Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm

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

Abstract

Standard approaches for inference in probabilistic formalisms with first-order constructs include lifted variable elimination (LVE) for single queries. To handle multiple queries efficiently, the lifted junction tree algorithm (LJT) uses a first-order cluster representation of a knowledge base and LVE in its computations. We extend LJT with a full formal specification of its algorithm steps incorporating (i) the lifting tool of counting and (ii) answering of conjunctive queries. Given multiple queries, e.g., in machine learning applications, our approach enables us to compute answers faster than LJT and existing approaches tailored for single queries.

1 Introduction

AI research and application areas such as natural language understanding and machine learning (ML) need efficient inference algorithms. Modeling realistic scenarios results in large probabilistic knowledge bases, also called models, that require reasoning about sets of individuals. Lifting uses symmetries in a model to speed up reasoning with known domain objects. We study the problem of reasoning in large models that exhibit symmetries. Our inputs are a model and queries for probabilities or probability distributions of random variables (randvars) given evidence. Inference tasks reduce to computing marginal distributions. We aim to enhance the efficiency of these computations when answering multiple queries, a common scenario in ML. We exploit that a model remains constant under multiple queries.

We [2016] have introduced a lifted junction tree algorithm (LJT) for multiple queries on models with first-order constructs. LJT is based on the junction tree algorithm [Lauritzen and Spiegelhalter, 1988] and lifted variable elimination (LVE) [Taghipour, 2013] using a first-order junction tree (FO jtree) to represent clusters of randvars in a model. This paper extends LJT and contributes the following. We give a formal specification of the LJT steps incorporating counting to lift more computations and allow a wider variety of model specifications, adapting the LVE heuristic for elimination order, and handling conjunctive queries. We adapt counting by Taghipour [2013] and extend answering queries that cover multiple clusters by Koller and Friedman [2009].

LJT imposes some static overhead for building an FO jtree and message passing. Counting allows accelerating computations during message passing and query answering. Handling conjunctive queries allows for answering more complex queries. We significantly speed up runtime compared to LVE and LJT. Overall, we handle multiple queries more efficiently than existing approaches tailored for single queries.

The remainder of this paper has the following structure: First, we look at related work on exact lifted inference and the junction tree algorithm. Then, we introduce basic notations and data structures and recap LVE and LJT. We present our extension incorporating counting and conjunctive queries, followed by a brief empirical evaluation. Last, we present a conclusion and upcoming work.

2 Related Work

In the last two decades, researchers have sped up runtimes for inference significantly. 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 dtree represents such a decomposition [Darwiche, 2001]. LVE, first introduced in [Poole and Zhang, 2003] and expanded in [de Salvo Braz, 2007], exploits symmetries at a global level. LVE saves computations by reusing intermediate results for isomorphic subproblems. Milch *et al.* [2008] introduce counting to lift certain computations where lifted summing out is not applicable. Taghipour [2013] extends the formalism to its current standard by generalizing counting. He formalizes lifting by defining lifting operators. The operators appear in internal calculations of LJT.

For multiple queries in a propositional setting, Lauritzen and Spiegelhalter [1988] introduce jtrees, a representation of clusters in a model, along with a reasoning algorithm. The algorithm distributes knowledge in a jtree with a message passing scheme, also known as probability propagation (PP), and answers queries on the smaller clusters. Well known PP schemes include [Shafer and Shenoy, 1990; Jensen *et al.*, 1990]. They trade off runtime and storage differently, making them suitable for certain uses. Darwiche [2009] demonstrates a connection between jtrees and VE, namely, the clusters of a dtree form a jtree. Taghipour *et al.* [2013] transfer the idea of dtrees to the first-order setting, introducing FO dtrees, allowing for a complexity analysis of lifted inference.

Lifted belief propagation (LBP) combines PP and lifting, often using lifted representations, e.g., with hyper-cubes [Singla and Domingos, 2008; Gogate and Domingos, 2010]. Kersting *et al.* [2009] and Ahmadi *et al.* [2013] present a counting LBP that runs a coloring algorithm with additional mechanisms for dynamic models. To the best of our knowledge, none of them use jtrees to focus on multiple queries.

Lifted inference sparks progress in various fields. Van den Broeck [2013] applies lifting to weighted model counting and first-order knowledge compilation, with newer work on asymmetrical models [van den Broeck and Niepert, 2015]. To scale lifting, Das *et al.* [2016] use graph data bases storing compiled models to count faster. Both works are interesting avenues for future work. Chavira and Darwiche [2007] focuses on knowledge compilation as well also addressing the setting of multiple queries and using local symmetries. Other areas incorporate lifting to enhance efficiency, including continuous or dynamic models [Choi *et al.*, 2010; Vlasselaer *et al.*, 2016], logic programming [Bellodi *et al.*, 2014], and theorem proving [Gogate and Domingos, 2011].

We [2016] apply lifting to jtrees introducing FO jtrees. Our algorithm does not include counting and handling of conjunctive queries. We widen the scope of the algorithm with our extension and speed up inference time.

3 Preliminaries

This section introduces basic notations, FO dtrees, and FO jtrees and recaps LVE and LJT based on [Taghipour, 2013; Braun and Möller, 2016]. We assume familiarity with common notions such as jtrees and dtrees (e.g., Darwiche [2009]).

3.1 Parameterized Models

Parameterized models compactly represent models with first-order constructs using logical variables (logvars) as parameters. We begin with denoting basic blocks.

Definition 1. Let $\mathbf{L}$ be a set of logvar names, Φ a set of factor names, and $\mathbf{R}$ a set of randvar names. A parameterized randvar (*PRV*) $R(L_1, \ldots, L_n), n \geq 0$, is a syntactical construct of a randvar $R \in \mathbf{R}$ combined with logvars $L_1, \ldots, L_n \in \mathbf{L}$ to represent a set of randvars that behave identically. Each logvar L has a domain, denoted $\mathcal{D}(L)$. The term $range(P)$ denotes the possible values of some PRV P. A *constraint* $(\mathbf{X}, C_{\mathbf{X}})$ is a tuple with a sequence of logvars $\mathbf{X} = (X_1, \ldots, X_n)$ and a set $C_{\mathbf{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X_i)$. C allows for restricting logvars to certain domain values. The symbol $\top$ marks that no restrictions apply and may be omitted.

