf{E}$ refers to changes in the

form of additional or retracted evidence or new observed values. For additional evidence, aLJT uses the current local model G_i and enters the additional evidence. For retracted evidence, aLJT resets parfactors where the evidence no longer appears. The reset may require a reentering of evidence if evidence for a PRV is partially retracted. For new values, aLJT only needs to reset parfactors that have absorbed the original evidence. These parfactors absorb the new values. If evidence leads to changes in G_i, aLJT marks $\mathbf{C}_i$. In the second case, parcluster marked during construction only need evidence entering if model changes are affected by evidence. That is if domains change, evidence needs to be entered anew, or if new parfactors are added, the respective parfactors possibly need to absorb evidence.

Example 11.3.2 (Adaptive evidence)**.** Continuing with Example 11.3.1, assume that evidence $sick(eve)$ had been entered in the FO jtree before. Then, aLJT only has to add $sick(eve)$ to the new parcluster $\mathbf{C}_4$, which is the only parcluster marked in Fig. 11.1.

Message Passing aLJT maintains the same two-pass scheme starting at the periphery going inward and returning to the periphery. Using the scheme preserves the ability for an automatic execution based on the two conditions given in Section 4.3. The adaptive part occurs during message calculation. A parcluster $\mathbf{C}_i$ calculates a new message if messages have become invalid or if $\mathbf{C}_i$ has to distribute changes in its local model or in received messages. Otherwise, it sends an empty message. The receiver replaces the old message with the new message (if not empty) and marks it changed or marks the old message as unchanged. Formally, $\mathbf{C}_i$ calculates a new message m_{ij} for neighbour $\mathbf{C}_j$ if

$$\mathbf{C}_i \text{ marked } \vee \ \exists k \in nbs(i), k \neq j : m_{ki} \text{ changed} \tag{11.3}$$

If Expression (11.3) holds, $\mathbf{C}_i$ computes m_{ij} using LVE with $G' \leftarrow G_i \cup \bigcup_{k \in nbs(i), k \neq j} m_{ki}$ as model (messages irregardless of whether they are marked changed) and $\mathbf{S}_{ij}$ as query. The following example illustrates how aLJT calculates messages according to changes at a parcluster.

Example 11.3.3 (Adaptive messages)**.** Continuing with Example 11.3.2, only $\mathbf{C}_4$ is marked. aLJT starts message passing at the leafs, i.e., $\mathbf{C}_1$ and $\mathbf{C}_4$. $\mathbf{C}_1$ is not marked, meaning, an empty messages arrives at $\mathbf{C}_2$. Message m_{12} is therefore marked unchanged. $\mathbf{C}_4$ is marked, i.e., a new message m_{43} is calculated and sent to $\mathbf{C}_3$, which stores the new message and marks it as changed. Then, $\mathbf{C}_2$ and $\mathbf{C}_3$ exchange messages. Messages m_{23} is empty as $\mathbf{C}_2$ is unmarked and m_{12} is marked as unchanged. Message m_{32} is recalculated as m_{43} is marked as changed. At $\mathbf{C}_2$, m_{32} arrives and is marked as changed, meaning a new message m_{21} is calculated for $\mathbf{C}_1$. After these messages, the respective marginal of the new parfactor at $\mathbf{C}_4$ has been made available at the other parclusters. At $\mathbf{C}_3$, aLJT calculates a new message m_{34} as $\mathbf{C}_4$ is new. With m_{34} arriving at $\mathbf{C}_4$, message passing is complete. The FO jtree is prepared for query answering.

Over the course of the previous examples, which started with adding a parfactor, aLJT could reuse the FO jtree in large parts, simply extending the structure with a new parcluster. Evidence entering was limited to the new parcluster, saving the effort of reentering evidence at two parclusters. Adaptive message passing allowed aLJT to avoid calculating half of the existing messages. The remaining step of aLJT is query answering.

Query Answering After message passing, aLJT starts answering queries, which follows the same procedure as before. Let us look at the extended FO jtree in Fig. 11.3a under various changes to see aLJT as a whole in action.

Example 11.3.4 (aLJT)**.** Assume that the FO jtree in Fig. 11.3a has as evidence $sick(eve)$ entered and valid messages passed. The FO jtree in Fig. 11.3a then undergoes a change by having a parfactor $\phi(Work(X), A_4)$ added, which leads to the FO jtree in Fig. 11.3d. When adjusting the FO jtree to $\phi(Work(X), A_4)$, ADJUST first marks $\mathbf{C}_4$ and $\mathbf{C}_7$, the two parclusters in N that cover $Work(X)$ and A_4. The path over indices $4, 3, 2, 5, 6, 7$ leads to parclusters $\mathbf{C}_3$, $\mathbf{C}_2$, $\mathbf{C}_5$, and $\mathbf{C}_6$ being marked. Merging parclusters leads to $\mathbf{C}'_4$ and $\mathbf{C}'_3$ being marked. After adjusting is complete, all parclusters in Fig. 11.3d are marked except $\mathbf{C}_1$, leading to an almost complete message passing. The only empty message is m_{12}. After message passing, aLJT answers queries as before, e.g., for query term $\{Treat(eve, injection)\}$ on $\mathbf{C}'_3$, for query terms $\{Treat(eve, injection), Travel(eve)\}$ on $\mathbf{C}_2$ and $\mathbf{C}'_3$, or for query term $\{Sick(X)\}$, choosing $\mathbf{C}_2$ for answering the query.

Next, assume that we add some evidence for $Nat(D)$ at $\mathbf{C}_1$, which leads aLJT to mark $\mathbf{C}_1$. With no further changes, aLJT only needs to redistribute G_1. Thus, messages m_{53} and m_{43} from $\mathbf{C}_5$ and $\mathbf{C}_4$ to $\mathbf{C}_3$ are empty as well as the messages from $\mathbf{C}_3$ over $\mathbf{C}_2$ to $\mathbf{C}_1$ since no changes occur in local models. Message m_{12} from $\mathbf{C}_1$ to $\mathbf{C}_2$ is newly calculated. The new message received by $\mathbf{C}_2$ leads to new messages from $\mathbf{C}_2$ to $\mathbf{C}_3$ and from $\mathbf{C}_3$ back to the leaf nodes $\mathbf{C}_4$ and $\mathbf{C}_5$. Again, aLJT can now answer queries as before.

The first case of adding a parfactor $\phi(Work(X), A_4)$ shows a rather large effect on the FO jtree, with a majority of parclusters marked after adjusting the FO jtree. Still, it allows aLJT to reuse local models where evidence has already been entered. With the next change in the inputs in the form of additional evidence, aLJT enters evidence at one parcluster and recalculates only half of the messages in the FO jtree. The upcoming section argues the correctness of the above algorithms.

11.4 Theoretical Analysis

This section provides a theoretical analysis that focusses primarily on the soundness of aLJT as well as the algorithms for adapting an FO jtree. Completeness is not affected by rearranging an FO jtree. This section concludes with a discussion of the effect of adaptive steps on runtime complexity.

Theorem 11.4.1. *Adding a parfactor g^+ to a model G with an FO jtree J yields an FO jtree J' for $G \cup \{g^+\}$.*

Proof. FO jtree J for model G fulfils the three properties that define an FO jtree. When adding parfactor g^+ to G, we have to add g^+ to a local model G_i where $rv(g^+) \subseteq \mathbf{C}_i$ in J. Let $\mathbf{A}^{old} \leftarrow rv(g^+) \cap rv(G)$ and $\mathbf{A}^{new} \leftarrow rv(g^+) \setminus rv(G)$ denote the sets of old and new PRVs in g^+. If $\mathbf{A}^{old} \subseteq \mathbf{C}_i$ for some i and $\mathbf{A}^{new} = \emptyset$, we add g^+ to G_i. As we do not change J apart from a local model, J is still an FO jtree, now for $G \cup \{g^+\}$. If $\mathbf{A}^{new} \neq \emptyset$, the second property ($rv(g^+) \subseteq \mathbf{C}_i$) does not hold for g^+. We distinguish the cases $\mathbf{A}^{old} \subset \mathbf{C}_i$ and $\mathbf{A}^{old} = \mathbf{C}_i$. In the first case, we add a new node $\mathbf{C}_k$ as a neighbour to i with $rv(g^+)$ in $\mathbf{C}_k$ and g^+ in G_k. Now, the second property holds. The first property (parclusters must contain model PRVs) holds for $\mathbf{C}_k = rv(g^+)$ given $G \cup \{g^+\}$. $\mathbf{C}_k$ does not violate the third property (a PRV appearing in two parclusters has to appear in each parcluster on the path between them): $\mathbf{A}^{new}$ does not appear any further. $\mathbf{A}^{old}$ appears in $\mathbf{S}_{ik}$, continuing on from $\mathbf{C}_i$ to $\mathbf{C}_k$. Thus, J with new parcluster $\mathbf{C}_k$ is an FO jtree for $G \cup \{g^+\}$. In the second case, we add $\mathbf{A}^{new}$ to $\mathbf{C}_i$. All three properties hold since $\mathbf{C}_i$ contains PRVs from $G \cup \{g^+\}$, $\mathbf{C}_i$ now includes $rv(g^+)$, and we add PRVs to $\mathbf{C}_i$ that do not appear any further. Therefore, J with an extended $\mathbf{C}_i$ is an FO jtree for $G \cup \{g^+\}$.

Now consider $\mathbf{A}^{old} \not\subseteq \mathbf{C}_i$ for any i, violating the second property. Adding $rv(g^+)$ to some $\mathbf{C}_i$ may violate the third property. Instead, we adjust J s.t. $\mathbf{A}^{old} \subseteq \mathbf{C}_i$ for some i. The basic procedure is merging. Merging two *neighbouring* parclusters preserves the FO jtree properties by building unions of parclusters and neighbours, not affecting any other parclusters. Let N be a set of parclusters s.t. $\mathbf{A}^{old} \subseteq rv(N)$. We successively merge the parclusters in N. Each merge of two parclusters $\mathbf{C}_i, \mathbf{C}_j \in N$ into $\mathbf{C}'_i$ causes a cycle if more than one parcluster lies between $\mathbf{C}_i$ and $\mathbf{C}_j$. To make J acyclic again, we (i) delete an edge to break the cycle or (ii) merge parclusters on the path between $\mathbf{C}_i$ and $\mathbf{C}_j$, which forms a cycle when merging $\mathbf{C}_i$ and $\mathbf{C}_j$. The first approach upholds the third property since we only delete an edge if Expression (11.1) holds, which ensures that the PRVs in a separator $\mathbf{S}_{kl}$ still reach $\mathbf{C}_k$ and $\mathbf{C}_l$ without the edge. The second approach merges the parclusters at the path ends, which upholds the FO jtree properties outside of the path. When merging the next two parclusters from the ends of the path between $\mathbf{C}_i$ and $\mathbf{C}_j$, the cycle becomes smaller, with the newly merged parcluster $\mathbf{C}_k$ forming the new starting point of the cycle while $\mathbf{C}_k$ becomes a neighbour of the parcluster that was previously merged on the cycle. We continue merging parclusters on the cycle until the cycle disappears, yielding an acyclic FO jtree. Thus, in all cases that arise, adding a parfactor g^+ to a model G with an FO jtree J yields an FO jtree J' for $G \cup \{g^+\}$. □

Theorem 11.4.2. *Deleting a parfactor g^- from a model G with an FO jtree J yields an FO jtree J' for $G \setminus \{g^-\}$.*

Proof. FO jtree J for model G fulfils the three FO jtree properties. When deleting a parfactor g^- from G, we remove g^- from G_i where $g^- \in G_i$ at a parcluster $\mathbf{C}_i$ in J

s.t. the local models partition $G \setminus \{g^-\}$. If the deletion removes PRVs from G, the first property (parclusters must contain model PRVs) does not hold anymore at $\mathbf{C}_i$ for $G \setminus \{g\}$. PRVs that no longer appear in $G \setminus \{g^-\}$ do not appear in any other parcluster and thus, fulfil Expression (11.2). Minimising $\mathbf{C}_i$ removes them, which restores the first property. Further, Expression (11.2) prevents deleting a PRV A that lies on a path between two parclusters that contain A, which requires A to remain in the parcluster owing to the third property. The MINIMISE procedure possibly merges $\mathbf{C}_i$ with a neighbour, which upholds the FO jtree properties. Thus, the resulting FO jtree J' is an FO jtree for $G \setminus \{g^-\}$. □

Theorem 11.4.3. *Replacing a parfactor g^- with a parfactor g^+ in a model G with an FO jtree J yields an FO jtree J'' for $(G \cup \{g^+\}) \setminus \{g\}$.*

Proof. To replace a parfactor, we add a parfactor g^+ and delete a parfactor g^-. We showed that adding a parfactor g^+ to J yields a valid FO jtree J' for $G' = G \cup \{g^+\}$. We also showed that deleting a parfactor g^- from an FO jtree, here J', yields a valid FO jtree J'' for $G' \setminus \{g\}$. Thus, J'' is a valid FO jtree for $(G \cup \{g^+\}) \setminus \{g\}$. □

Theorem 11.4.4. *aLJT is sound, i.e., computes a correct result for a query $\mathbf{Q}$ on an FO jtree J after adapting to changes in model G and evidence $\mathbf{E}$.*

Proof. As shown in Sections 5.1, 8.2 and 9.3, LJT is sound for (parameterised) conjunctive queries, yielding an FO jtree J, which partitions the input model G and allows for local computations. Further, we assume that LVE is sound, ensuring correct local computations during evidence entering, message passing, and query answering.

aLJT adapts J, which consists of correctly adding, deleting and replacing parfactors in G as shown in the previous proofs. Thus, adaptive construction outputs an FO jtree with marked parclusters. aLJT adaptively enters the new evidence version at all parclusters covering evidence and re-enters evidence at parclusters with changed local models, ensuring all appropriate parclusters have the current evidence. Evidence absorption is then handled by a sound LVE operator. When passing messages, aLJT distributes updated local models whenever a local model or any incoming message has changed. Messages are calculated by applying sound LVE operators. With messages and local models updated, aLJT uses local models and messages to correctly answer $\mathbf{Q}$ using LVE. □

Effects on Complexity The basic assumption for aLJT is that changes in a model occur incrementally. As such, the changes do not require a completely new FO jtree.

Adapting an FO jtree to changes has an effect on the worst case size of the underlying FO jtree J. The lifted width $(w_g, w_\#)$ of J may change as a result. The effort itself of adapting an FO jtree is polynomial in the number of parclusters but does *not* depend on the worst case size of a factor, i.e., $O(r^{w_g} \cdot n_\#^{w_\# \cdot r_\#})$: To add a parfactor g^+, aLJT has to find a set N of parclusters covering the PRVs in g^+, which means touching each parcluster in a worst case. An optimal set of parclusters, an even harder task to find,

would lead to parclusters as small as possible during adjustment. Adjusting an FO jtree includes merging the parclusters in N, meaning we have $|N| - 1$ times the costs of merging and breaking up a cycle of length $|P|$, where P is the path between the two parclusters to merge. In a best case scenario, aLJT deletes an edge for which we have to check each separator for being a subset of two parclusters. In the worst case, the check fails and aLJT merges $\lfloor \frac{|P|}{2} \rfloor$ times parclusters on the path, leading to as many parclusters. In no case does aLJT manipulate factors. W.r.t. $(w_g, w_\#)$, the width may change due to the arguments of g^+, which provide a lower bound on the width (see Expression (5.9)). Additionally, each resulting parcluster after merging parclusters on a cycle has a worst case size $2 \cdot O(r^{w_g} \cdot n_\#^{w_\# \cdot r_\#})$. An alternative to merging parclusters from both ends of P is merging neighbours, which may lead to better results in certain cases but has a worst case of $|P| - 1$ merges resulting in one parcluster with a worst case size of $(|P| - 1) \cdot O(r^{w_g} \cdot n_\#^{w_\# \cdot r_\#})$. J may deteriorate over time and need reconstruction.

If g^+ contains only new PRVs, one starts a new FO jtree for g^+ and stores a forest of independent FO jtrees. To answer queries for PRVs in one FO jtree, other FO jtrees are irrelevant. A conjunctive query for PRVs from several FO jtree leads to computing the query on the FO jtrees, ignoring external PRVs, and multiplying the results.

Deleting a parfactor g^- has less impact on the FO jtree structure. The worst case scenario for minimising $\mathbf{C}_i$ is checking each combination of two neighbours for each deleted PRV and leading to a change in $\mathbf{C}_i$. Thereby, the width $(w_g, w_\#)$ may decrease if PRVs disappear from the largest parcluster of J. Then, a check follows if $\mathbf{C}_i$ is a subset of some neighbour, which may lead to fewer parclusters in J. Replacing g^- with a parfactor g^+ combines adding and deleting. The parcluster whose local model contains g^- provides a starting point for adding g^+. The new parfactor usually has a connection to the old one, may it be that only potentials or ranges changed or arguments are dropped or added. The best case scenario happens if replacing a parfactor entails no structure change. Then, one parcluster $\mathbf{C}_i$ covers the known PRVs and no PRVs are deleted from $\mathbf{C}_i$. A domain change leaves J untouched, while parclusters are marked accordingly.

Adaptive *evidence entering* may apply to one parcluster in the best case and all parclusters in the worst case, which leads to a complexity of aLJT identical to LJT. Absorbing evidence requires the least work with additional or completely retracted evidence.