Example 1. We model that people attend conferences and do research on some topic depending on whether this topic is considered hot using PRVs to encode identical potentials for a set of people, e.g., $alice$, eve, and bob. Given randvar names Hot, $AttC$, and DoR and logvar name X, we build PRVs Hot, $AttC(X)$, and $DoR(X)$. The domain of X is given by $\{alice, eve, bob\}$. Each PRV has the range $\{true, false\}$.

The term $logvars(P)$ refers to the logvars in some P. The term $randvars(P)$ refers to the randvars in P. The term $gr(P)$ denotes the set of instances of P with all logvars in P grounded w.r.t. constraints or domains. We use basic blocks to form more complex structures.

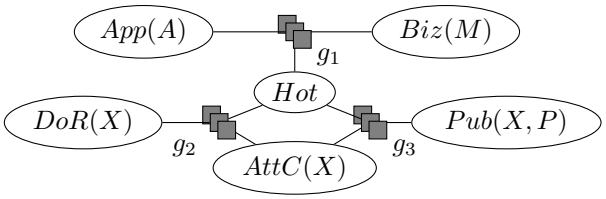

Figure 1: Parfactor graph for G_{ex}

Definition 2. A parametric factor (*parfactor*) g consists of a factor name, input arguments, possibly parameterized, and a function mapping argument values to a real value. We denote a parfactor by $\forall \mathbf{X} : \phi(\mathcal{A}) \mid C$ where $\mathbf{X} \in \mathbf{L}$ is a set of logvars that the factor generalizes over. $\mathcal{A} = (A_1, \ldots, A_n)$ is a sequence of PRVs, each PRV built from $\mathbf{R}$ and possibly $\mathbf{L}$. We omit $(\forall \mathbf{X} :)$ if $\mathbf{X} = logvars(\mathcal{A})$. $\phi : \times_{i=1}^{n} range(A_i) \mapsto \mathbb{R}^+$ is a function with name $\phi \in \Phi$. ϕ is identical for all instances of $\mathcal{A}$. For a full specification of ϕ, we have to list all input-output values for ϕ. C is a constraint on $\mathbf{L}$. A set of parfactors forms a *model* $G := \{g_i\}_{i=1}^{n}$. G represents the probability distribution $P_G = \frac{1}{Z} \prod_{f \in gr(G)} \phi_f(\mathcal{A}_f)$.

Example 2. With the above PRVs and a factor name ϕ, we build a parfactor $g = \phi(Hot, AttC(X), DoR(X))|\top$. Here, we omit explicit mappings for ϕ. The $\top$ constraint means ϕ holds for $alice$, eve, and bob. $gr(g)$ contains three factors with identical potential functions.

We compile a model G_{ex} building on Example 2: The topic allows for business markets and application areas. A person publishes on it in a publication. Logvars encode that there are several markets (M), areas (A), and publications (P).

Example 3. Let $\mathbf{L} = \{A, M, P, X\}$, $\Phi = \{\phi_1, \phi_2, \phi_3\}$, and $\mathbf{R} = \{Hot, Biz, App, AttC, DoR, Pub\}$. The domains are $\mathcal{D}(A) = \{ml, nlp\}$, $\mathcal{D}(M) = \{itsec, ehealth\}$, $\mathcal{D}(P) = \{p_1, p_2\}$, and $\mathcal{D}(X) = \{alice, eve, bob\}$. We build five binary PRVs with $n > 0$ and one with $n = 0$: Hot, $Biz(M)$, $App(A)$, $AttC(X)$, $DoR(X)$, and $Pub(X, P)$. The model reads $G_{ex} = \{g_1, g_2, g_3\}$,

- $g_1 = \phi_1(Hot, App(A), Biz(M))|C_1$,
- $g_2 = \phi_2(Hot, AttC(X), DoR(X))|C_2$, and
- $g_3 = \phi_3(Hot, AttC(X), Pub(X, P))|C_3$.

We omit concrete functions for ϕ_1, ϕ_2, and ϕ_3 at this point. C_1, C_2, and C_3 are $\top$ constraints, meaning ϕ_1, ϕ_2, and ϕ_3 apply for all possible tuples. Figure 1 depicts G_{ex} as a graph with six variable nodes for the PRVs and three factor nodes for g_1, g_2, and g_3 with edges to the PRVs involved.

The semantics of a model is given by grounding and building a full joint distribution. The query answering (QA) problem asks for a probability distribution of a randvar w.r.t. a model's joint distribution and fixed events (evidence). Formally, $P(Q|\mathbf{E})$ denotes a query where Q is a grounded PRV and $\mathbf{E}$ is a set of events (grounded PRVs with fixed range values). A query for G_{ex} is $P(Pub(eve, p_1)|attC(eve))$, with $attC(eve)$ a fixed event of eve attending conferences and asking for the probability distribution of eve publishing in p_1. Next, we look at algorithms for QA. They seek to avoid grounding as well as building a full joint distribution.

3.2 Lifted Variable Elimination

LVE employs two main techniques for QA, namely (i) decomposition into isomorphic subproblems and (ii) counting of domain values leading to a certain range value.

The first technique refers to lifted summing out. The idea is to compute VE for one case and then exponentiate the result with the number of isomorphic instances. The second technique, counting, exploits that all randvars of a PRV A evaluate to $range(A)$. We count the randvars evaluating to value $v \in range(A)$, forming a histogram.

Definition 3. We denote a counting randvar (*CRV*) P with inputs $\mathbf{X}$ and constraint C by $\#_{X \in C}[P(\mathbf{X})]$, where $logvars(\mathbf{X}) = \{X\}$ (meaning all other inputs are constant). The range of a CRV is the space of possible histograms. If $\{X\} \subset logvars(\mathbf{X})$, the CRV is a parameterized CRV (*PCRV*) and represents a set of CRVs. We *count-convert* a logvar X in a parfactor $\mathbf{L} : \phi(\mathcal{A})|C$ if converting a PRV $A_i \in \mathcal{A}$ into a CRV A'_i. In the new parfactor, ϕ' has a histogram h as input for A'_i. $\phi'(\dots, a_{i-1}, h, a_{i+1}, \dots)$ maps to $\prod_{a_i \in range(A_i)} \phi(\dots, a_{i-1}, a_i, a_{i+1}, \dots)^{h(a_i)}$ where $h(a_i)$ denotes the count of a_i in h.

The techniques have preconditions [Taghipour, 2013], e.g., to sum out PRV A in parfactor g, $logvars(A) = logvars(g)$. To count-convert logvar X in g, only one input in g contains X. Counting binds X, i.e., $logvars(\#_{X \in C}[P(\mathbf{X})]) = \mathbf{X} \setminus \{X\}$, possibly allowing summing out another PRV that we otherwise need to ground. LVE includes further techniques to enable lifted summing out. Grounding is its last resort where it replaces a logvar with each value in a constraint, duplicating the affected parfactors. Let us apply LVE to $g_1 \in G_{ex}$.

Example 4. In $\phi_1(Hot, App(A), Biz(M))$, we cannot sum out any PRV as neither includes both logvars. We employ counting to avoid grounding a logvar. For Hot and $App(a)$ of some area a, all randvars represented by $Biz(M)$ lead to true or false, making $Biz(M)$ a CRV. We can rewrite $Biz(M)$ into $\#_M[Biz(M)]$ and g_1 into $g'_1 = \phi'(Hot, App(A), \#_M[Biz(M)])|C_1$. The CRV refers to histograms that specify for each value $v \in range(Biz(M))$ how many grounded PRVs evaluate to v. Given the previous mappings $(hot, app, true) \mapsto x$ and $(hot, app, false) \mapsto y$ in ϕ, ϕ' maps $(hot, app, [n_1, n_2])$ to $x^{n_1} y^{n_2}$. Since M is no longer a regular logvar, we can sum out $App(A)$ using standard VE and exponentiating the result with $|\mathcal{D}(A)| = 2$.

To eliminate a next PRV, LVE chooses from operations applicable to the model based on the size of the intermediate result after applying an operation. Grounding has a high cost as it enlarges the model. A low cost usually applies to summing out, reducing the dimensions of the affected parfactor.

3.3 FO Dtrees

VE recursively decomposes a model into partitions that include randvars not part of any other partition. A dtree represents these decompositions. With lifting, a dtree needs to represent isomorphic instances as well. We do so by grounding a subset of the model logvars with representative objects, called decomposition into partial groundings (DPG; requires a normal form, see [Taghipour, 2013]). In an FO dtree, DPG nodes represent such DPGs.

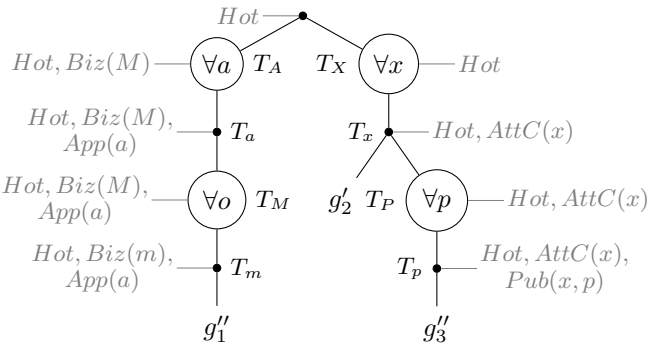

Figure 2: FO dtree for G_{ex} (clusters for inner nodes in gray)

Definition 4. A *DPG node* $T_{\mathbf{X}}$ is given by a 3-tuple $(\mathbf{X}, \mathbf{x}, C)$ where $\mathbf{X} = \{X_1, \dots X_k\}$ 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 objects* from $\mathcal{D}_{\mathbf{X}}$, and C is a constraint on $\mathbf{x}$ such that $\forall i, j : x_i \neq x_j$. We label $T_{\mathbf{X}}$ by $(\forall \mathbf{x} : C)$ in the FO dtree. $T_{\mathbf{X}}$ has a child $T_{\mathbf{x}}$. The decomposed model at $T_{\mathbf{x}}$ is a representative of $T_{\mathbf{X}}$ using a substitution $\theta = \{X_i \to x_i\}_{i=1}^k$ mapping $\mathbf{X}$ to $\mathbf{x}$.

Definition 5. An *FO dtree* for a model G is a tree in which (i) non-leaf nodes can be DPG nodes, (ii) each leaf contains a factor (parfactor with representative objects), (iii) each leaf with representative object x descends from exactly one DPG node $T_{\mathbf{X}}$ such that $x \in \mathbf{x}$, (iv) each leaf descending from DPG node $T_{\mathbf{X}}$ has all representative objects $\mathbf{x}$ in its factor, and (v) for each DPG node $T_{\mathbf{X}}$, $\mathbf{X} = \{X_1, \dots X_k\}$, $T_{\mathbf{x}}$ has $k!$ children $\{T_i\}_{i=1}^{k!}$, which are isomorphic up to permutation of $\mathbf{x}$. All leaf factors combined correspond to G.

The clusters of an FO dtree allow constructing an FO jtree. We compute clusters analogously to ground dtrees. A *cluster* of a node T is the union of its cutset and context. A cutset is the set of randvars shared between any two children minus the randvars in any ancestor cutset. A context is the intersection of its randvars and those in any ancestor cutset. We can count-convert logvars $\mathbf{X}$ if, at DPG node $T_{\mathbf{X}}$, $\mathbf{X}$ appear in the cluster at $T_{\mathbf{X}}$. Next, we inspect an FO dtree for G_{ex}.