In the worst case, *message passing* computes two full messages per edge. The worst case scenario occurs if at least each leaf in J is marked. The best case is one marked parcluster. Then, inward, full messages flow from the marked parcluster to the centre and empty messages otherwise. Outward, parclusters send full messages except on the edges from the centre to the marked parcluster, yielding one full message per edge. Assume that changes affect connected parclusters in one branch of J, which is reasonable given the third FO jtree property. Inward, full messages only occur on the branch to the center. The outward pass transports the updated information to the other branches of J and the outer-branch parclusters. In terms of complexity, aLJT and LJT have the same complexity. But, in practice, aLJT computes fewer messages than LJT in most cases.

The runtime complexity of *query answering* is identical as the procedure of answering a query does not change So, even though in terms of complexity, there is no gain for aLJT compared to LJT. aLJT adapts an FO jtree s.t. the parclusters are as small as possible in the worst case and only deals with evidence and messages if necessary. Overall, aLJT exploits the potential of reusing an FO jtree as well as messages as much as possible.

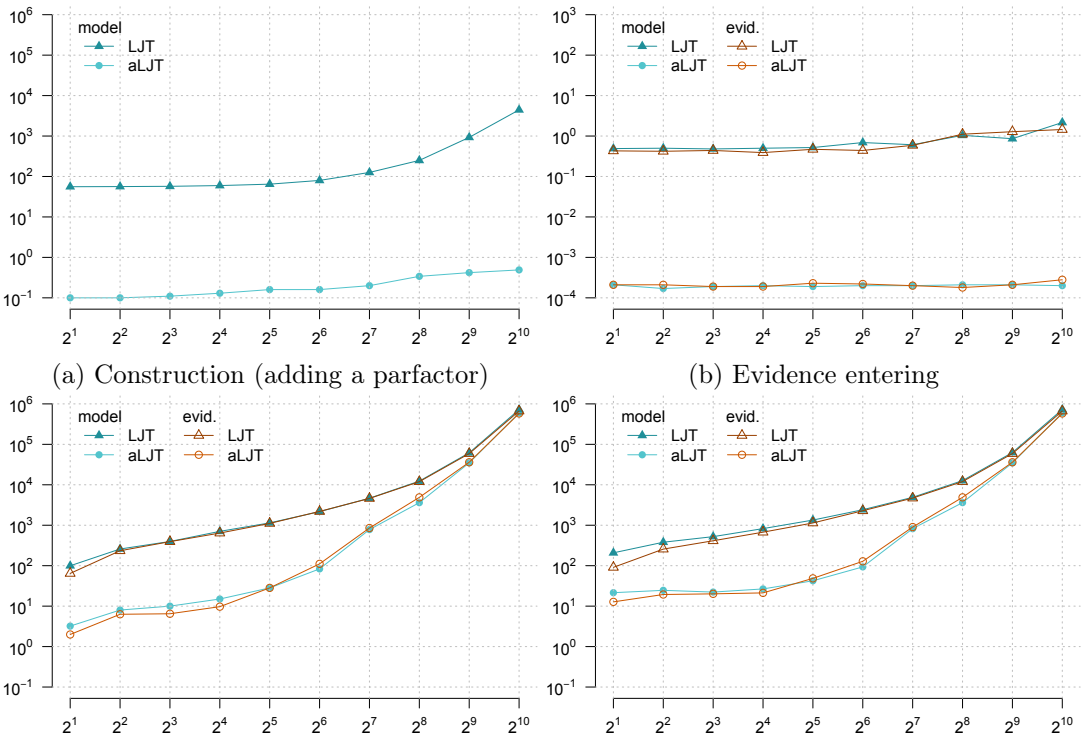
11.5 Empirical Evaluation

We have implemented a prototype of aLJT based on the prototype implementation of LJT and the LVE implementation of Taghipour (`https://dtai.cs.kuleuven.be/software/lve`). We compare aLJT against LJT, with LVE as a baseline. LJT constructs an FO jtree for its input model and enters evidence. It passes messages and answers queries. If a change occurs, it takes the altered input and starts again with construction or evidence entering. aLJT performs the same steps as LJT for an initial input. With each change, it reuses the current FO jtree, adapts it accordingly, and proceeds adaptively for the next steps until it answers queries again. LVE eliminates all non-query randvars for each query on its input model and the altered models.

This evaluation has two parts. First, we look at runtimes of the individual steps of LJT and aLJT for varying models G of sizes $|G|$ ranging from 2 to 1024 under a model change (adding a parfactor) and an evidence change (adding new evidence). Second, we look at runtimes for G_{ex} under changes, focussing on how fast the programs provide answers again after consecutive changes.

Step-wise Performance This first part of the evaluation looks at runtimes of the individual steps of LJT and aLJT given models of varying size. The model sizes start at 2 and double until they reach 1,024. The first model is $G_2 \cup G_3$ from the FO jtree of G_{ex}. The second model is G_{ex}. For the other models, we basically duplicate the current G, starting with G_{ex}, rename the PRVs and logvars of the duplicate, and connect the original part with the copied part through a parfactor. The largest model has 1,024 parfactors and logvars and 3,072 PRVs, resulting in an FO jtree with 770 parclusters. The largest parcluster contains 256 PRVs. Technical remark: The maximum parcluster size is larger than required due to the heuristics the FO dtree construction is based on, cf. (Taghipour *et al.*, 2013a). The largest parcluster contains all parameterless PRVs, because the heuristics leads the (a)LJT implementations to handle all parfactors without logvars separately at the beginning, resulting in one large parcluster as the parameterless PRVs also appear in all other parts of the model.

The domain sizes for all logvars are set to 1,000, leading to grounded model sizes, ranging from 1,001,000 to 513,256,256. A part of the model receives evidence for 50% of the instances of one PRV. We compare runtimes of the corresponding LJT and aLJT step for the following settings: (i) Add a parfactor with a new PRV. (ii) Reenter known evidence



(a) Construction (adding a parfactor)

(b) Evidence entering

(c) Message passing

(d) From changed input to first answer

Figure 11.4: Runtimes [ms] of the (a)LJT steps (y-axis in Fig. 11.4b varies). X-axis: increasing $|G|$ from 2 to 1,024. Both axes are log-scaled. Filled symbols show runtimes under model changes, hollow symbols show runtimes under evidence changes.

after model changes and enter new evidence in an unchanged model. (iii) Pass messages after changes in a model and pass messages after changes in evidence. (iv) Answer a query starting from a model change and from an evidence change.

Figure 11.4 shows the runtimes of both programs averaged over five runs for the four settings. The triangles indicate LJT, while the circles mark aLJT. The filled turquoise marks refer to steps handling changes in a model. The hollow orange marks refer to steps handling changes in evidence. The curves have a similar shape when setting the domain sizes to other values than 1,000. Figure 11.4a shows runtimes of construction. The aLJT program is two to three orders of magnitude faster than the LJT program, with the model change having only a local effect. Figure 11.4b shows the evidence entering runtimes. For both scenarios (reentering known evidence and entering new evidence), aLJT is faster than LJT by three orders of magnitude since both changes have only a local effect.

Figure 11.4c shows the message passing runtimes. The first six models allows for aLJT to be one order of magnitude faster than LJT. Even though aLJT only has to compute

roughly half of the messages that LJT has to compute, runtimes of both programs almost align, with aLJT being only a factor of 0.87 faster than LJT given the largest model. The reason for the similar runtimes lies in aLJT only being able to save calculating messages that are very easy to calculate (small parcluster, few eliminations). But, the program still has to compute the messages that take many eliminations. E.g., in the largest model, there are 512 messages that each take 255 eliminations, while the remaining messages take only 1 or 2 eliminations. Of the 512 costly messages, aLJT still has to compute 511. Figure 11.4d shows runtimes over three or four steps accumulated, namely from construction to query answering of one query when handling adding a parfactor and from evidence entering to query answering when handling new evidence. Since message passing dominates in the overall performance of (a)LJT with only one query, the runtimes in Fig. 11.4d resemble the runtimes in Fig. 11.4c.

Given the unbalanced FO jtrees of the example models, the payoff is strongest for models with a size of up to $|G| = 64$, which means a grounded size of 32,016,016. The aLJT runtimes are faster up to one order of magnitude. If changes only have a local effect, adaptive construction and evidence entering are up to three orders of magnitude faster. Overall, the savings add up under consecutive changes, which we look at next.

Consecutive Changes This second part of the evaluation looks at how fast the programs provide answers again. As input, we use G_{ex} with random potentials. We set $|\mathcal{D}(X)| =$ 1,000 and $|\mathcal{D}(.)| = 100$ for the other logvars, yielding $|gr(G_{ex})| = 111{,}001$. Evidence occurs for 200 instances of $Sick(X)$ with the value 1. There are two queries, $Sick(x_{1000})$ and $Treat(x_1, m_1)$. We apply three consecutive changes,

(i) adding a parfactor $\phi(Epid, Sick(X), Work(X))$ (referred to as model G^1_{ex}),

(ii) replacing parfactor g_2 with a parfactor $\phi(Sick(X), Travel(X))$ (model G^2_{ex}), and

(iii) adding evidence for 100 instances of $Work(X)$ with value *true* (model G^3_{ex}). The X values are a subset of the X values in the $Sick(X)$ evidence, leading to one additional split per affected parfactor due to evidence.

After each change, the programs answer both queries. We compare runtimes for inference averaged over five runs. Runtimes for LJT and aLJT include construction, evidence entering, message passing, and query answering.

Figure 11.5 shows runtimes in seconds accumulated over all four models with two queries each for LVE (crosses), LJT (circle), and aLJT (squares). The vertical lines indicate when the programs have answered both queries, after which LVE and LJT proceed with the next model, while aLJT starts with adaption. For a model, the points on the LJT and aLJT lines mark when an individual step is finished. As expected, LVE takes longer than LJT and aLJT, showcasing the advantage of using an FO jtree. After only two queries, LJT and aLJT provide answers faster than LVE.

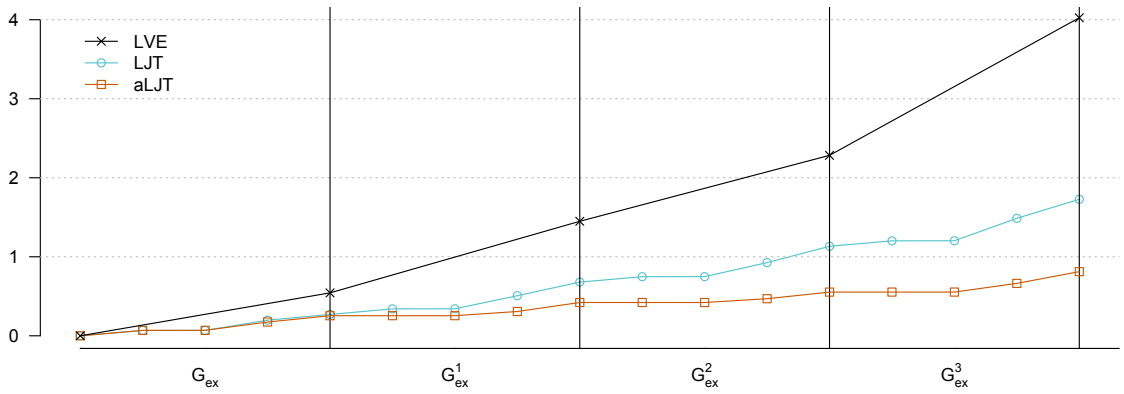


Figure 11.5: Runtimes [s] accumulated over a model with three consecutive adaptions, two queries each. Vertical lines indicate that QA is finished for the current model.

For G_{ex}, LJT and aLJT have the same runtimes since their runs are identical. As G_{ex} incrementally changes, aLJT displays its advantage of adaptive steps over LJT. Starting with G^1_{ex}, aLJT provides answers faster than LJT. Before LJT has completed message passing, aLJT has already answered both queries. Especially message passing is faster as aLJT only has to compute half of the messages that LJT computes. Evidence entering does not take long. But, evidence usually leads to longer runtimes for query answering compared to no evidence for LVE and LJT as the necessary splits lead to larger models. Since G^3_{ex} contains more evidence, all runtimes increase compared to the previous models.

aLJT fast reaches the point of answering queries again, providing answers to queries more timely than the other two programs. Since each change provides the possibility for aLJT to save computations, leading to savings in execution time, the savings add up over a sequence of changes. Thus, performing adaptive inference pays off, even more so with a sequence of changes. In summary, our empirical setup supports that performing adaptive inference pays off as aLJT is able to provide a faster online QA than LJT.

11.6 Interim Conclusion: Adaptive Inference with LJT

Algorithms for adaptive inference aim at answering queries more efficiently than starting from scratch after incremental changes in an input. Changes in a model arise gradually if the underlying scenario remains the same. In lifted inference, changes no longer only pertain to relations or potentials but also to domains. Domains change if the set of individuals they represent change. E.g., if a model includes employees or network components, changes in staff or a network topology lead to changed domains. If requiring fast online QA and having domains that naturally fluctuate, adaptive steps enable aLJT to quickly reach the point of answering queries again.

An ongoing learning task on a database may output updated potentials if new data comes in. Reinforcement learning regularly updates potentials. New data in a database may also lead to altered parfactors or new ranges for PRVs. A learning task yields more suitable ranges or derives new model components where the adjusted model allows for modelling a given scenario more accurately. An adaptive construction avoids the costs for rebuilding an FO jtree, which can be expensive for large models.

Evidence usually occurs for the same PRVs as certain parts of a model are observable while others are latent. Instead of new evidence for varying PRVs, there often are new observations for the same PRVs. The instances within a PRV that receive evidence may vary, which causes additional or retracted evidence. Such a scenario happens if sensors fail or data transfers are interrupted.

This chapter presents aLJT, an adaptive version of LJT, which incorporates incremental changes in its input model or evidence efficiently. We specify how to adapt an FO jtree when deleting, adding, or replacing parts of a model. We formalise under which conditions evidence entering and new messages are necessary. Given the adaptive steps, aLJT reduces its static overhead for construction and message passing under gradual changes compared to LJT. aLJT allows for fast online inference for answering multiple queries, minimising the lag in query answering when inputs change.

We currently work on learning lifted models, where we use aLJT as a subroutine to propagate updated potentials throughout the FO jtree. We also plan to involve aLJT in solving an MEU problem where one is interested in those actions that lead to a maximum expected utility. Solving an MEU problem requires calculating an expected utility for varying action settings. We have defined the MEU problem for parameterised models (Gehrke *et al.*, 2019e) and show how one may calculate an expected utility with LVE. The LJT variant for MEU allows for calculating expected utilities as well as answering probability queries. aLJT could handle varying action settings as part of evidence to efficiently calculate expected utilities.

Another possible use for aLJT is when arguing with value of information: If based on some value of information or on the trust one has in given evidence, certain parts of a model become (ir)relevant, aLJT can quickly update messages. aLJT also provides an opening into approximate inference regarding marking a message only as changed when the distribution represented by the message diverges from the distribution of old message by more than some ϵ value.

With aLJT, we have covered Contribution (6) regarding adaptive inference on FO jtrees. As described above, aLJT allows for various applications in different contexts. The final chapter of Part III looks at LJT in the form of a framework, requiring another inference algorithm to perform calculations during its steps, and discusses how FOKC can be a part of LJT.

Chapter 12

LJT as a Backbone for Lifted Inference

For certain inputs, LVE, LJT, and FOKC start to struggle either due to model structure or size. The implementations of LVE and, as a consequence, LJT ground parts of a model if randvars of the form $Q(X), Q(Y), X \neq Y$ appear, where parameters X and Y have the same domain, even though in theory, LVE handles those occurrences through counting and merge-counting. But not only implementation-wise does LVE struggle in this scenario. As seen in Chapter 6, count conversions strike down on runtimes of LVE and LJT, while FOKC does not show such an increase in runtimes. But, FOKC can struggle if the model size increases as FOKC requires an input model to be in a normal form, which may grow exponentially large and result in prohibitively large circuits.

The purpose of this chapter is to prepare LJT as a backbone for lifted QA to use any exact inference algorithm as a subroutine. Thereby, we are able to select different algorithms as a subroutine to cater towards a given input model. Using FOKC and LVE as subroutines, we fuse LJT, LVE, and FOKC to compute answers faster than LJT, LVE, and FOKC alone for the inputs described above. We first presented this idea in the following two papers. The second paper, published on a conference, is a slightly shorter version of first one, published on a workshop.

> Tanya Braun and Ralf Möller. Fusing First-order Knowledge Compilation and the Lifted Junction Tree Algorithm. In *8th International Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence*, 2018

> Tanya Braun and Ralf Möller. Fusing First-order Knowledge Compilation and the Lifted Junction Tree Algorithm. In *Proceedings of KI 2018: Advances in Artificial Intelligence*, pages 24–37. Springer, 2018

The remainder of this chapter provides the last contribution of this dissertation (Contribution 7). We begin with presenting conditions for subroutines of LJT and discuss how LVE works in this context and FOKC as a candidate. Then, we fuse LJT, LVE, and FOKC into LJTKC. We discuss the (non)-effects on correctness and complexity and show the promise of combining different algorithms in a short empirical evaluation. We conclude with future work, which also concludes this third part.

12.1 The LJT Framework

LJT provides general steps for efficient QA given a set of queries. It constructs an FO jtree and uses a subroutine to calculate messages and answer queries. To ensure a lifted algorithm run without groundings, evidence entering and message passing impose some requirements on the algorithm used as a subroutine. After presenting these requirements, we analyse how LVE matches the requirements and describe to what extent FOKC can provide the same service as LVE.

Requirements LJT has a domain-lifted complexity, meaning that if a model allows for computing a solution without grounding part of a model, LJT is able to compute the solution without groundings, i.e., has a complexity polynomial in the domain size of the logvars. Given a model that allows for computing solutions without grounding part of a model, the subroutine must be able to handle message passing and query answering without grounding to maintain the domain-lifted complexity of LJT.