Example 5. Figure 2 depicts an FO dtree without set braces and $\top$ constraints. The root partitions G_{ex} based on logvars with children $T_A = (A, a, \top)$ and $T_X = (X, x, \top)$. The models of both children share randvar Hot while the other PRVs appear in only one of them. T_A has a child T_a with model $\{g'_1 = \phi'_1(Hot, App(a), Biz(M))\}$ representative object a replacing A. Its child is a node $T_M = (M, m, \top)$ with child T_m and model $\{g''_1 = \phi''_1(Hot, App(a), Biz(m))\}$. g''_1 is ground so we have a leaf node. T_X has a child T_x with the model $\{g'_2 = \phi'_2(Hot, AttC(x), DoR(x))$, $g'_3 = \phi'_3(Hot, AttC(x), Pub(x, P))\}$. The children are a leaf node for g'_2 and a node $T_P = (P, p, \top)$ with child T_p and model $\{g''_3 = \phi''_3(Hot, AttC(x), Pub(x, p))\}$. g'_2 includes randvar $DoR(x)$ not part of the model under T_P, which in return contains $Pub(x, P)$. T_p has a leaf child for g''_3. The PRVs pinned to inner nodes are clusters. Leaf clusters consist of factor arguments. The clusters show that we can count-convert logvar M as M appears in the cluster of T_M.

3.4 FO Jtrees

LJT runs on FO jtrees using logvars to encode symmetries in FO dtree clusters. We define parclusters, a parameterized cluster, and FO jtrees, analogous to propositional jtrees.

Definition 6. A parcluster $\mathbf{C}$ is denoted by $\forall \mathbf{L} : \mathcal{A} \mid C$ where $\mathbf{L}$ is a set of logvars and $\mathcal{A}$ is a set of PRVs with $logvars(\mathcal{A}) \subseteq \mathbf{L}$. We omit ($\forall \mathbf{L}$:) if $\mathbf{L} = logvars(\mathcal{A})$. Constraint C puts limitations on logvars and representative objects. LJT assigns the parfactors of the input model to parclusters. A parfactor $\phi(\mathcal{A}_\phi)|C_\phi$ assigned to $\mathbf{C}$ must fulfill (i) $\mathcal{A}_\phi \subseteq \mathcal{A}$, (ii) $logvars(\mathcal{A}_\phi) \subseteq \mathbf{L}$, and (iii) $C_\phi \subseteq C$. We call the set of assigned parfactors a local model F.

Definition 7. An FO jtree for a model G is a pair $(\mathcal{J}, f_\mathbf{C})$ where $\mathcal{J}$ is a cycle-free graph and $f_\mathbf{C}$ is a function mapping each node i in $\mathcal{J}$ to a label $\mathbf{C}_i$ called a parcluster. An FO jtree must satisfy three properties: (i) A parcluster $\mathbf{C}_i$ is a set of PRVs from G. (ii) For every parfactor $g = \phi(\mathcal{A})|C$ in G, $\mathcal{A}$ appears in some $\mathbf{C}_i$. (iii) If a PRV from G appears in $\mathbf{C}_i$ and $\mathbf{C}_j$, it must appear in every parcluster on the path between nodes i and j in $\mathcal{J}$. Set $\mathbf{S}_{ij}$, called *separator* of edge i—j in $\mathcal{J}$, contains the shared randvars of $\mathbf{C}_i$ and $\mathbf{C}_j$.

An FO jtree is *minimal* if it ceases to be one if removing a PRV from any parcluster. The clusters of an FO dtree form a non-minimal FO jtree. To minimize, we merge neighboring nodes if one parcluster is a subset of the other. Grounding an FO jtree leads to a jtree that could have been built from a ground dtree. The set of factors in the grounded FO jtree is identical to the set of factors in the ground jtree.

3.5 Lifted Junction Tree Algorithm

LJT provides an efficient way for answering a set of queries $\mathbf{Q}$ given a model G and evidence $\mathbf{E}$. The main workflow is: (i) Construct an FO jtree for G. (ii) Enter $\mathbf{E}$. (iii) Pass messages. (iv) Compute answers for $\mathbf{Q}$.

FO jtree construction uses the clusters of an FO dtree for G. Message passing distributes local information at nodes to the other nodes. Two passes propagating information from the periphery to the inner nodes and back suffice [Lauritzen and Spiegelhalter, 1988]. LJT uses LVE to calculate the content of a message based on separators. If a node has received messages from all neighbors but one, it sends a message to the remaining neighbor (*inbound* pass). In the *outbound* pass, messages flow in the opposite direction. A query asks for the probability distribution (or the probability of a value) of a single grounded PRV, the query term. For each query, LJT finds a node whose parcluster contains the query term and sums out all non-query terms in its parfactors and received messages.

Since we extend LJT, we provide more details on the individual steps and an example in the next section.

4 Extended Lifted Junction Tree Algorithm

We extend LJT formally specifying its steps, incorporating counting and conjunctive queries. Algorithm 1 outlines LJT.

4.1 Construction

Constructing an FO jtree uses FO dtree clusters calculated using [Taghipour, 2013]. We formalize how we convert the clusters into parclusters and when and how merging proceeds.

Algorithm 1 Lifted Junction Tree Algorithm

function FOJT(Model G, Queries $\mathbf{Q}$, Evidence $\mathbf{E}$)
  FO jtree $\mathcal{J}$ = FO-JTREE(G)
  ENTEREVIDENCE($\mathcal{J}$,$\mathbf{E}$)
  PASSMESSAGES($\mathcal{J}$)
  GETANSWERS($\mathcal{J}$,$\mathbf{Q}$)
end function

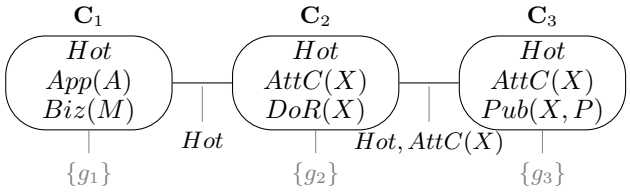

Figure 3: FO jtree for G_{ex} (parcluster models in gray)

Given a cluster $\mathcal{A}_T$ of an FO dtree node T, we construct a parcluster $\forall \mathbf{L} : \mathcal{A}|C$ by setting (i) $\mathcal{A} = \mathcal{A}_T$, (ii) $\mathbf{L} = logvars(\mathcal{A}_T)$, (iii) $C = \emptyset$, and (iv) $F_\mathbf{C} = \emptyset$. If T is an DPG node $(\mathbf{X}, \mathbf{x}, C_\mathbf{X})$, then $\mathbf{L} = logvars(\mathcal{A}_T) \cup \mathbf{X}$ and $C = C_\mathbf{X}$. If T is a leaf node with factor g, $F_\mathbf{C} = \{g\}$.