Evidence displays symmetries if observing the same value for n instances of a PRV. Thus, for evidence handling, the algorithm needs to be able to handle a set of observations for some instances of a single PRV in a lifted way. Calculating messages entails that the algorithm is able to calculate a form of parameterised, conjunctive query over the PRVs in the separator. In summary, LJT requires the following:

(i) Given evidence in the form of a set of observations for instances of a single PRV, the subroutine must be able to absorb evidence independent of the size of the set.

(ii) Given a parcluster with its local model, messages, and a separator, the subroutine must be able to eliminate all PRVs in the parcluster that do not appear in the separator in a domain-lifted way.

The subroutine also establishes which kind of queries LJT can answer. The expressiveness of the query language for LJT follows from the expressiveness of the inference algorithm used. If an algorithm answers queries that include a single randvar, LJT answers this type of query. If an algorithm is able to answer MAP queries, LJT is able to answer MAP queries. Next, we look at how LVE fits into LJT.

LVE as a Subroutine LVE as a subroutine provides lifted absorption for evidence handling. Lifted absorption splits a parfactor into one part, for which evidence exists, and one part without evidence. The part with evidence then absorbs the evidence by absorbing it once and exponentiating the result for all isomorphic instances if a logvar is eliminated during absorption.

For messages, LVE eliminates all non-query terms, without inducing a joint distribution, multiplying the remaining parfactors, and normalising the result. The original LVE from Section 3.2 answers queries for marginal or conditional distributions of a sin-

gle ground query term. LJT with this LVE as a subroutine answers the same queries. Extensions to LVE enable more query types, such as parameterised conjunctive queries or MPE queries as shown in previous chapters.

FOKC as a Subroutine Candidate Regarding FOKC as a subroutine for LJT, FOKC does not fulfil all requirements. FOKC can handle evidence through conditioning (Van den Broeck and Davis, 2012). But, a lifted message passing is not possible in a domain-lifted and exact way without restrictions. Using an FO d-DNNF circuit, FOKC answers queries for a probability of a single randvar as specified by Van den Broeck *et al.* (2011) and a conditional distribution for a single randvar given events (Van den Broeck and Davis, 2012). Inherently, conjunctive queries are only possible if the conjuncts are probabilistically independent (Darwiche and Marquis, 2002), which is rarely the case for separators. Otherwise, FOKC has to invest more effort to take into account that the probabilities overlap. Thus, the restricted query language means that LJT cannot use FOKC for message calculations in general. Given an FO jtree with singleton separators, message passing with FOKC as a subroutine may be possible. FOKC as such takes ground queries as input or computes answers for random groundings, so FOKC for message passing needs an extension to handle parameterised queries.

Given a submodel of a local model and appropriate messages, FOKC may answer queries if the parfactors do not contain CRVs. So, even though FOKC may not fulfil all requirements to become a subroutine during message passing, we can combine LJT, LVE, and FOKC into one algorithm, using LVE for evidence handling and message passing and FOKC for query answering, to answer queries for models where LJT with LVE as a subroutine struggles or where FOKC struggles because of model sizes.

12.2 LJTKC: Fusing LJT, LVE, and FOKC

Using LJT as a backbone and LVE and FOKC as subroutines, LJTKC is a fusion of LJT, LVE, and FOKC. Algorithm 16 shows LJTKC with a model G, a set of queries $\{Q_k\}_{k=1}^m$, and evidence $\mathbf{E}$ as inputs. Due to FOKC, each query has a single ground query term Q_k in contrast to a set of query terms $\mathbf{Q}_k$ in the previous chapters. As a consequence, LJTKC has the same expressiveness regarding the query language as FOKC.

The first three steps of LJTKC coincide with LJT as specified in Alg. 3: LJTKC builds an FO jtree J for G, enters $\mathbf{E}$ into J, and passes messages in J using LVE for message calculations. During evidence entering, each local model covering evidence randvars absorbs evidence. LJTKC calculates messages based on local models with absorbed evidence, making each parcluster independent of the remaining model given the messages at each parcluster. The local models and messages are sufficient to answer queries for randvars contained in $\mathbf{C}_i$ and remain valid as long as G and $\mathbf{E}$ do not change. At this point, FOKC starts to interlace with LJT.

Algorithm 16 LJTKC

procedure LJTKC(Model G, Queries $\{Q_k\}_{k=1}^m$, Evidence $\mathbf{E}$)
  Construct an FO jtree $J = (V, E)$ for G
  Enter $\mathbf{E}$ into J
  Pass messages on J ▷ LVE as subroutine
  for each parcluster $\mathbf{C}_i \in V$ with local model G_i **do**
    Form submodel $G' \leftarrow G_i \cup \bigcup_{j \in nbs(i)} m_{ij}$
    Reduce G' to WFOMC problem with Δ_i, w_T^i, w_F^i
    Compile a circuit $\mathcal{C}_i$ for Δ_i
    Compute $c_i = WFOMC(\mathcal{C}_i, w_T^i, w_F^i)$
  for each $Q_k \in \{Q_k\}_{k=1}^m$ **do**
    Find $\mathbf{C}_i \in V$ s.t. $Q_k \in \mathbf{C}_i$
    Compile a circuit $\mathcal{C}_q$ for Δ_i, Q_j
    Compute $c_q = WFOMC(\mathcal{C}_q, w_T^i, w_F^i)$
    Compute $P(Q_j|\mathbf{E}) = c_q/c_i$ ▷ Output or store result

LJTKC continues its preprocessing. For each parcluster $\mathbf{C}_i$, LJTKC extracts a submodel G' of local model G_i and all messages received and reduces G' to a WFOMC problem with theory Δ_i and weight functions w_F^i, w_T^i. LJTKC does not need to incorporate $\mathbf{E}$ as the information from $\mathbf{E}$ is contained in G' through evidence entering and message passing. LJTKC compiles an FO d-DNNF circuit $\mathcal{C}_i$ for Δ_i and computes a WFOMC c_i on $\mathcal{C}_i$. In precomputing a WFOMC c_i for each parcluster, LJTKC uses that the denominator of Expression (3.8) is identical for varying queries on the same model and evidence. For each query handled at $\mathbf{C}_i$, the submodel consists of G', resulting in the same circuit $\mathcal{C}_i$ and WFOMC c_i.

To answer a query Q_k, LJTKC finds a parcluster $\mathbf{C}_i$ that covers Q_k and compiles an FO d-DNNF circuit $\mathcal{C}_q$ for Δ_i and Q_j. It computes a WFOMC c_q in $\mathcal{C}_q$ and determines an answer to $P(Q_j|\mathbf{E})$ by dividing the just computed WFOMC c_q by the precomputed WFOMC c_i of this parcluster. LJTKC reuses Δ_i, w_T^i, and w_F^i from preprocessing. Let us look at G_{ex} with evidence $sick(eve)$ to see how LJTKC works.

Example 12.2.1. For G_{ex}, LJTKC builds an FO jtree as depicted in Fig. 4.1. Entering $sick(eve)$ as evidence in the FO jtree leads to local models as depicted in Fig. 4.3. LJTKC sends messages from parclusters $\mathbf{C}_1$ and $\mathbf{C}_3$ to parcluster $\mathbf{C}_2$ and back. The messages are depicted in Fig. 4.4. For message m_{12}, LJTKC eliminates $Nat(D)$ and $Man(W)$ from G_1 using LVE. For message m_{32}, LJTKC eliminates $Treat(X, M)$ from G_3 using LVE. For the messages back, LJTKC eliminates $Travel(X)$ and $Sick(X)$ from $G_2 \cup m_{32}$ for message m_{21} and $Travel(X)$ from $G_2 \cup m_{12}$ for message m_{23}.

The local model and received messages at each parcluster form the submodels for the compilation steps. For each parcluster, LJTKC reduces the submodel to a WFOMC

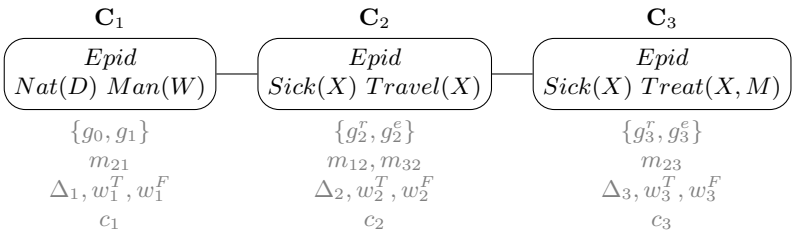


Figure 12.1: FO jtree after preprocessing with LJTKC with local information in grey

problem, compiles a circuit for the problem specification, and computes a parcluster WFOMC. Figure 12.1 shows the FO jtree after preprocessing is complete.

Given a query term $Treat(eve, injection)$, LJTKC takes a parcluster that contains the query term, here $\mathbf{C}_3$. It compiles a circuit for the query term and $\mathbf{\Delta}_3$, computes a query WFOMC c_q, and divides c_q by c_3 to determine $P(treat(eve, injection))$. $P(\neg treat(eve, injection))$ is then given by $1 - \frac{c_q}{c_3}$.

Instead of compiling one large model into a circuit, FOKC compiles three smaller circuits for query answering. As one can see even in Fig. 12.1, LJTKC needs more memory than LJT as local models and messages as well as theories, weight functions, and counts need to be stored. But, one can now choose a QA algorithm, either LVE or FOKC, to answer a query at hand. Next, we briefly dive into a theoretical discussion and then into an empirical evaluation.

12.3 Theoretical Discussion

We show soundness for LJTKC and discuss the differences between LJT, LVE, FOKC, and LJTKC regarding memory and runtime.

Theorem 12.3.1. *LJTKC is sound, i.e., computes a correct result for a query Q given a model G and evidence $\mathbf{E}$.*

Proof. As shown in Section 5.1, LJT is sound, yielding an FO jtree J for model G, which fulfils the three junction tree properties, which allow for local computations (Shenoy and Shafer, 1990). Further, we assume that LVE is correct, ensuring correct computations for evidence entering and message passing, (Taghipour *et al.*, 2013c) and that FOKC is correct (Van den Broeck, 2013), computing correct answers for single term queries.

LJTKC coincides with LJT in the first three steps, construction, evidence entering, and message passing, i.e., the first three steps of LJTKC are sound. After message passing, each parcluster holds G and $\mathbf{E}$ combined in its local model and received messages, which allows for answering queries for randvars that the parcluster contains. At this point, the FOKC part takes over, taking all information present at a parcluster and compiling a

circuit and computing a WFOMC, which produces correct results given FOKC is correct. The same holds for the compilation and computations done for query Q. Thus, LJTKC computes a correct result for Q given G and $\mathbf{E}$. □

Complexity We discuss memory and runtime performance of LJT, LVE, FOKC, and LJTKC in comparison with each other.

LJT requires *memory* for storing an FO jtree with messages, while FOKC takes up space for storing circuits. As a combination of LJT and FOKC, LJTKC stores the preprocessing information produced by both LJT and FOKC. Next to the FO jtree structure and messages, LJTKC stores a WFOMC problem specification and a circuit for each parcluster. If LJTKC does not use the messages again after forming a WFOMC problem, it could drop the messages to free up space. Since LJTKC sometimes avoids large intermediate results using FOKC, memory requirements during QA are smaller than for LJT. LJTKC stores more circuits than FOKC but the individual circuits are smaller and do not require conditioning, which avoids a significant blow-up for circuits.

LJTKC accomplishes speeding up QA for certain challenging inputs by fusing LJT, LVE, and FOKC. The new algorithm has a faster *runtime* than LJT, LVE, and FOKC as it is able to precompute reusable parts and provide smaller models for answering a specific query using an FO jtree. In comparison with FOKC, LJTKC speeds up runtimes as answering queries works with smaller models. In comparison with LJT and LVE, LJTKC is faster when avoiding groundings in LVE and when many count conversions with large domain sizes are necessary. Instead of precompiling each parcluster, which adds to its overhead before starting with answering queries, LJTKC could compile on demand. On-demand compilation means less runtime and space required in advance but more time per initial query at a parcluster. One could further optimise LJTKC by speeding up internal computations in LVE or FOKC (e.g., caching for message calculations or pruning circuits using context-specific information).

In terms of *complexity*, LVE and FOKC have a time complexity polynomial w.r.t. the domain sizes of the model logvars for liftable models. LJT with LVE as a subroutine also has a time complexity polynomial in terms of the domain sizes for query answering. For message passing, a factor of n, which is the number of parclusters, multiplies into the complexity, which basically is the same time complexity as answering a single query with LVE. LJTKC has the same time complexity as LJT for message passing since the algorithms coincide. For query answering, the complexity is determined by the FOKC complexity, which is polynomial in terms of domain sizes (Van den Broeck, 2013). Therefore, LJTKC has a time complexity polynomial in terms of the domain sizes. Further, the runtime complexity of FOKC is determined in terms of circuit size, on which FOKC depends linearly (Van den Broeck, 2013), which also holds for LJTKC.

The next section presents an empirical evaluation, showing how LJTKC speeds up QA compared to FOKC and LJT for challenging inputs.

12.4 Empirical Evaluation

This evaluation demonstrates the speed up one can achieve for certain inputs when using LJT and FOKC in conjunction. For LJTKC, we integrate the FOKC implementation by Van den Broeck into the LJT implementation to compute marginals at parclusters. As the FOKC implementation does not handle evidence in a lifted manner and FOKC is part of the evaluation, we do not consider evidence as FOKC runtimes explode. This evaluation also includes JT as a baseline.

The evaluation has two parts: First, we test an input model with inequalities, which theoretically, LVE can handle domain-lifted, but does not handle in its implementation without grounding, to highlight how runtimes of LVE and LJT explode, and how LJTKC provides a speedup. Second, we test a version of the model without inequalities to highlight how runtimes of LVE and LJT compare to FOKC without inequalities.

We compare overall runtimes (preprocessing and query answering, answering each possible representative query) without input parsing averaged over five runs with a working memory of 16GB. LVE eliminates all non-query randvars from its input model for each query, grounding in the process. LJT builds an FO jtree for its input model, passes messages, and then answers queries on submodels. FOKC forms a WFOMC problem for its input model, compiles a model circuit, compiles for each query a query circuit, and computes the marginals of all PRVs in the input model with random groundings. LJTKC starts like LJT for its input model until answering queries. It then calls FOKC at each parcluster to compute marginals of parcluster PRVs with random groundings. JT receives the grounded input models and otherwise proceeds like LJT.

Inputs with Inequalities For the first part of this evaluation, we test a model G_l. Model G_l has 4 logvars, 12 PRVs with one or two parameters, and 20 parfactors with 1 to 3 arguments. Two logvars X and Y have the same domain and one parfactor contains an inequality of the form $X \neq Y$. The FO jtree for G_l has six parclusters, the largest one containing five PRVs. We vary the domain sizes from 2 to 1000, resulting in $|gr(G_l)|$ from 52 to 8,010,000. We query each PRV grounded randomly, leading to 12 queries.

Figure 12.2 shows for G_l runtimes in milliseconds [ms] with increasing $|gr(G_l)|$ on log-scaled axes for FOKC (circle), JT (star), LJT (filled square), LJTKC (hollow square), and LVE (triangle) (points are connected for readability). The JT runtimes are much longer with the first setting than the other runtimes. Up to the third setting, LVE and LJT perform better than FOKC with LJT being faster than LVE. From the seventh setting on, memory errors occur for both LVE and LJT. LJTKC performs best from the third setting onwards. LJTKC and FOKC show the same steady increase in runtimes. LJTKC runtimes have a speedup of a factor from 0.13 to 0.76 for G_l compared to FOKC. Up to a domain size of 100 ($|gr(G_l)| = 81{,}000$), LJTKC saves around one order of magnitude.

For small domain sizes, LJTKC and FOKC perform worst. With increasing domain sizes, they outperform the other programs. Though not part of the numbers in this

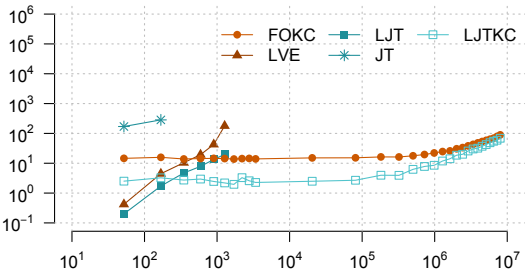


Figure 12.2: Runtimes [ms] for G_l; on x-axis: $|gr(G_l)|$ from 52 to 8,010,000

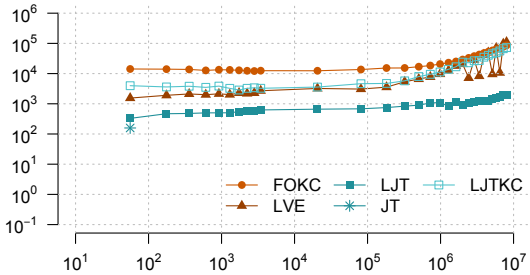


Figure 12.3: Runtimes [ms] for G'_l; on x-axis: $|gr(G'_l)|$ from 56 to 8,012,000

evaluation, with an increasing number of parclusters, LJTKC promises to outperform FOKC even more, especially with smaller domain sizes. The same behaviour of LJTKC compared to FOKC, LVE, and LJT can be observed when looking at runtimes for models requiring many count conversions. Compared to LVE and LJT, LJTKC avoids large intermediate results after count conversions. Compared to FOKC, LJTKC exploits the sparseness of the model, allowing for smaller circuits.

Inputs without Inequalities For the second part of this evaluation, we test an input model G'_l, that is the model from the first part but with Y receiving an own domain as large as X, making the inequality superfluous. Domain sizes vary from 2 to 1000, resulting in $|gr(G'_l)|$ from 56 to 8,012,000. Figure 12.3 shows for G'_l runtimes [ms] with increasing $|gr(G)|$ set up as before.

JT is the fastest for the first setting. With the following settings, JT runs into memory problems while runtimes explode. LVE and LJT do not exhibit the runtime explosion without inequalities. LVE has a steadily increasing runtime for most parts, though a few settings lead to shorter runtimes with higher domain sizes. We could not find an explanation for the decrease in runtime for those handful of settings. Overall, LVE runtimes rise more than the other runtimes apart from JT. LJTKC exhibits an unsteady runtime performance on the smaller model, though again, we could not find an explanation for the jumps between various sizes. With the larger model, LJTKC shows a more steady performance that is better than the one of FOKC. LJTKC is a factor of 0.2 to 0.8 faster. FOKC and LJT runtimes steadily increase with rising $|gr(G)|$. LJT gains over an order of magnitude compared to FOKC. In the larger model, LJT is a factor of 0.02 to 0.06 faster than FOKC over all domain sizes. LJTKC does not perform best as the overhead introduced by FOKC does not pay off as much for this model without inequalities. In fact, LJT performs best in almost all cases.

In summary, without inequalities LJT performs best on our input models. Though, LJTKC does not perform worst, LJT performs better and steadier. With inequalities, LJTKC exhibits a better runtime performance.

12.5 Interim Conclusion: LJTKC

This chapter has presented LJT as a backbone for lifted inference. We have determined conditions for inference algorithms to become a subroutine for LJT, while keeping the domain-lifted nature of LJT. The conditions require a subroutine to be able to handle evidence in a lifted manner and to compute messages through a form of parameterised query. The expressiveness of the query language of the subroutine during query answering then determines the expressiveness of the query language of LJT. LJT provides a means to cluster a model into submodels, on which any exact lifted inference algorithm can answer queries given the algorithm can handle evidence and messages in a lifted way.

We have implemented the LJT framework using LVE for evidence and messages as well as FOKC for queries, calling the resulting algorithm LJTKC. As both LVE and FOKC struggle given certain inputs, due to size or limited implementations, LJTKC alleviates certain problems by allowing for FOKC as a subroutine for query answering. FOKC fused with LJT and LVE can handle larger models more easily. In turn, FOKC boosts LJT by avoiding large intermediate results in certain cases. LJTKC enables us to compute answers faster than LJT with LVE for certain inputs and LVE and FOKC alone.

We currently work on incorporating FOKC into message passing for cases where a problematic elimination occurs during message calculation, which includes adapting an FO jtree accordingly. Additionally, we examine FOKC further to possibly extend it for parameterised queries with a possible use during message passing as well as query answering as in Chapter 9. Further work includes investigating other lifted QA algorithms, e.g., probabilistic theorem proving (Gogate and Domingos, 2011), or even allow for sampling algorithms (e.g., Gogate and Domingos, 2011; Niepert, 2012, 2013) as a subroutine.

With this look at ongoing work, this last chapter of Part III covering Contribution (7), LJT as a framework for QA algorithms, ends. Lifted inference as discussed requires a known universe, which manifests itself in LVE for example through conditional counts during lifted summing out. The upcoming chapter ventures into unknown universes, discussing how inference is possible without knowing the inhabitants of a universe.

Chapter 13

Outlook: Unknown Universes

> Space, the final frontier.
> These are the voyages of the Starship Enterprise.
> Its five year mission:
> To explore strange new worlds.
> To seek out new life.
> And new civilisations.
> To boldly go where no one has gone before.
>
> *(James T. Kirk)*

Parameterised models contain logvars that have domains and constraints that restrict logvars to certain constants in a parfactor. The constraints are significant as they allow LVE to determine conditional counts that LVE needs to check preconditions of operators as well as to complete a lifted summing out by taking a resulting potential to the power of such a count. Hence, LVE relies on constraints for its operation, and the constraints are populated with constants from a known universe. So, what happens if the universe is unknown, i.e., the domains are unspecified, making constraints empty?

LVE as specified above does not work with empty constraints. Even a $\top$ constraint, which we have often omitted in this dissertation, is only a shorthand notation for a Cartesian product of domains, which are now empty. This is not an entirely new research question and an interesting one for diverse research areas: Ceylan et al. define semantics for an open-world probabilistic database (PDB), keeping a fixed upper bound on domains Ceylan *et al.* (2016). Srivastava et al. specify first-order open-universe partially observable Markov decision processes to generate strategies based on sampling Srivastava *et al.* (2014). Regarding a slightly different problem, Singla and Domingos present how to work with infinite domains in MLNs Singla and Domingos (2007). Max-entropy semantics allow for specifying local probability distributions partially, distributing the remaining probability over the unspecified portion of the distribution equally, which still requires some knowledge about existing constants and leads to more complex operators for calculations (Thimm *et al.*, 2010). But, lifted inference given *unknown* finite universes has no satisfying solution.

Therefore, we explore lifted inference given unknown universes in this last chapter before the conclusion of this thesis. The aim is to define a semantics that allows for easily

transferring a model from one domain to the next. We first consider the constraints more closely, detaching them from a known universe and instead providing a generative description. Second, we examine how to define semantics for models with such a constraint description combined with a distribution over possible domains. Additionally, we consider how to restrict the number of universes generated for feasibility. From the semantics, it follows how QA works and what kind of new queries arise. Although the idea behind our approach is applicable to any lifted formalism and inference algorithm, we consider parameterised models in combination with LVE and LJT since the constraint language has been decoupled from the inference algorithms Taghipour *et al.* (2013c). This decoupling makes it possible to consider domains, constraints, and QA separately, which allows for streamlining the approach. A conference paper about semantics and unknown universes has been accepted Braun and Möller (2019).

13.1 Template Models

Parameterised models contain constraints that restrict logvars in a parfactor to constants from a known universe. Without a known universe, the set of constants $\mathbf{D}$ becomes empty. As a consequence, logvar domains no longer exist as the domains are defined as subsets of $\mathbf{D}$. In turn, constraints are no longer defined since they are combinations of subsets of domains. Last, semantics lose its meaning as it involves grounding a model, which is not possible without constraints. We assume, though, that the model itself accurately describes relations. Thus, a parameterised model without $\mathbf{D}$ becomes a template model that specifies local distributions for unknown instances of PRVs.

Definition 13.1.1 (Template model)**.** A *template model* $\mathcal{G}$ is a set of parfactors $\{\tilde{g}_i\}_{i=1}^n$, in which each $\tilde{g}_i = \phi_i(\mathcal{A}_i)_{|C}$ has an *empty* constraint $C = (\mathcal{X}, C_\mathcal{X})$ with $C_\mathcal{X} = \bot$.

For ease of exposition, assume a slightly smaller example model, namely, G_{ex} without g_1, reducing the logvars to X and M. which we name G_{un}.

Example 13.1.1 (Template model)**.** $G_{un} = \{g_0, g_2, g_3\}$ becomes template model $\mathcal{G}_{un}$ by replacing the constraints in g_2 and g_3 with $((X), \bot)$ and $((X, M), \bot)$, respectively.

Next, we fill empty constraints in template models with tuples by generating them using a so-called constraint program.

13.2 Worlds of Constraints

With an unknown universe, we implicitly specify constraints through a set of rules that generates tuples for constraints given a specific domain at a later point. To model constraints, one could use, e.g., description logic or probabilistic Datalog Fuhr (1995), with the latter leading to probabilities associated with constraints.

Definition 13.2.1 (Constraint program, constraint world)**.** Given a template model $\mathcal{G}$ and a specific domain set D for the logvars in $\mathcal{G}$, a *constraint program* $\mathcal{C}$ returns a set of ordered constraint sets $\mathbf{C} = \{\{C_{j,i}\}_{i=1}^{n}\}_{j=1}^{m}$ or, if a probability p_j exists for each j, $\mathbf{C} = \{(\{C_{j,i}\}_{i=1}^{n}, p_j)\}_{j=1}^{m}$, i.e., $\mathcal{C}$ generates a constraint for each parfactor in $\mathcal{G}$. We call each generated constraint set $CW_j = \{C_{j,i}\}_{i=1}^{n}$ a *constraint world*. Instantiating $\mathcal{G}$ with CW, i.e., replacing the empty constraints in $\mathcal{G}$ with the constraints in CW, leads to a parameterised model, denoted by $G_{|CW}$, which follows distribution semantics.

Let us consider two constraint programs, $\top$ constraints as well as a Datalog program, to generate constraint worlds for $\mathcal{G}_{un}$.

Example 13.2.1 (Constraint program, constraint worlds)**.** The shorthand $\top$ already defines a constraint program $\mathcal{C}^{\top}$ that generates tuples by building the Cartesian product given specific domains. It also generates exactly one constraint world. Given $\mathcal{G}_{un}$, $\mathcal{C}^{\top}$ returns $\{\{C_2, C_3\}\}$ if D contains the domains $\mathcal{D}(X) = \{alice, bob, eve\}$ and $\mathcal{D}(M) = \{injection, tablet\}$. For a more complex example, assume that there are three possible treatments t_1, t_2, t_3 with only two treatments applicable at a time, i.e., only X is unknown. Each combination has a different probability, e.g., 0.7 for (t_1, t_2), 0.3 for (t_2, t_3), and 0.2 for (t_1, t_3). The following Datalog program captures this setup:

```
element_of_C3(X,Y1) :- linked(X,Y1,Y2).
element_of_C3(X,Y2) :- linked(X,Y1,Y2).
linked(X,Y1,Y2) :- instance_of_X(X) & pair(Y1,Y2).
0.7 pair(t1,t2).  0.3 pair(t2,t3).  0.2 pair(t1,t3).
```

The first three lines denote rules according to which one can generate (X, M)-tuples. The last line denotes probabilistic facts that are disjoint, with probabilities adding up to 1, to model the combination of treatments. If given a domain such $\{alice, bob, eve\}$ for X, one can add corresponding facts to the program:

```
instance_of_X(alice).    instance_of_X(bob).    instance_of_X(eve).
```

Asking the queries `?- element_of_C3(X, Y)` and `?- instance_of_X(X)` generates tuples for the constraints in $\mathcal{G}_{un}$. Using `0.7 pair(t1, t2)`, the program returns the following facts, which contain tuples for the constraints in $\mathcal{G}_{un}$:

```
instance_of_X(alice).  instance_of_X(bob).  instance_of_X(eve).
0.7 element_of_C3(alice,t1).  0.7 element_of_C3(alice,t2).
0.7 element_of_C3(bob,t1).  0.7 element_of_C3(bob,t2).
0.7 element_of_C3(eve,t1).  0.7 element_of_C3(eve,t2).
```

Arising from the `pair` facts, the Datalog program as constraint program $\mathcal{C}^{DL}$ returns three constraint worlds $\{(\{C_{j,i}\}_{i=1}^{2}, p_j\}_{j=1}^{3}$ with $p_1 = 0.7$, $p_2 = 0.3$, and $p_3 = 0.2$ and

constraints

$$C_{1,2} = C_{2,2} = C_{3,2} = ((X), \{(alice), (bob), (eve)\})$$
$$C_{1,3} = ((X, M), \{(alice, t1), (alice, t2), (bob, t1), (bob, t2), (eve, t1), (eve, t2)\})$$
$$C_{2,3} = ((X, M), \{(alice, t2), (alice, t3), (bob, t2), (bob, t3), (eve, t2), (eve, t3)\})$$
$$C_{3,3} = ((X, M), \{(alice, t1), (alice, t3), (bob, t1), (bob, t3), (eve, t1), (eve, t3)\})$$

Going forward, we use $\mathcal{C}^{DL}$ as the example constraint program for the remaining examples. Using rules in a constraint program is a form of meta-level logic programming, which allows for formulating constraints on constraints without a specific domain. The semantics of a parameterised model still depends on a constraint world, which requires a specific domain. Next, we consider domains.

13.3 Worlds of Domains

Constraint programs still need domains or constants to generate constraint worlds. In unknown universes, these constants are not available. In a naive way, one could generate all possible domains, from one constant for each logvar to infinite domains, leading to infeasibly many possible domains. Given knowledge about the setting in which one wants to reason (like in the example above about treatments t_1, t_2, t_3), one may list all possible domains. Assumptions may further limit the number of worlds: Since logvars require discrete domains of at least one, and small worlds (domains) are usually more likely than large ones, we can set up a discrete distribution over domain sizes.

Definition 13.3.1 (Domain world)**.** Given a template model $\mathcal{G}$, a *domain world DW* is a set of domains $\{\mathcal{D}(X)\}_{X \in lv(\mathcal{G})}$ that specifies a domain for each logvar in $\mathcal{G}$. Given a set of domain worlds $\{DW_k\}_{k=1}^{l}$ and probabilities p_k associated with each DW_k s.t. $\forall k : p_k \in [0, 1]$ and $\sum_k p_k = 1$, then $\mathbf{D} = \{(DW_k, p_k)\}_{k=1}^{l}$ forms a *distribution over domain worlds*. Providing a constraint program $\mathcal{C}$ with a domain world DW_k with probability p_k leads to a set of constraint worlds $\mathbf{C}_k = \{(\{C_{k,j,i}\}_{i=1}^{n}, p_k \cdot p_j)\}_{j=1}^{m}$ in which $p_j = 1$ if $\mathcal{C}$ does not assign probabilities. For $p_k \cdot p_j$ to be correct, p_k and p_j need to be independent (else, replace with an appropriate expression).

One may start with a set of guaranteed constants and add varying numbers of possible constants for domain worlds, inspired by the λ-completion of open-world PDBs Ceylan *et al.* (2016). The probabilities allow for measuring how likely a particular instantiation is compared to others. Given a distribution, one could specify a threshold τ to account only for domains with a probability larger τ, which enables some filtering even before generating parameterised models for efficiency. Another way of restricting the number of worlds is to take domains that lie within the standard deviation from the mean or whose probability make up around 95% of the distribution around its mean or maximum value.

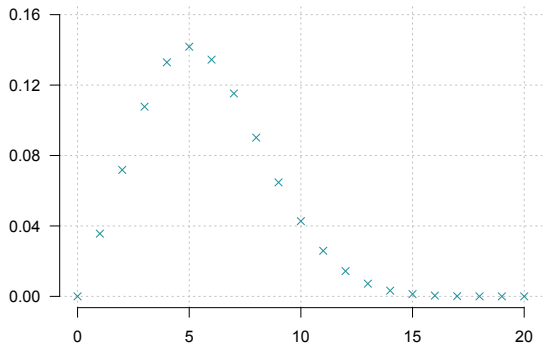


Figure 13.1: Discrete distribution over domain sizes of a logvar

Passing on a domain world to a constraint program $\mathcal{C}$ enables $\mathcal{C}$ to generate constraint worlds to instantiate the template model with. Let us consider an example distribution for a single logvar, namely, X, which is the only unknown quantity given $\mathcal{G}_{un}$ and $\mathcal{C}^{DL}$, and how it allows for generating domain worlds.

Example 13.3.1 (Distribution over domain worlds)**.** Figure 13.1 shows a beta-binomial distribution ($\alpha = 6$, $\beta = 15$). Possible domain sizes go from 1 to 20, which are comparatively small for readability. A domain size of 0 has a probability of 0. This distribution represents the assumptions from before (domains with at least one constant, smaller domains more likely than larger domains). The highest probability lies with a domain size of 5, after which probabilities decrease again. The probability of a domain size of 20 is around $3.85 \cdot 10^{-7}$.

Given $\mathcal{G}_{un}$ and $\mathcal{C}^{DL}$ for generating constraint worlds, assume the distribution just introduced for X, denoted by $p_x(d)$ with d as the input argument referring to the domain size of X. There are 20 domain worlds $\mathbf{D}^{bb} = \{(\{x_i\}_{i=1}^{d}, p_x(d)\}_{d=1}^{20}$ with probabilities $p_x(d)$ between $3.85 \cdot 10^{-7}$ and $1.42 \cdot 10^{-1}$. Given each domain world, $\mathcal{C}^{DL}$ yields three constraint worlds $\{(\{C_{d,j,i}\}_{i=1}^{2}, p_x(d) \cdot p_j)\}_{j=1}^{3}$, i.e., overall 60 constraint worlds, where each constraint world contains a constraint for both $\tilde{g}_1$ and $\tilde{g}_2$. For $d = 3$, the constraints look like the ones given in the previous section but with $x1$, $x2$, and $x3$ instead of *alice*, *bob*, and *eve*. The probabilities of $p_1 = 0.7$, $p_2 = 0.3$, and $p_3 = 0.2$ are multiplied with $p_x(3) = 1.08 \cdot 10^{-1}$.

Some of the 60 constraint worlds in the example have very small probabilities. Hence, one could use a threshold of $\tau = 0.01$ to first restrict the domain worlds in $\mathbf{D}^{bb}$ to those with a probability at least τ as inputs for $\mathcal{C}^{DL}$. Given the distribution of Fig. 13.1, τ would restrict the domains to sizes between 1 and 12, i.e., 12 domain worlds. With $\tau = 0.05$, the set of domain worlds $\mathbf{D}^{bb5}$ contains domains with a size between 2 and 9, which would lead to $8 \cdot 3 = 24$ constraint worlds. One could cascade the filtering and drop constraint worlds if their probability goes below τ as well (or choose a new τ). Given

$\tau = 0.05$ and $\mathbf{D}^{bb5}$ as an input to $\mathcal{C}^{DL}$, the number of constraint worlds goes down to 7, i.e., domain sizes 2 to 8 combined with `0.7 pair(t1,t2)`. The other constraint worlds arising from `0.3 pair(t2,t3)` and `0.2 pair(t1,t3)` have a probability lower than τ. For the distribution of Fig. 13.1, the domains that make up 95% of the distribution around its maximum value go from 1 to 12. Restricting that percentage to, e.g., 50% permits domain sizes between 4 and 8.