For *merging*, we need set relations and operations. A parcluster $\mathbf{C}_i$ is a *subset* of parcluster $\mathbf{C}_j$, denoted by $\mathbf{C}_i \subseteq \mathbf{C}_j$, iff $gr(\mathbf{C}_i) \subseteq gr(\mathbf{C}_j)$. Exploiting that parclusters have certain properties by way of construction (e.g., domains are either distinct or identical), we need not ground but check parclusters component-wise. Other relations and operations are defined analogously.

Parclusters $\mathbf{C}_i$ and $\mathbf{C}_j$ with models F_i and F_j are mergeable if $\mathbf{C}_i \subseteq \mathbf{C}_j \vee \mathbf{C}_j \subseteq \mathbf{C}_i$. The merged parcluster $\mathbf{C}_k$ and its model F_k are given by $\mathbf{C}_k = \mathbf{C}_i \cup \mathbf{C}_j$ and $F_k = F_i \cup F_j$. The new node k takes over all neighbors of i and j. If we merge two parcluster, one with logvars $\mathbf{X}$ and one with symbolic constants $\mathbf{x}$, we first perform the inverse of the substitution $\theta = \{X_i \to x_i\}_{i=1}^k$ performed at DPG node $T_\mathbf{X}$ in the underlying FO dtree, mapping $\mathbf{x}$ back onto $\mathbf{X}$.

Example 6. After converting the clusters in Fig. 2 into parclusters, we look at the leaf node with local model $\{g_3''\}$. As the neighboring parcluster is identical, we merge them. Keeping them separate would mean sending a message with g_3'' leading to two nodes with identical information. We merge the next neighbor as well but replacing p with P again. Merging continues until we reach the node corresponding to the root in the FO dtree. The node with g_2' in its local model does not merge since its parcluster includes PRV $DoR(x)$ (but applies $x \mapsto X$). The same procedure iteratively merges the nodes containing PRVs Hot, $App(A)$, and $Biz(M)$. Fig. 3 shows the final result with three parclusters, (i) $\mathbf{C}_1 = \forall A, O : \{Hot, App(A), Biz(M)\}|\top$, (ii) $\mathbf{C}_2 = \forall X : \{Hot, AttC(X), DoR(X)\}|\top$, and (iii) $\mathbf{C}_3 = \forall X, P : \{Hot, AttC(X), Pub(X, P)\}|\top$. The separators are $\mathbf{S}_{12} = \mathbf{S}_{21} = \{Hot\}$ and $\mathbf{S}_{23} = \mathbf{S}_{32} = \{Hot, AttC(X)\}$. Each local model consists of one parfactor, which is not a common scenario. It is a worst case for message passing with a maximum number of messages and a best case for query answering with minimal local models.

Counting We allow (P)CRVs in the input model, increasing its expressivity. (P)CRVs facilitate specifying counting behavior explicitly in the model description. Construction handles (P)CRVs along with PRVs, becoming part of parclusters and possibly separators. Though we can identify logvars for counting conversion in the FO dtree, we do not use this feature as explained in the next subsection.

4.2 Message Passing

We give a formal definition of a message, discuss the effects of counting on messages, and adapt the heuristic selecting the next step during LVE for calculating a message.

For a message from node i to node j, we pack the information present at i into parfactors over separator $\mathbf{S}_{ij}$ since j can process the PRVs in $\mathbf{S}_{ij}$. Formally, a *message* m_{ij} from i with parcluster $\mathbf{C}_i$ and local model F_i to j is a set of parfactors, each with a subset of $\mathbf{S}_{ij}$ as arguments. To calculate m_{ij}, we eliminate all PRVs not in $\mathbf{S}_{ij}$ from F_i and the messages from all other neighbors using LVE, as described by

$$m_{ij} = \sum_{E \in \mathbf{E}_i} \prod_{g \in F'} g,\ \mathbf{E}_i = \mathbf{C}_i \setminus \mathbf{S}_{ij},\ F' = F \cup \{m_{ik}\}_{k \neq j}.$$

m_{ij} can be a set of parfactors as LVE multiplies parfactors if necessary. Let us look at messages in the FO jtree for G_{ex}.

Example 7. Messages flow from nodes 1 and 3 to node 2 and back. Messages between nodes 1 and 2 have the argument Hot, between nodes 2 and 3 the arguments Hot and $AttC(X)$. Inbound, the messages are m_{12} and m_{32}. For m_{12}, we sum out $\mathbf{E}_1 = \{App(A), Biz(M)\}$ from $F' = F_1$ as in Example 4. For m_{32}, we sum out $\mathbf{E}_3 = \{Pub(X, P)\}$ from $F' = F_3$ using lifted summing out on $Pub(X, P)$. At this point, node 2 has all information in the model in its local model and received messages, encoded in its parcluster PRVs. Outbound, node 2 propagates this information to node 1 with message m_{21} and to node 3 with message m_{23}. For m_{21}, we sum out $\mathbf{E}_2 = \{AttC(X), DoR(X)\}$ from $F' = F_2 \cup \{m_{32}\}$ and for m_{23}, $\mathbf{E}_2 = \{DoR(X)\}$ from $F' = F_2 \cup \{m_{12}\}$.

Counting As mentioned above, counting appears in the form of (P)CRVs in an input model. As part of a model, (P)CRVs appear during message passing in a separator or in the set of randvars to eliminate. A (P)CRV in a separator does not have any special effect on messages. Eliminating a (P)CRV can cause groundings: Assume we eliminate a (P)CRV from a parfactor that also includes a separator PRV with additional logvars. One logvar of the separator PRV may be count-convertible but we may need to ground further logvars. These groundings are necessary within LJT but possibly unnecessary if doing LVE on the original model.