With worlds of domains in place, which each in turn leads to worlds of constraints, we define semantics.

13.4 Distribution-based Semantics

To fully specify a model with an unknown universe, we require three components: (i) A *template model* $\mathcal{G}$ provides a structure and local distributions. (ii) A *constraint program* $\mathcal{C}$ generates constraint worlds if given specific domains. A template model can be instantiated with a constraint world, leading to a parameterised model as in Definition 3.1.1, which follows distribution semantics. (iii) A *set of domain worlds* $\mathbf{D}$ specifies (a distribution over) possible domain worlds. Each domain world can be passed to the constraint program, which then generates constraint worlds. Each constraint world and thus each instantiated model has a probability associated. The semantics are defined as follows.

Definition 13.4.1 (Semantics)**.** A model with unknown universe is specified through a template model $\mathcal{G}$, a constraint program $\mathcal{C}$, and domain worlds $\mathbf{D}$. The *semantics* are given by $\mathcal{C}$ generating constraint worlds $\mathbf{C}$ for each $DW \in \mathbf{D}$. Then, a parameterised model $G_{|CW}$ with distribution semantics is instantiated for each $CW \in \mathbf{C}$. The result is a set of parameterised models $\mathbf{G} = \{(G_{|CW}, p)\}_{CW \in \mathcal{C}(\mathcal{G},DW),DW \in \mathbf{D}}$.

Example 13.4.1 (Semantics)**.** The example of the previous section has already shown how many parameterised models derive from a given setting. With $\mathcal{G}_{un}$, $\mathcal{C}^{DL}$, $\mathbf{D}^{bb}$, and cascading filtering with $\tau = 0.05$, the semantics yields seven constraint worlds $\mathbf{C}^{bb5} = \{(\{C_{d,j=1,i}\}_{i=2}^{3}, p_x(d) \cdot p_{j=1})\}_{d=2}^{8}$, leading to a set of parameterised models $\mathbf{G}_{un} = \{(G_{un|CW_{d,1}}, p_x(d) \cdot p_1)\}_{d=2}^{8}$. Each $G \in \mathbf{G}_{un}$ contains parfactors g_0, g_2, g_3 with the following signatures and identical mappings (omitted).

$$
\begin{aligned}
g_0 &= \phi_0(Epid), \\
g_2 &= \phi_1(Epid, Sick(X), Travel(X))_{C_{d,1,2}}, \\
g_3 &= \phi_2(Epid, Sick(X), Travel(X))_{C_{d,1,3}}.
\end{aligned}
$$

Constraints $C_{d,1,2}$ and $C_{d,1,3}$ as well as the associated probability differ between the models. For $d = 2$, the probability is $7.18 \cdot 10^{-2} \cdot 0.7$ and the constraints are

$$
\begin{aligned}
C_{2,1,2} &= ((X), \{(x_1)\}), \\
C_{2,1,3} &= ((X, M), \{(x_1, t_1), (x_1, t_2)\}).
\end{aligned}
$$

For $d = 8$, the probability is $9.02 \cdot 10^{-2} \cdot 0.7$ and the constraints are

$$C_{8,1,2} = ((X), \{(x_1), (x_2), (x_3), (x_4), (x_5), (x_6), (x_7), (x_8)\})$$
$$C_{8,1,3} = ((X, M), \{(x_1, t_1), (x_1, t_2), (x_2, t_1), (x_2, t_2), (x_3, t_1), (x_3, t_2), (x_4, t_1), (x_4, t_2), (x_5, t_1), (x_5, t_2), (x_6, t_1), (x_6, t_2), (x_7, t_1), (x_7, t_2), (x_8, t_1), (x_8, t_2)\})$$

A domain size of $d = 5$ leads to the most probable model. Within each model, one can use LVE (or any other algorithm of one's liking) again to answer queries.

With semantics in place for models with unknown universes, the last step on our mission of exploring unknown universes is QA.

13.5 Seeking Answers in Unknown Universes

The semantics of a model with an unknown universe yields a set of parameterised models. For each parameterised model, QA works as before, using, e.g., LVE to answer queries for marginal or conditional distributions. If each parameterised model has a probability associated, each result has a probability associated. Let us look at the running example given a query for a marginal distribution of $Sick(X)$ instantiated with x_1

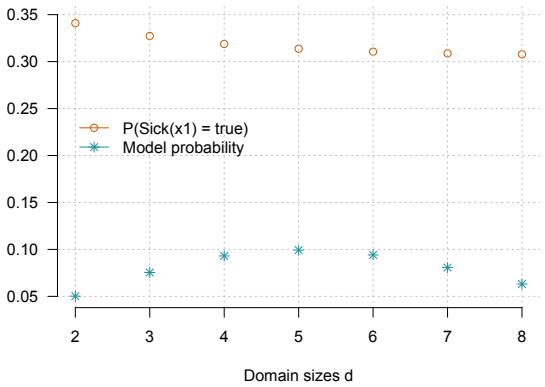
Example 13.5.1 (Query and model probabilities)**.** Considering $P(Sick(x_1))$ as a query, each of the parameterised models in $\mathbf{G}_{un}$ provides an answer, i.e., a marginal distribution for $Sick(x_1)$. Figure 13.2a shows the probabilities of $Sick(x_1) = true$ plotted for each parameterised model with rising domain sizes on the x-axis, denoted by a circle. The stars denote the probability associated with each parameterised model. As mentioned before, the model with domain size $d = 5$ is most probable and returns a probability of 0.31 for $Sick(x_1) = true$. The model probabilities decrease to the left and right of 5 while the queried probability declines the larger the domain size becomes.

Given a set of parameterised models, new query types emerge beyond the queries looked at so far, which we explore next.

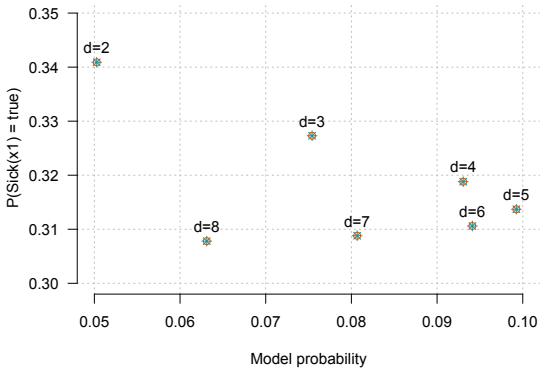
Emerging New Queries As we have a set of parameterised models and, therefore, a set of results, new forms of query processing and queries emerge. If asking for the probability of a certain event, e.g., $Sick(x_1) = true$, one might be interested in those models that provide answers with highest probability (top-k query w.r.t. query probability).

Example 13.5.2 (Top-k query)**.** A top-3 query w.r.t. query probabilities in Fig. 13.2 returns the models with domain sizes 2 to 4 (highest probabilities for $Sick(x_1) = true$).

If events such as $Sick(x_1) = true$ have been observed, guaranteed constants are available and a top-k query may support identifying most probable domain sizes for other logvars. Given the associated probabilities, one might be interested in a top-k query w.r.t.



(a) $P(Sick(x_1) = true)$ and model probability for each parameterised model in $\mathbf{G}_{un}$



(b) Model probability and $P(Sick(x_1) = true)$ plotted for a Skyline query.

Figure 13.2: Plots of query and model probabilities

model probabilities or those models that have the highest combined probabilities of event and model (skyline query w.r.t. event and model probability). Asking for distributions, the results over different models might exhibit shifts or clusters worth investigating.

Example 13.5.3 (Skyline query)**.** Figure 13.2b plots the model probabilities versus the query probabilities. The skyline here consists of the dots labeled $d = 2, d = 3, d = 4, d = 5$, which form the outskirt of the dots from the origin of the plane.

As shown, given the semantics of models with unknown universe, one can answer various queries. Handling unknown universes leads to more work as an algorithm performs QA for multiple instances, which share certain aspects. So, while this chapter focusses on semantics, we briefly consider how one would implement it.

Arriving at an Implementation As the model structure is identical for each constraint world and multiple queries probably have to be answered, an algorithm for repeated inference like LJT or FOKC would be fitting to implement the semantics. Given that the model structure is the same over different instantiations from constraint worlds, the underlying FO jtree can be reused, constraints adapted using adaptive inference, and calculations cached to a certain extend Kazemi and Poole (2016a). Given top-k queries w.r.t. query probabilities, one would aim at adapting an implementation in the vain of top-k queries on PDBs Fagin (1999) as to not evaluate more models than necessary.

13.6 Interim Conclusion: Exploring Unknown Universes

Detaching a parameterised model from a known universe allows for handling unknown universes. Using a constraint program and a domain distribution, one can generate

possible worlds to instantiate a template model, yielding a set of parameterised models. For each resulting parameterised model, distribution semantics holds again. Given the probabilistic nature of the domain distribution and possibly the constraint program, one can restrict the number of worlds to a number that is feasible to handle. As the same template model is instantiated with different constraint worlds, it is possible to perform efficient QA, reusing helper structures or previous calculations. Additionally, new and interesting forms of queries (top-k, skyline) arise that allow for further exploring a given model under changing domains.

New inference tasks arise as well, e.g., regarding automatic generation of new instances that are guaranteed to exist in open universes or regarding learning constraint rules in completely unknown universes. Detaching a model from a known universe brings us closer to understanding how transfer learning works: Transferring a model from one domain to a next opens up possibilities for assumptions changing w.r.t. indistinguishable individuals and their relations.

Chapter 14

Conclusion

Probabilistic relational models lie at the heart of many algorithms. They are the structure that stores relations and features, describing universes of individuals. Posing queries for such models is a major task and being able to efficiently answer these queries is an ongoing research quest. We summarise the contributions, which this dissertation has made towards efficient repeated inference, and provide some directions for future work.

14.1 Summary of Contributions

In this dissertation, we focus on exact repeated inference, i.e., solving multiple instances of different query answering problems. We use the lifting idea to exploit the relational part in models for compact representations and efficient calculations. We combine lifting with junction trees for efficient repeated inference. Overall, we can summarise the contributions of this thesis as follows.

Lifting the Junction Tree Algorithm We lift jtrees to compactly represent symmetric nodes, allowing QA algorithms like LVE to use repeated patterns for efficient QA within LJT. We prevent algorithm-induced groundings stemming from forced eliminations orders during message passing. On a theoretical level, we show that the completeness results of LVE also hold for LJT. We analyse the complexity of LJT, demonstrating that the connection that exists between VE and JT also exists between LVE and LJT and that lifting JT to LJT mirrors the effect of lifting VE to LVE.

Conjunctive Queries in LJT and LVE On our way to supporting more complex queries, we extend the query answering step of LJT to handle conjunctive queries, in which a single query may contain a set of query terms. We apply lifting to queries, which enables lifted QA for queries that contain a set of interchangeable query terms by extending LVE appropriately. The theoretical analysis shows that now queries influence whether an algorithm run can be lifted. Thus, completeness also depends on the class of queries.

Solving MPE and MAP queries We redefine the LVE operator suite by Taghipour *et al.* (2013c) to compute solutions to MPE problems. We arrange LVE and MPE operators to

solve not only MPE queries but also MAP queries. This work culminates in one combined algorithm for a set of queries, where each query could be a probability query as well as an assignment query, reusing computations as much as possible in an FO jtree. Whereas MPE queries correspond to empty probability queries, MAP queries show a connection to parameterised conjunctive queries. Given an FO jtree, we are able to characterise MAP queries that do not lead to larger intermediate results than MPE queries.

Algorithm Extensions *Adaptive* LJT handles incremental changes in a model or evidence, which allows LJT to fast reach the point of answering queries again. Looking at LJT as a *backbone for lifted QA*, we show that any QA algorithm can take on the role of LVE in LJT as long as the QA algorithm supports lifted evidence handling as well as the queries needed for message calculations. The query language supported by the subroutine QA algorithm determines the types of queries LJT can answer.

The last chapter takes on unknown universes. LVE and, as a consequence, LJT require known universes to be able to calculate counts during computations when performing inference. Actually, all lifted algorithms need to know the universe. However, the goal is to facilitate universes where the constants or the numbers of constants are unknown. While the chapter sketches semantics regarding how to handle unknown domain sizes, it also points directly to future work, which we consider next.

14.2 Future Work

Moving forward from this dissertation, there are a myriad of ways to continue research in exact repeated inference and beyond. We single out four topics.

Constraints LVE as defined by the operator suite is decoupled from the constraint language. The implementation realises constraints through decision trees. Looking into suitable approaches to implementing constraint handling opens up a whole new universe of open questions. The actual setting of a scenario, for which one wants to use lifted QA, influences the way constraints may be stored. But also, LVE puts requirements forward regarding the counts it needs to actually carry out computations during QA. One way to go about constraint handling may be encoding constraints as an answer set program, concentrating on required counts. A Datalog program as for unknown universes could also provide a suitable avenue. Some generic rules or programs specifying how constraints are generated is desirable if one wants to keep constraints detached from specific domain sizes to then instantiate constraints for given domains.

Unknown and Open Universes Constraints that are generated given specific domains provide the basis for handling unknown universes as discussed in Chapter 13. While

this discussion provides a first look into a possible way of handling unknown universes, questions regarding how to implement unknown universes are far from answered. In open and unknown universes, new inference tasks arise, e.g., regarding automatic generation of new instances that are guaranteed to exist in open universes or regarding learning constraint rules in completely unknown universes.

Domain Transfer The semantics of a parameterised model is given by grounding and building a full joint distribution. But, what happens to the distribution if the domain sizes change or domains are replaced with completely new constants? Changing a domain size basically means transferring a given model to a new domain, as the underlying joint distribution changes. Both completely new sets of constants as well as changing domain sizes bear the question whether the full joint has an inextricable connection to the domains it has been specified or learned for in the first place. From this observation, new research questions arise when domains change regarding, e.g., (i) how a distribution should change (ii) what behaviour is implicitly encoded and whether that behaviour is wanted, or (iii) if another behaviour is expected, e.g., if more people are able to travel in a universe, the model should reveal a higher chance of an epidemic occurring.

Model Integration Applying machine learning techniques on large data sets might lead to one large model that might not represent data that well. But learning fragments might yield a reasonable representation of certain aspects in data. Given various fragments, we may build one model by integrating the fragments, which requires aligning randvars and logvars of the fragments. One might even consider integrating FO jtrees of the fragments or, keeping the fragments, one could integrate query results.

Part IV

Appendix

Appendix A

Further Operators of MPE-LVE

The operators are based on the LVE operators as defined by Taghipour *et al.* (2013c) and Taghipour (2013). We first provide some helper functions.

A.1 Helper Functions

Definition A.1.1 (Splitting on Overlap)**.** Splitting a constraint C_1 on its $\mathbf{Y}$-overlap with C_2, denoted $C_1/_{\mathbf{Y}}C_2$, partitions C_1 into two subsets, containing all tuples for which the $\mathbf{Y}$ part occurs or does not occur, respectively, in C_2. $C_1/_{\mathbf{Y}}C_2 = \{\{t \in C_1 | \pi_{\mathbf{Y}}(t) \in \pi_{\mathbf{Y}}(C_2)\}, \{t \in C_1 | \pi_{\mathbf{Y}}(t) \notin \pi_{\mathbf{Y}}(C_2)\}\}$

Definition A.1.2 (Parfactor Partitioning)**.** Given a parfactor $g = \phi(\mathcal{A})|C$ and a partition $\mathbb{C} = \{C_i\}_{i=1}^n$ of C, $\textsc{partition}(g, \mathbb{C}) = \{\phi(\mathcal{A})^{\mathcal{B}}_{|C_i}\}_{i=1}^n$

Definition A.1.3 (Group-by)**.** Given a constraint C and a function $f : C \to R$, $\textsc{Group-By}(C, f)$ partitions C into subsets of elements that have the same result for f. Formally, $\textsc{Group-By}(C, f) = C/\sim_f$, with $x \sim_f y \Leftrightarrow f(x) = f(y)$ and $/$ denoting set quotient.

Definition A.1.4 (Joint-count)**.** Given a constraint C over variables $\mathbf{X}$, partitioned into $\{C_1, C_2\}$, and a counted logvar $X \in \mathbf{X}$; then for any $t \in C$, with $L = \mathbf{X} \backslash \{X\}$ and $l = \pi_L(t)$, $\textsc{joint-count}_{X,\{C_1,C_2\}}(t) = (|\pi_X(\sigma_{L=l}(C_1))|, |\pi_X(\sigma_{L=l}(C_2))|)$.

A.2 Transforming Operators

Operator 4 Splitting

Operator SPLIT
Inputs:
(1) $g = \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$
(2) $A = P(\mathbf{Y}) \in \mathcal{A}$, to be split
(3) $A' = P(\mathbf{Y})_{|C'}$ or $\#_Y[P(\mathbf{Y})]_{|C'}$, to split on
Output: $\textsc{partition}(g, \mathbb{C})$, with $\mathbb{C} = C/_{\mathbf{Y}}C' \setminus \emptyset$
Postcondition: $G \equiv G \setminus \{g\} \cup \textsc{split}(g, A, A')$

Operator 5 Expansion

Operator EXPAND
Inputs:
(1) $g = \forall \mathbf{L} : \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$
(2) $A = \#_X[R(\mathbf{X})] \in \mathcal{A}$, to be expanded
(3) $A' = R(\mathbf{X})|C'$ or $\#_Y[R(\mathbf{X})]|C'$, to expand on
Preconditions:
(1) $\forall B \in \mathcal{B} : X \notin B$
Output: $\{g_i = \phi'_i(\mathcal{A}'_i)^{\mathcal{B}}_{|C'_i}\}_{i=1}^{n}$ where
(1) $C/_{\mathbf{X}}C' = \{C^{com}, C^{excl}\}$
(2) $\{C_1, \ldots, C_n\} = \text{GROUP-BY}(C, \text{JOINT-COUNT}_{X, C/_{\mathbf{X}}C'})$
(3) for all i where $C_i \bowtie C^{com} = \emptyset$ or $C_i \bowtie C^{excl} = \emptyset$: $\phi'_i = \phi, \mathcal{A}'_i = \mathcal{A}, C'_i = C_i$, and
(4) for all other i:
$C'_i = \pi_{lv(\mathcal{A})}(C_i) \bowtie (\rho_{X \to X_{com}}(C^{com}) \bowtie \rho_{X \to X_{excl}}(C^{excl}))$,
$\mathcal{A}'_i = \mathcal{A} \setminus \{A\} \cup \{A\theta_{com}, A\theta_{excl}\}$ with $\theta_{com} = \{X \to X_{com}\}$, $\theta_{excl} = \{X \to X_{excl}\}$,
and for each valuation $(\mathbf{l}, h_{com}, h_{excl})$ of $\mathcal{A}'_i$,

$$\phi'_i(\mathbf{l}, h_{com}, h_{excl}) = \big(\phi^P(\mathbf{l}, h_{com} \oplus h_{excl}), \phi^A(\mathbf{l}, h_{com} \oplus h_{excl})\big)$$

Postcondition: $G \equiv G \setminus \{g\} \cup \text{EXPAND}(g, A, A')$

Operator 6 Counting Normalisation

Operator COUNT-NORMALISE
Inputs:
(1) $g = \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$
(2) $\mathbf{Y}|\mathbf{Z}$, sets of logvars indicating the desired normalisation property in C
Preconditions:
(1) $\mathbf{Y} \subset lv(\mathcal{A})$ and $\mathbf{Z} \subseteq lv(\mathcal{A}) \setminus \mathbf{Y}$
Output: $\text{PARTITION}(g, \text{GROUP-BY}(C, \text{COUNT}_{\mathbf{Y}|\mathbf{Z}}))$
Postcondition: $G \equiv G \setminus \{g\} \cup \text{COUNT-NORMALISE}(g, \mathbf{Y}|\mathbf{Z})$

Operator 7 Grounding

Operator GROUND-LOGVAR
Inputs:
(1) $g = \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$
(2) $X \in lv(\mathcal{A})$, to be grounded in g
Output: $\text{PARTITION}(g, \text{GROUP-BY}(C, \pi_X))$
Postcondition: $G \equiv G \setminus \{g\} \cup \text{GROUND-LOGVAR}(g, X)$

The operators shown here, *split*, *expand*, *count-normalise*, and *ground-logvar*, are identical to their LVE counterparts, copying along assignments.

A.3 Evidence Operator

Operator 8 Lifted Absorption

Operator ABSORB
Inputs:
(1) $g = \phi(\mathcal{A})^{\mathcal{B}=()}_{|C}, g \in G$
(2) $A_i \in \mathcal{A}$ with $A_i = R(\mathbf{X})$ or $A_i = \#_{X_i}[R(\mathbf{X})]$
(3) $g_E = \phi_E(R(\mathbf{X}))_{|C_E}$, an evidence parfactor
Let $\mathbf{X}^{excl} = \mathbf{X} \setminus lv(\mathcal{A} \setminus A_i)$;
$\mathbf{X}^{nce} = \mathbf{X}^{excl} \setminus \{X_i\}$ if $A_i = \#_{X_i}[R(\mathbf{X})]$, $\mathbf{X}^{excl}$ otherwise;
$o =$ the observed value for $R(\mathbf{X})$ in g_E
Preconditions:
(1) $rv(A_{i|C_i}) \subseteq rv(A_{i|C_E})$
(2) $\mathbf{X}^{nce}$ is count-normalised w.r.t. $lv(\mathcal{A}) \setminus \mathbf{X}^{excl}$ in C.
Output: $\phi'(\mathcal{A}')_{|C'}$ such that
(1) $\mathcal{A}' = (A_1, \ldots, A_{i-1}) \circ (A_{i+1}, \ldots, A_n)$,
(2) $C' = \pi_{logvar(C) \setminus \mathbf{X}^{excl}}(C)$, and
(3) for each valuation $\mathbf{a}' = (\ldots, a_{i-1}, a_{i+1}, \ldots)$ of $\mathcal{A}'$,

$$\phi'(\mathbf{a}') = \left(\phi^P(\ldots, a_{i-1}, e, a_{i+1}, \ldots)^r, \phi^A(\ldots, a_{i-1}, e, a_{i+1}, \ldots)\right),$$

- $r = \text{COUNT}_{\mathbf{X}^{nce}|\mathbf{L}'}(C)$
- $e = \begin{cases} o & \text{if } A_i = R(\mathbf{X}), \\ h_e & \text{otherwise (namely if } A_i = \#_{X_i}[R(\mathbf{X})]). \end{cases}$

  where $h_e(o) = \text{COUNT}_{X_i|lv(\mathcal{A})}(C)$ and $h_e(.) = 0$ elsewhere.

Postcondition: $G \cup \{g_E\} \equiv G \setminus \{g\} \cup \{g_E, \text{ABSORB}(g, A_i, g_E)\}$

The operator *absorb* is usually applied when the assignments are still empty, thus, absorbing evidence changes an input parfactor only w.r.t. arguments, potentials, and possibly constraints.

A.4 Generalised Counting Operators

As already mentioned in Sections 5.2 and 9.3, the generalisations allow for

- counting logvars that appear in more than one PRV,
- merging CRVs with counted logvars of the same domain into one CRV, and
- merge-counting a PRV and a CRV with an inequality constraint into one CRV.

Operator 9 Generalised Count Conversion

Operator COUNT-CONVERT

Inputs:

(1) $g = \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$

(2) $X \in lv(\mathcal{A})$, to count in g

Preconditions:

(1) There is no (P)CRV in the set $\mathcal{A}_X = \{A \in \mathcal{A} | X \in lv(A)\}$.

(2) There is no (P)CRV $\gamma = \#_{X_i}[\dots] \in \mathcal{A}$ such that $X_i \neq X$ in C

(3) For all $g' \in G, g' \neq g : rv(A_{i|C} \cap rv(g) = \emptyset$.

Output: $\phi'(\mathcal{A}')^{\mathcal{B}}_{|C}$ such that

(1) $\mathcal{A}' = \sigma_{rv(\mathcal{A}) \setminus rv(\mathcal{A}_X)}(\mathcal{A}) \circ (\#_X[\mathcal{A}_X])$, and

(2) for each valuation $\mathbf{a}' = (\mathbf{a}, h(.))$,

$$\phi'(\mathbf{a}') = \left(\prod_{\mathbf{a}_i \in \mathcal{R}(\mathcal{A}_X)} \phi^P(\mathbf{a}, \mathbf{a}_i)^{h(\mathbf{a}_i)}, \bigcirc_{B \in \mathcal{B}} \frac{1}{r_b} \cdot \sum_{\mathbf{a}_i \in \mathcal{R}(\mathcal{A}_X)} h(\mathbf{a}_i) \cdot \phi^A(\mathbf{a}, \mathbf{a}_i) \right)$$

- $h = \{(\mathbf{a}_i, n_i)\}_{i=1}^m$, $m = |\mathcal{R}(\mathcal{A}_X)|$, $\mathbf{a}_i \in \mathcal{R}(\mathcal{A}_X)$, $n_i \in \mathbb{N}$, $\sum_{\mathbf{a}_i \in \mathcal{R}(\mathcal{A}_X)} h(\mathbf{a}_i) = r$,
- $r = \text{COUNT}_{X|\mathbf{X}'}(\pi_{\mathbf{X}}(C))$, $\mathbf{X}' = \mathbf{X} \setminus \{X\}$,
- $r_B = \begin{cases} 1 & \text{if } X \in lv(B) \\ r & \text{otherwise} \end{cases}$

Postcondition: $G \equiv G \setminus \{g\} \cup \{\text{COUNT-CONVERT}(g, X)\}$

Operator 10 Merging

Operator MERGE

Inputs:

(1) $g = \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$

(2) $(\gamma_1, \gamma_2) = (\#_{X_1}[\mathcal{A}_1], \#_{X_2}[\mathcal{A}_2])$, $\gamma_1, \gamma_2 \in \mathcal{A}$, a pair of (P)CRVs to merge

Preconditions:

(1) $gr(X_{1|C}) = gr(X_{2|C})$

Output: $\phi'(\mathcal{A}')^{\mathcal{B}}_{|C}$ such that

(1) $\mathcal{A}' = \sigma_{rv(\mathcal{A}) \setminus rv((\gamma_1, \gamma_2))} \circ (\#_X[\mathcal{A}_{12}])$, $\mathcal{A}_{12} = \sigma_{rv(\mathcal{A}_1) \setminus rv(\mathcal{A}_2)}(\mathcal{A}_1) \circ \mathcal{A}_2\theta$, $\theta = \{X_2 \to X_1\}$

(2) for each valuation $\mathbf{a}' = (\mathbf{a}, h(.))$,

$$\phi'(\mathbf{a}') = \left(\phi^P(\mathbf{a}, h_{[\mathcal{A}_1]}, h_{[\mathcal{A}_2\theta]}), \phi^A(\mathbf{a}, h_{[\mathcal{A}_1]}, h_{[\mathcal{A}_2\theta]})\right)$$

- $h = \{(\mathbf{a}_i, n_i)\}_{i=1}^m$, $m = |\mathcal{R}(\mathcal{A}_{12})|$, $\mathbf{a}_i \in \mathcal{R}(\mathcal{A}_{12})$, $n_i \in \mathbb{N}$, $\sum_{\mathbf{a}_i \in \mathcal{R}(\mathcal{A}_{12})} h(\mathbf{a}_i) = r$,
- $r = \text{COUNT}_{X_1|\mathbf{X}}(\pi_{\mathbf{X}}(C))$, and
- $h_{[\mathcal{A}_i]}$ denotes the projection of h on $\mathcal{A}_i$.

Postcondition: $G \equiv G \setminus \{g\} \cup \{\text{MERGE}(g, \gamma_1, \gamma_2)\}$

Both operators are combined into one operator to merge and count a PRV into a CRV.

Operator 11 Merge-counting

Operator MERGE-COUNT

Inputs:

(1) $g = \phi(\mathcal{A})^{\mathcal{B}}_{|C}, g \in G$

(2) $\gamma = \#_{X_1}[\mathcal{A}_1] \in \mathcal{A}$

(3) $X_2 \in lv(\mathcal{A})$, to merge-count into γ

Preconditions:

(1) There is no (P)CRV in the set $\mathcal{A}_2 = \{A \in \mathcal{A} | X_2 \in lv(A)\}$.

(2) γ is the only (P)CRV for whose counted logvar X_1 holds $X_1 \neq X_2$.

Output: $\phi'(\mathcal{A}')^{\mathcal{B}}_{|C}$ such that

(1) $\mathcal{A}' = \sigma_{rv(\mathcal{A})\setminus rv(\mathcal{A}_2)\setminus\{\gamma\}}(\mathcal{A}) \circ (\#_{X_1}[\mathcal{A}_{12}])$, $\mathcal{A}_{12} = \sigma_{rv(\mathcal{A}_1)\setminus rv(\mathcal{A}_2)}(\mathcal{A}_1) \circ \mathcal{A}_2\theta$, $\theta = \{X_2 \to X_1\}$,

(2) for each valuation $\mathbf{a}' = (\mathbf{a}, h(.))$,

$$\phi'(\mathbf{a}') = \left(\prod_{\mathbf{a}_{12} \in \mathcal{R}(\mathcal{A}_{12})} \phi^P(\mathbf{a}, h^{-\mathbf{a}_1}_{[\mathcal{A}_1]}, \mathbf{a}_2)^{h(\mathbf{a}_{12})}, \bigcirc_{B \in \mathcal{B}} \frac{1}{r_b} \cdot \sum_{\mathbf{a}_{12} \in \mathcal{R}(\mathcal{A}_{12})} h(\mathbf{a}_{12}) \cdot \phi^P(\mathbf{a}, h^{-\mathbf{a}_1}_{[\mathcal{A}_1]}, \mathbf{a}_2) \right)$$

- $\mathbf{a}_i$ denotes the projection of the valuation of $\mathbf{a}_{12}$ on $\mathcal{A}_i\{X_2 \to X_1\}$,
- $h^{-\mathbf{r}}$ is such that $h^{-\mathbf{r}}(\mathbf{r}) = h(\mathbf{r}) - 1$, and $h^{-\mathbf{r}}(\mathbf{r}') = h(\mathbf{r}')$ for $\mathbf{r} \neq \mathbf{r}'$,
- $h = \{(\mathbf{a}_i, n_i)\}_{i=1}^m$, $m = |\mathcal{R}(\mathcal{A}_{12})|$, $\mathbf{a}_i \in \mathcal{R}(\mathcal{A}_{12})$, $n_i \in \mathbb{N}$, $\sum_{\mathbf{a}_i \in \mathcal{R}(\mathcal{A}_{12})} h(\mathbf{a}_i) = r$,
- $r = \text{COUNT}_{X_1|\mathbf{X}'}(\pi_{\mathbf{X}}(C))$,
- $r_B = \begin{cases} 1 & \text{if } X \in lv(B) \\ r & \text{otherwise} \end{cases}$

Postcondition: $G \equiv G \setminus \{g\} \cup \{\text{MERGE-COUNT}(g, \gamma, X_2)\}$

Appendix B

Inputs of the Empirical Evaluation

During the empirical evaluation, we vary (i) the largest domain size n, (ii) the number of parclusters n_J, and (iii) the lifted width $(w_g, w_\#)$ in which w_g is the ground width and $w_\#$ the counting width. In the following, we provide a description of the basic model as well as the variations for n, n_J, w_g, and $w_\#$, the evidence setup, and the fusion model.

B.1 Basic Setting

The basic input model is G_{ex} with boolean ranges, $n = 1000$, which we use for each logvar, $n_J = 3$, $w_g = 3$, and $w_\# = 1$. The domain sizes are $n = |\mathcal{D}(D)| = |\mathcal{D}(W)| = |\mathcal{D}(M)| = |\mathcal{D}(X)| = 1000$. The parfactors include random potentials between 0 and 1.

B.2 Varying the Domain Size

G_{ex} remains the same, while $n = |\mathcal{D}(D)| = |\mathcal{D}(W)| = |\mathcal{D}(M)| = |\mathcal{D}(X)|$ varies. The last line holds the basic setting. $\max|\mathcal{A}|$ refers to the largest number of arguments.

Table B.1: Key numbers about the input models used when varying n

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max\lvert\mathcal{A}\rvert$
2	3	3	1	0	4	11	$\{1,2\}$	3
4	3	3	1	0	4	37	$\{1,2\}$	3
6	3	3	1	0	4	79	$\{1,2\}$	3
8	3	3	1	0	4	137	$\{1,2\}$	3
10	3	3	1	0	4	211	$\{1,2\}$	3
12	3	3	1	0	4	301	$\{1,2\}$	3
14	3	3	1	0	4	407	$\{1,2\}$	3
16	3	3	1	0	4	529	$\{1,2\}$	3
18	3	3	1	0	4	667	$\{1,2\}$	3
20	3	3	1	0	4	821	$\{1,2\}$	3
50	3	3	1	0	4	5051	$\{1,2\}$	3

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max\lvert\mathcal{A}\rvert$
100	3	3	1	0	4	20101	$\{1,2\}$	3
150	3	3	1	0	4	45151	$\{1,2\}$	3
200	3	3	1	0	4	80201	$\{1,2\}$	3
250	3	3	1	0	4	125251	$\{1,2\}$	3
300	3	3	1	0	4	180301	$\{1,2\}$	3
350	3	3	1	0	4	245351	$\{1,2\}$	3
400	3	3	1	0	4	320401	$\{1,2\}$	3
450	3	3	1	0	4	405451	$\{1,2\}$	3
500	3	3	1	0	4	500501	$\{1,2\}$	3
550	3	3	1	0	4	605551	$\{1,2\}$	3
600	3	3	1	0	4	720601	$\{1,2\}$	3
650	3	3	1	0	4	845651	$\{1,2\}$	3
700	3	3	1	0	4	980701	$\{1,2\}$	3
750	3	3	1	0	4	1125751	$\{1,2\}$	3
800	3	3	1	0	4	1280801	$\{1,2\}$	3
850	3	3	1	0	4	1445851	$\{1,2\}$	3
900	3	3	1	0	4	1620901	$\{1,2\}$	3
950	3	3	1	0	4	1805951	$\{1,2\}$	3
1000	3	3	1	0	4	2001001	$\{1,2\}$	3

B.3 Varying the Number of Parclusters

When varying n_J from 2 to 11, w_g and $w_\#$ remain fixed. Each new parcluster has a lifted width of $(3,0)$, leaving the overall lifted width at $(3,1)$. G_{ex} appears where $n_J = 3$ and $n = 1000$. A small jump in $\lvert gr(G)\rvert$ occurs when additional PRVs trigger a new parcluster. A larger jump occurs if an additional logvar triggers a new parcluster.