We do not count-convert logvars identified for counting conversion in the FO dtree. Consider a scenario where PRVs $App(A)$ and $Biz(M)$ are in a parcluster and $App(A)$ in one separator. Assume given an FO dtree, we converted $Biz(M)$ into a CRV. Then, we still need to count-convert $App(A)$ to sum out $Biz(M)$, making the counting conversion of logvar M superfluous. Since we cannot always determine from the clusters in the FO dtree if counting conversion is reasonable for message passing, we do not count-convert in the FO dtree.

Algorithm 2 Conjunctive Query Answering

```
function GETANSWERS(FO Jtree 𝒥, Queries Q)
    for 𝒬 ∈ Q do
        Subtree 𝒥^𝒬 ← GETSUBTREE(𝒥, 𝒬)
        Model G^𝒬 ← GETMODEL(𝒥^𝒬)
        GETANSWER(G^𝒬, 𝒬)
    end for
end function
```

Example 7 references another use of counting, namely, as a means to enable a sum-out operation after counting conversion when calculating a message. Without counting, the algorithm would need to ground a logvar. After counting conversion, the new (P)CRV becomes part of the model that is used for further calculating the message. Then, the scenario plays out as described above when (P)CRVs are part of the model itself. The new (P)CRV either needs to be eliminated or becomes part of the message if the original PRV is part of the separator. If the new (P)CRV is part of the message, it becomes part of message calculations at the receiving node.

Heuristic The heuristic LVE uses no longer works for LJT in all cases. Consider the scenario from before with PRVs $App(A)$ and $Biz(M)$ in a parcluster and $App(A)$ in a separator. Using counting conversion on A, we can sum out $Biz(M)$. Assume that A has 50 domain values while O has 10. LVE would count-convert M, after which it still cannot sum out $\#_M[Biz(M)]$, count-convert A, and then sum out $\#_M[Biz(M)]$, making the first counting conversion unnecessary. For LJT, we require the heuristic to consider the PRVs, especially their logvars, in a separator.

We adapt the heuristic by dividing applicable counting operations into one part with operations for PRVs to eliminate and another part with operations for separator PRVs. If the operation with the lowest cost comes from the first part, we select the cheapest operation from the second part if not empty. With the adapted heuristic, we save superfluous applications of LVE operators.

4.3 Query Answering

We extend query answering to include conjunctive queries, allowing for multiple grounded PRVs Q in a query. A query now may have the form $P(\mathcal{Q}|\mathbf{E})$ with a set of PRVs $\mathcal{Q}$ and a set of fixed events $\mathbf{E}$. For LJT, the extension means the input $\mathbf{Q}$ consists of sets of PRVs $\mathcal{Q}$ that need an answer.

With a set of PRVs in a query, we can have query randvars that are not part of one parcluster. We could force LJT to build an FO jtree with all query randvars in one parcluster but the forced construction inhibits fast query answering for other queries. Additionally, it assumes that we know a query in advance. Hence, we adapt the idea of so called out-of-clique inference by Koller and Friedman [2009] where we extract necessary information per query from a standard jtree.

Algorithm 2 shows a pseudo code description of our approach. We find a subtree of the FO jtree that covers all query randvars. From the parclusters in the subtree, we extract a model to answer $\mathcal{Q}$ with LVE handling multiple query randvars. A more detailed description of each step follows.

Subtree Identification The goal is to find a subtree of the FO jtree where the subtree parclusters cover all query randvars $\mathcal{Q}$. Since the subtree is the basis for model extraction, we want a subtree that results in the smallest model possible in terms of number of PRVs. Investigating ways of finding such a subtree is part of future work.

In a straight forward way, we find a first node that covers at least part of $\mathcal{Q}$, use this node as the first node in the subtree $\mathcal{J}^{\mathcal{Q}}$, and add further nodes that cover still missing query randvars closest to the current $\mathcal{J}^{\mathcal{Q}}$.

Model Extraction We build a model $G^{\mathcal{Q}}$ from subtree $\mathcal{J}^{\mathcal{Q}}$ avoiding duplicate information. We use the local models at the nodes in $\mathcal{J}^{\mathcal{Q}}$ and the messages that the nodes at the borders of $\mathcal{J}^{\mathcal{Q}}$ received from outside $\mathcal{J}^{\mathcal{Q}}$.

Since LJT assigns each parfactor in G to exactly one parcluster, the local models hold no duplicate information. The border messages store all information from outside the subtree. Ignoring the messages within the subtree, we do not duplicate information through a message.

Query Answering Using the model $G^{\mathcal{Q}}$ built during model extraction, we perform LVE to answer a query over the randvars $\mathcal{Q}$. Though LVE as described by Taghipour [2013] does not explicitly mention conjunctive queries, the formalism allows for multiple query randvars. Query answering needs an operation called shattering that splits the parfactors in a model based on query randvars. For one query randvar Q, a split means we add a duplicate of each parfactor that covers Q and use the constraint to restrict the PRV in one parfactor to Q and the other to the remaining instances of the PRV. Multiple query randvars mean a finer granularity in the model after shattering, leading to more operations during LVE.

To compute an answer to a conjunctive query over $\mathcal{Q}$, we shatter $G^{\mathcal{Q}}$ based on $\mathcal{Q}$. We use LVE to compute a joint probability for $\mathcal{Q}$ and normalize. The algorithm still works with single query randvars Q as we find a node i that covers Q, extract a model, namely the local model F_i and all messages to i, and perform LVE.

Example 8. After message passing, we can answer conjunctive query $P(DoR(eve), Pub(eve, p_1))$. Nodes 2 and 3 cover the query randvars. The extracted model $G^{\mathcal{Q}}$ consists of F_2, F_3, and m_{12}. We shatter $G^{\mathcal{Q}}$ w.r.t. $\mathcal{Q}$, leading to five parfactors. We sum out $Pub(X, P)$, $X \neq eve$ and $P \neq p_1$ from the g_3 duplicate where X and P are not equal to eve and $article$, resulting in a parfactor g' with arguments Hot and $AttC(X)$, $X \neq eve$. Next, we sum out $DoR(X)$, $X \neq eve$, from the g_2 duplicate without eve, resulting in a parfactor g'' with arguments Hot and $AttC(X)$, $X \neq eve$. Summing out $AttC(X)$, $X \neq eve$, from the product of g' and g'' yields a parfactor g''' with argument Hot. Summing out $AttC(eve)$ from the product of g_2 and g_3 where $X = eve$ and $P = p_1$ yields a parfactor $\hat{g}$ with arguments Hot, $DoR(eve)$, and $Pub(eve, p_1)$. Last, we multiply m_{12}, g''', $\hat{g}$, sum out Hot, and normalize, leading to the queried distribution.

For simple query $P(AttC(eve))$, we can use node 2. We sum out $DoR(X)$, Hot, and $AttC(X)$ where $X \neq eve$ from $F \cup \{m_{12}, m_{32}\}$ after shattering.

5 Theoretical Analysis

We look at soundness and best and worst case scenarios of LJT extended with counting and conjunctive queries.

Soundness For the soundness of our LJT version, we assume that the junction tree algorithm, LVE, and Taghipour's work on FO dtrees and clusters are sound.

Theorem 1. LJT with counting is sound, i.e., is equivalent to inference using a ground inference algorithm.

Proof sketch. Each input model needs to fulfill a specific normal form for FO dtree construction. Given the normal form, LJT constructs a sound FO dtree with correct clusters. Converting the clusters into parclusters preserves the correctness of the clusters and leads to a sound FO jtree with each parfactor of the input model assigned to exactly one node. Merging models preserves the FO jtree soundness. Additionally, local models fulfill the normal form. With sound LVE operations, the algorithm carries out sound computations at the local models. With a sound FO jtree, information sent between nodes is sound and due to the normal form, is interpreted correctly at the receiving end. With sound information at the nodes, LJT computes a correct answer for a query. □

Theorem 2. LJT with conjunctive queries is sound, i.e., is equivalent to inference using a ground inference algorithm.

Proof sketch. Given that LJT is sound, we have sound information at the nodes in the FO jtree. By way of constructing the model for the query randvars in a query, we combine all necessary information without duplicates. Given that LVE is sound, LJT computes a correct answer for a query. □

Best and Worst Case Scenario LJT complete with counting allows for efficient query answering given multiple queries. It imposes some static overhead due to FO jtree construction and message passing. After these steps, we can answer queries based on smaller models compared to the input model until the input model or evidence changes. We currently investigate incrementally changing information.

Characteristics that influence runtimes include (i) during construction, the number of logvars and parfactors in G, (ii) during message passing, the number of nodes in the FO jtree, the size of the parclusters, and the degree of each node, and (iii) during query answering, the size of the model used for a query and the effort spent on building the model. The goal is to have efficient query answering with smallest models possible and spend effort on construction and message passing only once per input model (and evidence set).

In a worst case scenario, as is for LVE, we need to ground all logvars in the model and perform inference at a propositional level to calculate correct results. In such a case, we cannot avoid groundings. Unfortunately, LJT may induce unnecessary groundings during message passing because calculating a message over PRVs with logvars may inhibit a reasonable elimination order. For a lifted run, the logvars of a PRV to eliminate need to be a superset of the logvars in affected separator PRVs. A separator PRV with the most logvars of all

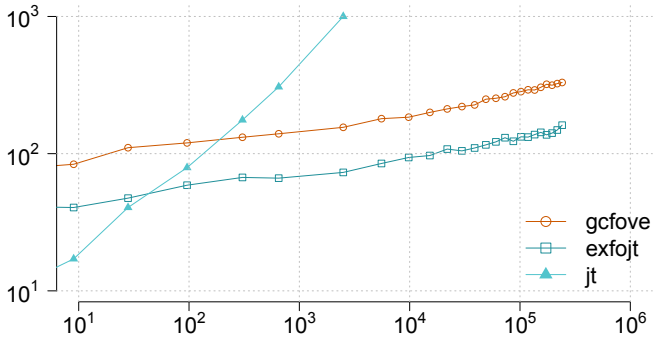

Figure 4: Runtimes [ms] with $|gr(G_{ex})|$ ranging from 3 to $241,000$ on log scales accumulated over 6 queries

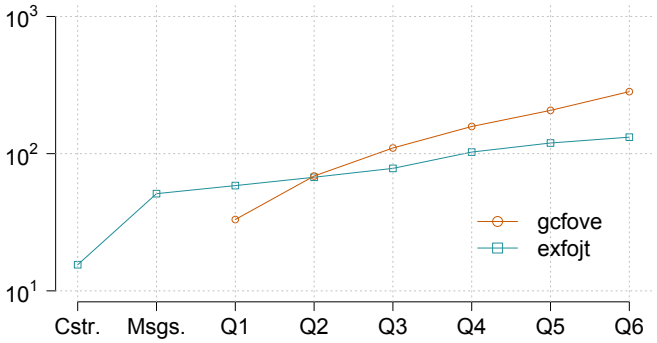

Figure 5: Runtimes [ms] on log scale accumulated over 6 queries with $|gr(G_{ex})| = 102,050$

PRVs in a parcluster automatically results in at least a counting conversion and, in the worst case, groundings. And since received messages are part of further calculations, groundings might carry forward. The results are still correct, but the LJT run degrades to a propositional algorithm run. We currently work on preventing unnecessary groundings.

For simple queries, we gain the most if the model permits an FO jtree with few PRVs per parcluster. With a clever access function, we quickly identify a parcluster for the query and sum out the few non-query PRVs. For conjunctive queries, the best case is when the nodes that cover all query PRVs are close together such that we find the subtree fast and build a model with few PRVs to eliminate. Needing the whole tree represents the worst case as we build a model equal to the original model and do standard LVE, adding overhead without payoff. Over many queries, LJT offsets queries needing a large model with queries needing a small model.