Table B.3: Key numbers about the input models used when varying n_J

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max\lvert\mathcal{A}\rvert$
10	2	3	1	0	3	111	$\{1,2\}$	3
10	3	3	1	0	4	211	$\{1,2\}$	3
10	4	3	1	0	5	221	$\{1,2\}$	3
10	5	3	1	0	6	231	$\{1,2\}$	3
10	6	3	1	0	7	331	$\{1,2\}$	3
10	7	3	1	0	8	341	$\{1,2\}$	3
10	8	3	1	0	9	441	$\{1,2\}$	3
10	9	3	1	0	10	451	$\{1,2\}$	3

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max\lvert\mathcal{A}\rvert$
10	10	3	1	0	11	461	$\{1,2\}$	3
10	11	3	1	0	12	561	$\{1,2\}$	3
100	2	3	1	0	3	10101	$\{1,2\}$	3
100	3	3	1	0	4	20101	$\{1,2\}$	3
100	4	3	1	0	5	20201	$\{1,2\}$	3
100	5	3	1	0	6	20301	$\{1,2\}$	3
100	6	3	1	0	7	30301	$\{1,2\}$	3
100	7	3	1	0	8	30401	$\{1,2\}$	3
100	8	3	1	0	9	40401	$\{1,2\}$	3
100	9	3	1	0	10	40501	$\{1,2\}$	3
100	10	3	1	0	11	40601	$\{1,2\}$	3
100	11	3	1	0	12	50601	$\{1,2\}$	3
1000	2	3	1	0	3	1001001	$\{1,2\}$	3
1000	3	3	1	0	4	2001001	$\{1,2\}$	3
1000	4	3	1	0	5	2002001	$\{1,2\}$	3
1000	5	3	1	0	6	2003001	$\{1,2\}$	3
1000	6	3	1	0	7	3003001	$\{1,2\}$	3
1000	7	3	1	0	8	3004001	$\{1,2\}$	3
1000	8	3	1	0	9	4004001	$\{1,2\}$	3
1000	9	3	1	0	10	4005001	$\{1,2\}$	3
1000	10	3	1	0	11	4006001	$\{1,2\}$	3
1000	11	3	1	0	12	5006001	$\{1,2\}$	3

B.4 Varying the Ground Width

Varying w_g from 2 to 11 means that n_J and $w_\#$ appear fixed and each parcluster has a ground width of w_g. G_{ex} appears where $w_g = 3$ and $n = 1000$.

Table B.5: Key numbers about the input models used when varying w_g

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max\lvert\mathcal{A}\rvert$
10	3	2	1	0	4	31	$\{2\}$	2
10	3	3	1	0	4	121	$\{1,2\}$	3
10	3	4	1	0	10	631	$\{3,4\}$	3
10	3	5	1	0	16	1231	$\{5,7\}$	3
10	3	6	1	0	25	2131	$\{8,9\}$	3
10	3	7	1	0	34	2761	$\{11,12\}$	3
10	3	8	1	0	43	3661	$\{14,15\}$	3

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max \lvert\mathcal{A}\rvert$
10	3	9	1	0	52	4561	$\{17, 18\}$	3
10	3	10	1	0	61	5191	$\{20, 21\}$	3
10	3	11	1	0	70	5821	$\{23, 24\}$	3
100	3	2	1	0	4	301	$\{2\}$	2
100	3	3	1	0	4	10201	$\{1, 2\}$	3
100	3	4	1	0	10	60301	$\{3, 4\}$	3
100	3	5	1	0	16	120301	$\{5, 7\}$	3
100	3	6	1	0	25	210301	$\{8, 9\}$	3
100	3	7	1	0	34	270601	$\{11, 12\}$	3
100	3	8	1	0	43	360601	$\{14, 15\}$	3
100	3	9	1	0	52	450601	$\{17, 18\}$	3
100	3	10	1	0	61	510901	$\{20, 21\}$	3
100	3	11	1	0	70	571201	$\{23, 24\}$	3
1000	3	2	1	0	4	3001	$\{2\}$	2
1000	3	3	1	0	4	1002001	$\{1, 2\}$	3
1000	3	4	1	0	10	6003001	$\{3, 4\}$	3
1000	3	5	1	0	16	12003001	$\{5, 7\}$	3
1000	3	6	1	0	25	21003001	$\{8, 9\}$	3
1000	3	7	1	0	34	27006001	$\{11, 12\}$	3
1000	3	8	1	0	43	36006001	$\{14, 15\}$	3
1000	3	9	1	0	52	45006001	$\{17, 18\}$	3
1000	3	10	1	0	61	51009001	$\{20, 21\}$	3
1000	3	11	1	0	70	57012001	$\{23, 24\}$	3

B.5 Varying the Counting Width

When varying $w_\#$ from 0 to 9, n_J and w_g are fixed and each of the three parclusters has a counting width of $w_\#$. G_{ex} appears where $w_\# = 1$ and $n = 1000$.

Table B.7: Key numbers about the input models used when varying $w_\#$

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max \lvert\mathcal{A}\rvert$
10	3	3	0	0	4	31	$\{2\}$	2
10	3	3	1	0	4	301	$\{1, 2\}$	3
10	3	3	2	0	4	3001	$\{1, 2\}$	4
10	3	3	3	0	7	6001	$\{2, 3\}$	4
10	3	3	4	0	10	9001	$\{3, 4\}$	4
10	3	3	5	0	13	12001	$\{4, 5\}$	4

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max\lvert\mathcal{A}\rvert$
10	3	3	6	0	16	15001	$\{5,6\}$	4
10	3	3	7	0	19	18001	$\{6,7\}$	4
10	3	3	8	0	22	21001	$\{7,8\}$	4
10	3	3	9	0	25	24001	$\{8,9\}$	4
100	3	3	0	0	4	301	$\{2\}$	2
100	3	3	1	0	4	30001	$\{1,2\}$	3
100	3	3	2	0	4	3000001	$\{1,2\}$	4
100	3	3	3	0	7	6000001	$\{2,3\}$	4
100	3	3	4	0	10	9000001	$\{3,4\}$	4
100	3	3	5	0	13	12000001	$\{4,5\}$	4
100	3	3	6	0	16	15000001	$\{5,6\}$	4
100	3	3	7	0	19	18000001	$\{6,7\}$	4
100	3	3	8	0	22	21000001	$\{7,8\}$	4
100	3	3	9	0	25	24000001	$\{8,9\}$	4
1000	3	3	0	0	4	3001	$\{2\}$	2
1000	3	3	1	0	4	$3,000,001$	$\{1,2\}$	3
1000	3	3	2	0	10	$3\cdot 1000^3+1$	$\{1,2\}$	4
1000	3	3	3	0	16	$6\cdot 1000^3+1$	$\{2,3\}$	4
1000	3	3	4	0	25	$9\cdot 1000^3+1$	$\{3,4\}$	4
1000	3	3	5	0	34	$12\cdot 1000^3+1$	$\{4,5\}$	4
1000	3	3	6	0	43	$15\cdot 1000^3+1$	$\{5,6\}$	4
1000	3	3	7	0	52	$18\cdot 1000^3+1$	$\{6,7\}$	4
1000	3	3	8	0	61	$21\cdot 1000^3+1$	$\{7,8\}$	4
1000	3	3	9	0	70	$24\cdot 1000^3+1$	$\{8,9\}$	4

B.6 Varying the Evidence Coverage

For evidence testing, we add evidence for the 1-logvar PRVs in G_{ex}, i.e., four evidence parfactors, in 10% steps. The line with $n = 1000$ and $|\mathbf{E}| = 0$ is G_{ex}.

Table B.9: Key numbers about the input models used when varying evidence coverage

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max\lvert\mathcal{A}\rvert$
10	3	3	1	0	4	211	$\{1,2\}$	3
10	3	3	1	4	4	211	$\{1,2\}$	3
10	3	3	1	8	4	211	$\{1,2\}$	3
10	3	3	1	12	4	211	$\{1,2\}$	3
10	3	3	1	16	4	211	$\{1,2\}$	3

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$ in	$\max\lvert\mathcal{A}\rvert$
10	3	3	1	20	4	211	$\{1,2\}$	3
10	3	3	1	24	4	211	$\{1,2\}$	3
10	3	3	1	28	4	211	$\{1,2\}$	3
10	3	3	1	32	4	211	$\{1,2\}$	3
10	3	3	1	36	4	211	$\{1,2\}$	3
10	3	3	1	36	4	211	$\{1,2\}$	3
100	3	3	1	0	4	20101	$\{1,2\}$	3
100	3	3	1	40	4	20101	$\{1,2\}$	3
100	3	3	1	80	4	20101	$\{1,2\}$	3
100	3	3	1	120	4	20101	$\{1,2\}$	3
100	3	3	1	160	4	20101	$\{1,2\}$	3
100	3	3	1	200	4	20101	$\{1,2\}$	3
100	3	3	1	240	4	20101	$\{1,2\}$	3
100	3	3	1	280	4	20101	$\{1,2\}$	3
100	3	3	1	320	4	20101	$\{1,2\}$	3
100	3	3	1	360	4	20101	$\{1,2\}$	3
100	3	3	1	396	4	20101	$\{1,2\}$	3
1000	3	3	1	0	4	2001001	$\{1,2\}$	3
1000	3	3	1	400	4	2001001	$\{1,2\}$	3
1000	3	3	1	800	4	2001001	$\{1,2\}$	3
1000	3	3	1	1200	4	2001001	$\{1,2\}$	3
1000	3	3	1	1600	4	2001001	$\{1,2\}$	3
1000	3	3	1	2000	4	2001001	$\{1,2\}$	3
1000	3	3	1	2400	4	2001001	$\{1,2\}$	3
1000	3	3	1	2800	4	2001001	$\{1,2\}$	3
1000	3	3	1	3200	4	2001001	$\{1,2\}$	3
1000	3	3	1	3600	4	2001001	$\{1,2\}$	3
1000	3	3	1	3996	4	2001001	$\{1,2\}$	3

B.7 Unnecessary Groundings and Fusion

For fusion, we use a slight variation of G_{ex} with $n_J = 4$, which has unnecessary groundings with LJT. After fusion, $n_J = 3$ and $w_g = 5$. We vary the domain size n again.

Table B.11: Key numbers about the input models used for evaluating fusion

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$	$\max\lvert\mathcal{A}\rvert$
2	$4 \to 3$	$3 \to 5$	1	0	5	15	$\{1,2\} \to \{2,3\}$	3

n	n_J	w_g	$w_\#$	$\lvert\mathbf{E}\rvert$	$\lvert G\rvert$	$\lvert gr(G)\rvert$	$\lvert G_i\rvert$	$\max\lvert\mathcal{A}\rvert$
4	$4 \to 3$	$3 \to 5$	1	0	5	53	$\{1,2\} \to \{2,3\}$	3
6	$4 \to 3$	$3 \to 5$	1	0	5	115	$\{1,2\} \to \{2,3\}$	3
8	$4 \to 3$	$3 \to 5$	1	0	5	201	$\{1,2\} \to \{2,3\}$	3
10	$4 \to 3$	$3 \to 5$	1	0	5	311	$\{1,2\} \to \{2,3\}$	3
12	$4 \to 3$	$3 \to 5$	1	0	5	445	$\{1,2\} \to \{2,3\}$	3
14	$4 \to 3$	$3 \to 5$	1	0	5	603	$\{1,2\} \to \{2,3\}$	3
16	$4 \to 3$	$3 \to 5$	1	0	5	785	$\{1,2\} \to \{2,3\}$	3
18	$4 \to 3$	$3 \to 5$	1	0	5	991	$\{1,2\} \to \{2,3\}$	3
20	$4 \to 3$	$3 \to 5$	1	0	5	1221	$\{1,2\} \to \{2,3\}$	3
50	$4 \to 3$	$3 \to 5$	1	0	5	7551	$\{1,2\} \to \{2,3\}$	3
100	$4 \to 3$	$3 \to 5$	1	0	5	30101	$\{1,2\} \to \{2,3\}$	3
150	$4 \to 3$	$3 \to 5$	1	0	5	67651	$\{1,2\} \to \{2,3\}$	3
200	$4 \to 3$	$3 \to 5$	1	0	5	120201	$\{1,2\} \to \{2,3\}$	3
250	$4 \to 3$	$3 \to 5$	1	0	5	187751	$\{1,2\} \to \{2,3\}$	3
300	$4 \to 3$	$3 \to 5$	1	0	5	270301	$\{1,2\} \to \{2,3\}$	3
350	$4 \to 3$	$3 \to 5$	1	0	5	367851	$\{1,2\} \to \{2,3\}$	3
400	$4 \to 3$	$3 \to 5$	1	0	5	480401	$\{1,2\} \to \{2,3\}$	3
450	$4 \to 3$	$3 \to 5$	1	0	5	607951	$\{1,2\} \to \{2,3\}$	3
500	$4 \to 3$	$3 \to 5$	1	0	5	750501	$\{1,2\} \to \{2,3\}$	3
550	$4 \to 3$	$3 \to 5$	1	0	5	908051	$\{1,2\} \to \{2,3\}$	3
600	$4 \to 3$	$3 \to 5$	1	0	5	1080601	$\{1,2\} \to \{2,3\}$	3
650	$4 \to 3$	$3 \to 5$	1	0	5	1268151	$\{1,2\} \to \{2,3\}$	3
700	$4 \to 3$	$3 \to 5$	1	0	5	1470701	$\{1,2\} \to \{2,3\}$	3
750	$4 \to 3$	$3 \to 5$	1	0	5	1688251	$\{1,2\} \to \{2,3\}$	3
800	$4 \to 3$	$3 \to 5$	1	0	5	1920801	$\{1,2\} \to \{2,3\}$	3
850	$4 \to 3$	$3 \to 5$	1	0	5	2168351	$\{1,2\} \to \{2,3\}$	3
900	$4 \to 3$	$3 \to 5$	1	0	5	2430901	$\{1,2\} \to \{2,3\}$	3
950	$4 \to 3$	$3 \to 5$	1	0	5	2708451	$\{1,2\} \to \{2,3\}$	3
1000	$4 \to 3$	$3 \to 5$	1	0	5	3001001	$\{1,2\} \to \{2,3\}$	3

Bibliography

Umut A. Acar, Alexander T. Ihler, Ramgopal R. Mettu, and Özgür Sümer. Adaptive Bayesian Inference. In *NIPS-07 Advances in Neural Information Processing Systems 21*, pages 1441–1448. Curran Associates, Inc., 2008.

Umut A. Acar, Alexander T. Ihler, Ramgopal R. Mettu, and Özgür Sümer. Adaptive Inference on General Graphical Models. In *UAI-08 Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, pages 1–8. AUAI Press, 2008.

Babak Ahmadi, Kristian Kersting, and Scott Sanner. Multi-evidence Lifted Message Passing with Application to PageRank and the Kalman Filter. In *IJCAI-11 Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 1152–1158. IJCAI Organization, 2011.

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 *UAI-11 Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*, pages 74–83. AUAI Press, 2011.

Udi Apsel and Ronen I. Brafman. Exploiting Uniform Assignments in First-Order MPE. In *UAI-12 Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence*, pages 74–83. AUAI Press, 2012.

Udi Apsel and Ronen I. Brafman. Lifted MEU by Weighted Model Counting. In *AAAI-12 Proceedings of the 26th AAAI Conference on Artificial Intelligence*, pages 1861–1867. AAAI Press, 2012.

Udi Apsel, Kristian Kersting, and Martin Mladenov. Lifting Relational MAP-LPs Using Cluster Signatures. In *AAAI-12 Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 2403–2409. AAAI Press, 2012.

Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-Loss Markov Random Fields and Probabilistic Soft Logic. *Journal of Machine Learning Research*, 18:1–67, 2017.

Elena Bellodi, Evelina Lamma, Fabrizio Riguzzi, Vitor Santos Costa, and Riccardo Zese. Lifted Variable Elimination for Probabilistic Logic Programming. *Theory and Practice of Logic Programming*, 14(4–5):681–695, 2014.

Julian Besag. Spatial Interaction and the Statistical Analysis of Lattice Systems. *Journal of the Royal Statistical Society. Series B: Methodological*, 36(2):192–236, 1974.

Tanya Braun and Marcel Gehrke. Inference in Statistical Relational AI. In *Proceedings of the International Conference on Conceptual Structures 2019*. Springer, 2019.

Tanya Braun and Ralf Möller. Avoiding Repetition in Repeated Inference on Probabilistic Relational Models: The Lifted Junction Tree Algorithm. Submitted.

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. Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm. In *GKR 2017 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning*, 2017.

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. Adaptive Inference on Probabilistic Relational Models. In *Proceedings of AI 2018: Advances in Artificial Intelligence*, pages 487–500. Springer, 2018.

Tanya Braun and Ralf Möller. Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm - Extended. In *Postproceedings of the 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning at the 26th International Joint Conference on Artificial Intelligence*, pages 54–72. Springer, 2018.

Tanya Braun and Ralf Möller. Fusing First-order Knowledge Compilation and the Lifted Junction Tree Algorithm. In *Proceedings of KI 2018: Advances in Artificial Intelligence*, pages 24–37. Springer, 2018.

Tanya Braun and Ralf Möller. Fusing First-order Knowledge Compilation and the Lifted Junction Tree Algorithm. In *8th International Workshop on Statistical Relational AI at the 27th International Joint Conference on Artificial Intelligence*, 2018.

Tanya Braun and Ralf Möller. Lifted Most Probable Explanation. In *Proceedings of the 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. IJCAI Organization, 2018.