6 Empirical Evaluation

We have implemented a prototype of LJT with our extensions, named `exfojt` in this section. Taghipour provides a baseline implementation of GC-FOVE including its operators (available at `https://dtai.cs.kuleuven.be/software/gcfove`), named `gcfove`, which we use to test our implementation against. We include the `gcfove` operators in `exfojt`. We also implemented the propositional junction tree algorithm, named `jt`, as a reference point.

We use G_{ex} as input. Standard lifting examples such as the smokers model are too simple, leading to an FO jtree with one node. Runtimes of `exfojt` compared to `gcfove` are slightly higher due to the static overhead for constructing an FO dtree, converting its clusters to parclusters, and then merging them into one node. Query answering takes the same time for each query as both carry out the same operations.

We vary the domain sizes, yielding grounded model sizes $|gr(G_{ex})|$ between 3 and $241,000$. We query each PRV once, resulting in 6 queries: (i) Hot, (ii) $Biz(m_1)$, (iii) $App(a_1)$, (iv) $DoR(x_1)$, (v) $AttC(x_1)$, and (vi) $Pub(x_1, p_1)$. The result for queries using other groundings (within the current model size) would be identical.

We compare runtimes for inference accumulated over the given queries, averaged over several runs. We do not compare against the original LJT version since our example model leads to groundings that carry forward during message passing. Its runtime comes close to the runtime of `jt`.

`exfojt` constructs an FO jtree and passes messages once. Then, it answers the given queries based on the local models and messages. `jt` follows the same protocol with propositional data structures and VE operations. `gcfove` eliminates all non-query randvars from G_{ex} for each query.

Figure 4 shows runtimes for inference with $|gr(G_{ex})|$ on the x-axis, ranging from 3 to $241,000$, both on log scale. The squares mark the runtimes for `gcfove`, the circles the runtimes for `exfojt`, and the filled triangles the runtimes for `jt`. With small models, `jt` outperforms both lifted approaches. With an increase in the grounded model size, memory and time requirements of `jt` surge. `exfojt` outperforms `gcfove` on all grounded model sizes, needing 43% to 51% of the time `gcfove` requires. The savings in runtime are mirrored in the number of LVE operations performed, with a maximum of 63 by `gcfove` versus 46 by `exfojt`. `exfojt` trades off runtime with storage, needing slightly more memory to store the underlying FO jtree.

Since `exfojt` has some static overhead, we look at what point `exfojt` outperforms `gcfove`. Figure 5 shows runtimes on log scale accumulated over the six queries for $|gr(G_{ex})| = 102,050$, an average model in terms of performance for both implementations. We ordered the given queries by increasing runtime for `gcfove`. With the second query, `gcfove` needs marginally more time. With each passing query, `exfojt` saves more time compared to `gcfove`.

Conjunctive queries that `exfojt` answers using one parcluster have similar runtimes compared to the simple queries from above. With more complex queries that require more than one parcluster, the runtimes rise since subtree identification takes longer and the models become larger. We do not compare runtimes for conjunctive queries as `gcfove` does not support queries with more than one query randvar.

In summary, even in our small example model and only a prototype implementation, spending effort on building an FO jtree and passing messages pays off. LJT has even more potential when considering scenarios where the FO jtree structure remains the same and only parts of a model or prior information changes.

7 Conclusion

We present extensions to LJT to answer multiple queries efficiently in the presence of symmetries in a model. We formally specify the different steps of LJT and incorporate the lifting tool of counting to lift computations where LJT previously needed to ground. We extend the scope of LJT by allowing conjunctive queries and handling them efficiently. These extensions provide us with a deeper understanding of how LVE and FO jtrees interact. If a model has a full lifted LJT run, we speed up runtimes significantly for answering multiple queries compared to the original LJT and GC-FOVE.

We currently work on adapting LJT to incrementally changing models. Other interesting algorithm features include parallelization, construction using hypergraph partitioning, and different message passing strategies as well as using local symmetries. Additionally, we look into areas of application to see its performance on real-life scenarios.

References

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.

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.

Tanya Braun and Ralf Möller. Lifted Junction Tree Algorithm. In *KI 2016: Advances in Artificial Intelligence - 39th Annual German Conference on AI, Klagenfurt, Austria, September 26–30, 2016*, pages 30–42, 2016.

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, 2007.

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, 2010.

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, 2016.

Rodrigo de Salvo Braz. *Lifted First-order Probabilistic Inference*. PhD thesis, University of Illinois at Urbana Champaign, 2007.

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, 2011.

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.

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, 2009.

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.

Brian Milch, Luke S. Zettelmeyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted Probabilistic Inference with Counting Formulas. In *AAAI-08 Proceedings of the 23rd Conference on Artificial Intelligence*, pages 1062–1068, 2008.

David Poole and Nevin L. Zhang. Exploiting Contextual Independence in Probabilistic Inference. *Jounal of Artificial Intelligence*, 18:263–313, 2003.

Glenn R. Shafer and Prakash P. Shenoy. Probability Propagation. *Annals of Mathematics and Artificial Intelligence*, 2(1):327–351, 1990.

Parag Singla and Pedro Domingos. Lifted First-order Belief Propagation. In *AAAI-08 Proceedings of the 23rd Conference on Artificial Intelligence*, pages 1094–1099, 2008.

Nima Taghipour, Jesse Davis, and Hendrik Blockeel. First-order Decomposition Trees. In *Advances in Neural Information Processing Systems 26*, pages 1052–1060. Curran Associates, Inc., 2013.

Nima Taghipour. *Lifted Probabilistic Inference by Variable Elimination*. PhD thesis, KU Leuven, 2013.

Guy van den Broeck and Mathias Niepert. Lifted Probabilistic Inference for Asymmetric Graphical Models. In *AAAI-15 Proceedings of the 29th Conference on Artificial Intelligence*, pages 3599–3605, 2015.

Guy van den Broeck. *Lifted Inference and Learning in Statistical Relational Models*. PhD thesis, KU Leuven, 2013.

Jonas Vlasselaer, Wannes Meert, Guy van den Broeck, and Luc De Raedt. Exploiting Local and Repeated Structure in Dynamic Baysian Networks. *Artificial Intelligence*, 232:43–53, 2016.

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, 1994.