Tanya Braun and Ralf Möller. Exploring Unknown Universes in Probabilistic Relational Models. In *Proceedings of AI 2019: Advances in Artificial Intelligence*. Springer, 2019.

Tanya Braun, Felix Kuhr, and Ralf Möller. Unsupervised Text Annotations. In *Formal and Cognitive Reasoning - Workshop at the 40th Annual German Conference on AI (KI-2017)*, 2017.

Tanya Braun. StaRAI or StaRDB? - A Tutorial on Statistical Relational AI. In *BTW-19 Proceedings Datenbanksysteme für Business, Technologie und Web – Workshopband*, pages 263–266. Gesellschaft für Informatik, 2019.

İsmail İlkan Ceylan, Adnan Darwiche, and Guy Van den Broeck. Open-world Probabilistic Databases. In *KR-16 Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning*, pages 339–348. AAAI Press, 2016.

İsmail İlkan Ceylan, Stefan Borgwardt, and Thomas Lukasiewicz. Most Probable Explanations for Probabilistic Database Queries. In *IJCAI-17 Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 950–956. IJCAI Organization, 2017.

Mark Chavira and Adnan Darwiche. Compiling Bayesian Networks Using Variable Elimination. In *IJCAI-07 Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2443–2449. IJCAI Organization, 2007.

Jaesik Choi and Eyal Amir. Lifted Relational Variational Inference. In *UAI-12 Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence*, pages 196–206. AUAI Press, 2012.

Jaesik Choi, Eyal Amir, and David J. Hill. Lifted Inference for Relational Continuous Models. In *UAI-10 Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence*, pages 13–18. AUAI Press, 2010.

Nilesh Dalvi and Dan Suciu. The Dichotomy of Probabilistic Inference for Unions of Conjunctive Queries. *Journal of the ACM*, 59(6):1–97, 2012.

Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, 17(1):229–264, 2002.

Adnan Darwiche. Recursive Conditioning. *Artificial Intelligence*, 2(1–2):4–51, 2001.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

Mayukh Das, Yunqing Wu, Tushar Khot, Kristian Kersting, and Sriraam Natarajan. Scaling Lifted Probabilistic Inference and Learning Via Graph Databases. In *Proceedings of the SIAM International Conference on Data Mining*, pages 738–746. SIAM, 2016.

Alexander Philip Dawid. Applications of a General Propagation Algorithm for Probabilistic Expert Systems. *Statistics and Computing*, 2(1):25–36, 1992.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A Probabilistic Prolog and its Application in Link Discovery. In *IJCAI-07 Proceedings of 20th International Joint Conference on Artificial Intelligence*, pages 2062–2467. IJCAI Organization, 2007.

Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. Lifted First-order Probabilistic Inference. In *IJCAI-05 Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1319–1325. IJCAI Organization, 2005.

Rodrigo de Salvo Braz, Eyal Amir, and Dan Roth. MPE and Partial Inversion in Lifted Probabilistic Variable Elimination. In *AAAI-06 Proceedings of the 21st AAAI Conference on Artificial Intelligence*, pages 1123–1130. AAAI Press, 2006.

Rina Dechter. Bucket Elimination: A Unifying Framework for Probabilistic Inference. In *Learning and Inference in Graphical Models.*, pages 75–104. MIT Press, 1999.

Arthur L. Delcher, Adam J. Grove, Simon Kasif, and Judea Pearl. Logarithmic-time Updates and Queries in Probabilistic Networks. In *UAI-95 Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*, pages 116–124. AUAI Press, 1995.

Ronald Fagin. Combining Fuzzy Information from Multiple Systems. *Journal of Computer and System Sciences*, 58(1):83–99, 1999.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and Learning in Probabilistic Logic Programs Using Weighted Boolean Formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

Tal Friedman and Guy Van den Broeck. Approximate Knowledge Compilation by Online Collapsed Importance Sampling. In *NIPS-18 Advances in Neural Information Processing Systems 31*, pages 8034–8044. Curran Associates, Inc., 2018.

Nir Friedman. The Bayesian Structural EM Algorithm. In *UAI-98 Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pages 129–138. AUAI Press, 1998.

Norbert Fuhr. Probabilistic Datalog - A Logic for Powerful Retrieval Methods. In *SIGIR-95 Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 282–290. ACM, 1995.

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 AI 2018: Advances in Artificial Intelligence*. Springer, 2018.

Marcel Gehrke, Tanya Braun, and Ralf Möller. Lifted Dynamic Junction Tree Algorithm. In *Proceedings of the 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 *Proceedings of AI 2018: Advances in Artificial Intelligence*. 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. 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 Steinhï¿½user. Towards Lifted Maximum Expected Utility. In *Proceedings of the First Joint Workshop on Artificial Intelligence in Health at the 27th International Joint Conference on Artificial Intelligence*, volume 2142, 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*. Springer, 2019.

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

Marcel Gehrke, Tanya Braun, and Ralf Möller. Relational Forward Backward Algorithm for Multiple Queries. In *FLAIRS-19 Proceedings of the 32nd International Florida Artificial Intelligence Research Society Conference*. 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*. Springer, 2019.

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

Vibhav Gogate and Pedro Domingos. Exploiting Logical Structure in Lifted Probabilistic Inference. In *Working Note of the Workshop on Statistical Relational Artificial Intelligence at the 24th Conference on Artificial Intelligence*, pages 19–25, 2010.

Vibhav Gogate and Pedro Domingos. Probabilistic Theorem Proving. In *UAI-11 Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*, pages 256–265. AUAI Press, 2011.

Eric Gribkoff, Guy Van den Broeck, and Dan Suciu. The Most Probable Database Problem. In *Proceedings of the 1st International Workshop on Big Uncertain Data*, pages 1–7, 2014.

John Hammersley and Peter Clifford. Markov Fields on Finite Graphs and Lattices. unpublished, 1971.

Manfred Jaeger and Guy Van den Broeck. Liftability of Probabilistic Inference: Upper and Lower Bounds. In *Proceedings of the 2nd International Workshop on Statistical Relational AI*, 2012.

Ariel Jaimovich, Ofer Meshi, and Nir Friedman. Template Based Inference in Symmetric Relational Markov Random Fields. In *UAI-07 Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence*, pages 191–199. AUAI Press, 2007.

Finn V. Jensen, Steffen L. Lauritzen, and Kristian G. Olesen. Bayesian Updating in Recursive Graphical Models by Local Computations. *Computational Statistics Quarterly*, 4:269–282, 1990.

Seyed Mehran Kazemi and David Poole. Knowledge Compilation for Lifted Probabilistic Inference: Compiling to a Low-Level Language. In *KR-16 Proceedings of the 15th International Conference on Principles of Knowledge Representation and Reasoning*, pages 561–564, 2016.

Seyed Mehran Kazemi and David Poole. Why is Compiling Lifted Inference into a Low-Level Language so Effective? In *6th International Workshop on Statistical Relational AI at the 25th International Joint Conference on Artificial Intelligence*, pages 1–7, 2016.

Seyed Mehran Kazemi, Angelika Kimmig, Guy Van den Broeck, and David Poole. Domain Recursion for Lifted Inference with Existential Quantifiers. In *7th International Workshop on Statistical Relational AI at the 33rd Conference on Uncertainty in Artificial Intelligence*, pages 1–6, 2017.

Kristian Kersting, Babak Ahmadi, and Sriraam Natarajan. Counting Belief Propagation. In *UAI-09 Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, pages 277–284. AUAI Press, 2009.

Kristian Kersting, Martin Mladenov, and Pavel Tokmakov. Relational Linear Programming. *Artificial Intelligence*, 244:188–216, 2017.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. An Algebraic Prolog for Reasoning about Possible Worlds. In *AAAI-11 Proceedings of the 25th AAAI Conference on Artificial Intelligence*, pages 209–214. AAAI Press, 2011.

Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic Model Counting. *Journal of Applied Logic*, 22(C):46–62, 2017.

Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

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*, 50:157–224, 1988.

Yann LeCun. Learning World Models: the Next Step Towards AI. Invited Talk at IJCAI-ECAI 2018, 2018. https://www.youtube.com/watch?v=U2mhZ9E8Fk8, accessed November 19, 2018.

Stefan Lüdtke, Max Schröder, Sebastian Bader, Kristian Kersting, and Thomas Kirste. Lifted Filtering via Exchangeable Decomposition. In *IJCAI-18 Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 5067–5073. IJCAI Organization, 2018.

Brian Milch, Luke S. Zettelmoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted Probabilistic Inference with Counting Formulas. In *AAAI-08 Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1062–1068. AAAI Press, 2008.

Martin Mladenov, Kristian Kersting, and Amir Globerson. Efficient Lifting of MAP LP Relaxations Using k-Locality. In *AISTATS-14 Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, pages 623–632. PMLR, 2014.

Luis Muñoz-González, Daniele Sgandurra, Martín Barrère, and Emil C. Lupu. Exact Inference Techniques for the Analysis of Bayesian Attack Graphs. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–14, 2017.

Kevin Patrick Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, University of California, Berkeley, 2002.

Aniruddh Nath and Pedro Domingos. Efficient Belief Propagation for Utility Maximisation and Repeated Inference. In *AAAI-10 Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 1187–1192. AAAI Press, 2010.

Aniruddh Nath and Pedro Domingos. Efficient Lifting for Online Probabilistic Inference. In *AAAI-10 Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pages 64–69. AAAI Press, 2010.

Mathias Niepert and Guy Van den Broeck. Tractability through Exchangeability: A New Perspective on Efficient Probabilistic Inference. In *AAAI-14 Proceedings of the 28th AAAI Conference on Artificial Intelligence*, pages 2467–2475. AAAI Press, 2014.

Mathias Niepert. Markov Chains on Orbits of Permutation Groups. In *UAI-12 Proceedings of the 28th Conference on Uncertainty in Artificial Intelligence*, pages 624–633. AUAI Press, 2012.

Mathias Niepert. Symmetry-aware Marginal Density Estimation. In *AAAI-13 Proceedings of the 27th AAAI Conference on Artificial Intelligence*, pages 725–731. AAAI Press, 2013.

David A. Papa and Igor L. Markov. Hypergraph Partitioning and Clustering. In *Handbook of Approximation Algorithms and Metaheuristics*. Chapman and Hall/CRC, 2007.

James D. Park. Using Weighted MAX-SAT Engines to Solve MPE. In *Proceedings of the 18th National Conference on Artificial intelligence*, pages 682–687. AAAI Press, 2002.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

David Poole. First-order Probabilistic Inference. In *IJCAI-03 Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 985–991. IJCAI Organization, 2003.

Somdeb Sarkhel, Deepak Venugopal, Parag Singla, and Vibhav Gogate. Lifted MAP Inference for Markov Logic Networks. In *AISTATS-14 Proceedings of the 17th International Conference on Artificial Intelligence and Statistics*, pages 859–867. AAAI Press, 2014.

Taisuke Sato. A Statistical Learning Method for Logic Programs with Distribution Semantics. In *Proceedings of the 12th International Conference on Logic Programming*, pages 715–729. MIT Press, 1995.

Glenn R. Shafer and Prakash P. Shenoy. Probability Propagation. *Annals of Mathematics and Artificial Intelligence*, 2(1):327–351, 1990.

Vishal Sharma, Noman Ahmed Sheikh, Happy Mittal, Vibhav Gogate, and Parag Singla. Lifted Marginal MAP Inference. In *UAI-18 Proceedings of the 34th Conference on Uncertainty in Artificial Intelligence*, pages 917–926. AUAI Press, 2018.

Prakash P. Shenoy and Glenn R. Shafer. Axioms for Probability and Belief-Function Propagation. *Uncertainty in Artificial Intelligence 4*, 9:169–198, 1990.

Dimitar Shterionov, Joris Renkens, Jonas Vlasselaer, Angelika Kimmig, Wannes Meert, and Gerda Janssens. The most probable explanation for probabilistic logic programs with annotated disjunctions. In *Revised Selected Papers of the 24th International Conference on Inductive Logic Programming - Volume 9046*, pages 139–153. Springer, 2015.

Parag Singla and Pedro Domingos. Markov Logic in Infinite Domains. In *UAI-07 Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence*, pages 368–375. AUAI Press, 2007.

Parag Singla and Pedro Domingos. Lifted First-order Belief Propagation. In *AAAI-08 Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1094–1099. AAAI Press, 2008.

Siddharth Srivastava, Stuart Russell, Paul Ruan, and Xiang Cheng. First-order Open-universe POMDPs. In *UAI-14 Proceedings of the 30th Conference on Uncertainty in Artificial Intelligence*, pages 742–751. AUAI Press, 2014.

Nima Taghipour and Jesse Davis. Generalised Counting for Lifted Variable Elimination. In *Proceedings of the 2nd International Workshop on Statistical Relational AI*, pages 1–8, 2012.

Nima Taghipour, Jesse Davis, and Hendrik Blockeel. First-order Decomposition Trees. In *NIPS-13 Advances in Neural Information Processing Systems 26*, pages 1052–1060. Curran Associates, Inc., 2013.

Nima Taghipour, Jesse Davis, and Hendrik Blockeel. Generalised Counting for Lifted Variable Elimination. In *ILP-13 Proceedings of the 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 *AISTATS-13 Proceedings of the 16th International Conference on Artificial Intelligence and Statistics*, pages 572–580. AAAI Press, 2013.

Nima Taghipour. *Lifted Probabilistic Inference by Variable Elimination.* PhD thesis, KU Leuven, 2013.

Matthias Thimm, Marc Finthammer, Sebastian Loh, Gebriele Kern-Isberner, and Christoph Beierle. A System for Relational Probabilistic Reasoning on Maximum Entropy. In *FLAIRS-10 Proceedings of the 23rd International Florida Artificial Intelligence Research Society Conference*, pages 116–121. AAAI Press, 2010.

Guy Van den Broeck and Jesse Davis. Conditioning in First-Order Knowledge Compilation and Lifted Probabilistic Inference. In *AAAI-12 Proceedings of the 26th 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 *AAAI-15 Proceedings of the 29th AAAI Conference on Artificial Intelligence*, pages 3599–3605. AAAI Press, 2015.

Guy Van den Broeck, Nima Taghipour, Wannes Meert, Jesse Davis, and Luc De Raedt. Lifted Probabilistic Inference by First-order Knowledge Compilation. In *IJCAI-11 Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 2178–2185. IJCAI Organization, 2011.

Guy Van den Broeck. On the Completeness of First-order Knowledge Compilation for Lifted Probabilistic Inference. In *NIPS-11 Advances in Neural Information Processing Systems 24*, pages 1386–1394. Curran Associates, Inc., 2011.

Guy Van den Broeck. *Lifted Inference and Learning in Statistical Relational Models.* PhD thesis, KU Leuven, 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.

Nevin L. Zhang and David Poole. A Simple Approach to Bayesian Network Computations. In *Proceedings of the 10th Canadian Conference on Artificial Intelligence*, pages 171–178. Springer, 1994.

Publications

Conference Papers

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 - Extended. In *Postproceedings of the 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning at the 26th International Joint Conference on Artificial Intelligence*, pages 54–72. Springer, 2018

Tanya Braun and Ralf Möller. Lifted Most Probable Explanation. In *Proceedings of the 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. IJCAI Organization, 2018

Tanya Braun and Ralf Möller. Fusing First-order Knowledge Compilation and the Lifted Junction Tree Algorithm. In *Proceedings of KI 2018: Advances in Artificial Intelligence*, pages 24–37. Springer, 2018

Tanya Braun and Ralf Möller. Adaptive Inference on Probabilistic Relational Models. In *Proceedings of AI 2018: Advances in Artificial Intelligence*, pages 487–500. Springer, 2018

Tanya Braun and Ralf Möller. Exploring Unknown Universes in Probabilistic Relational Models. In *Proceedings of AI 2019: Advances in Artificial Intelligence*. Springer, 2019

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

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

Marcel Gehrke, Tanya Braun, and Ralf Möller. Relational Forward Backward Algorithm for Multiple Queries. In *FLAIRS-19 Proceedings of the 32nd International Florida Artificial Intelligence Research Society Conference*. AAAI Press, 2019

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

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

Journal Articles

Tanya Braun and Ralf Möller. Avoiding Repetition in Repeated Inference on Probabilistic Relational Models: The Lifted Junction Tree Algorithm. Submitted

Workshop Papers

Tanya Braun, Felix Kuhr, and Ralf Möller. Unsupervised Text Annotations. In *Formal and Cognitive Reasoning - Workshop at the 40th Annual German Conference on AI (KI-2017)*, 2017

Tanya Braun and Ralf Möller. Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm. In *GKR 2017 5th International Workshop on Graph Structures for Knowledge Representation and Reasoning*, 2017

Tanya Braun and Ralf Möller. Fusing First-order Knowledge Compilation and the Lifted 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, Ralf Möller, Alexander Waschkau, Christoph Strumann, and Jost Steinhï¿½user. Towards Lifted Maximum Expected Utility. In *Proceedings of the First Joint Workshop on Artificial Intelligence in Health at the 27th International Joint Conference on Artificial Intelligence*, volume 2142, pages 93–96. CEUR-WS.org, 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. 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

Extended Abstracts

Tanya Braun. StaRAI or StaRDB? - A Tutorial on Statistical Relational AI. In *BTW-19 Proceedings Datenbanksysteme für Business, Technologie und Web – Workshopband*, pages 263–266. Gesellschaft für Informatik, 2019

Tanya Braun and Marcel Gehrke. Inference in Statistical Relational AI. In *Proceedings of the International Conference on Conceptual Structures 2019*. Springer, 2019