Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm

Tanya Braun and Ralf Möller

Institute of Information Systems, Universität zu Lübeck, Lübeck
{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 the current 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 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 have introduced a lifted junction tree algorithm (LJT) for multiple queries on models with first-order constructs [3]. LJT is based on the junction tree algorithm [17] and lifted variable elimination (LVE) as specified in [26]. LJT uses 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 construction, message passing, and query answering. We incorporate counting as defined in [26] to lift more computations and allow a wider variety of model specifications. We adapt the LVE heuristic for elimination order for message passing and extend query answering for conjunctive queries that may cover multiple clusters based on [16].

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 more complex queries. We significantly speed up runtime compared to LVE and LJT. Overall, we handle multiple queries more efficiently than 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) [28]. VE decomposes a model into subproblems to evaluate them in an efficient order. A decomposition tree (dtree) represents such a decomposition [9]. LVE, first introduced in [19] and expanded in [20], exploits symmetries at a global level. LVE saves computations by reusing intermediate results for isomorphic subproblems. Milch *et al.* introduce counting to lift certain computations where lifted summing out is not applicable [18]. Taghipour *et al.* extend the formalism to its current standard by generalising counting [26]. He formalises lifting by defining lifting operators. The operators appear in internal calculations of LJT.

For multiple queries in a propositional setting, Lauritzen and Spiegelhalter introduce junction trees (jtrees), a representation of clusters in a propositional model, along with a reasoning algorithm [17]. 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. Shafer and Shenoy as well as Jensen *et al.* propose well known PP schemes [21,14]. They trade off runtime and storage differently, making them suitable for certain uses. Darwiche demonstrates a connection between jtrees and VE, namely, the clusters of a dtree form a jtree [10]. Taghipour *et al.* transfer the idea of dtrees to the first-order setting, introducing FO dtrees, allowing for a complexity analysis of lifted inference [25].

Lifted belief propagation (LBP) combines PP and lifting, often using lifted representations, e.g., with hyper-cubes [23,12]. Kersting and Ahmadi *et al.* present a counting LBP that runs a colouring algorithm with additional mechanisms for dynamic models [15,1]. 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 applies lifting to weighted model counting [5] and first-order knowledge compilation, with newer work on asymmetrical models [6]. To scale lifting, Das *et al.* use graph data bases storing compiled models to count faster [11]. Both works are interesting avenues for future work. Chavira and Darwiche focus on knowledge compilation as well also addressing the setting of multiple queries and using local symmetries [7]. Other areas incorporate lifting to enhance efficiency, including continuous or dynamic models [8,27], logic programming [2], and theorem proving [13].

We apply lifting to jtrees, introducing FO jtrees and provide LJT as a reasoning algorithm using LVE as a subroutine [3]. Currently, LJT 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, the FO dtree and FO jtree data structures, and recaps LVE and LJT based on [26,3]. We assume familiarity with common notions such as jtrees and dtrees (for an introduction, see, e.g., [10]).

3.1 Parameterised Models

Parameterised models compactly represent models with first-order constructs using logical variables (logvars) as parameters. We begin with denoting basic blocks on our way to build a full model.

Definition 1. *Let* $\mathbf{L}$ *be a set of logvar names,* Φ *a set of factor names, and* $\mathbf{R}$ *a set of randvar names. A parameterised randvar* (PRV) $R(L_1, \dots, L_n), n \geq 0$*, is a syntactical construct of a randvar* $R \in \mathbf{R}$ *combined with logvars* $L_1, \dots, L_n \in \mathbf{L}$ *to represent a set of randvars that behave identically. Each logvar* L *has a domain, denoted by* $\mathcal{D}(L)$*. The term* $range(A)$ *denotes the possible values of some PRV* A*. A* constraint $(\mathbf{X}, C_{\mathbf{X}})$ *is a tuple with a sequence of logvars* $\mathbf{X} = (X_1, \dots, X_n)$ *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.*

The terms $lv(P)$ and $rv(P)$ refer to the logvars and PRVs with constraints, respectively, in some P. The term $gr(P)$ denotes the set of instances of P with all logvars in P grounded w.r.t. constraints or domains. Let us look at an example.

Example 1. We model that people attend conferences and do research on some topic depending on whether this topic is considered hot. Later, we want to encode that, e.g., the potential increases that a person does research on some topic if this topic is hot. The potential should be identical for different people. So, we use PRVs to represent a set of people, e.g., *alice*, *eve*, and *bob*, that have the same behaviour for some randvar.
Given randvar names $HoTpc$, $AttCnf$, and Res and logvar name X, we build PRVs $HoTpc$, $AttCnf(X)$, and $Res(X)$. The domain of X is given by $\mathcal{D}(X) = \{alice, eve, bob\}$. Each PRV has the range $\{true, false\}$. $HoTpc$ is not parameterised and represents a propositional randvar, while $AttCnf(X)$ and $Res(X)$ represent sets of randvars. A constraint can modify which randvars $AttCnf(X)$ and $Res(X)$ represent. A constraint $C = \top$ for X does not restrict X. A constraint $C' = (X, \{alice, eve\})$ restricts X to not take the value *bob*. $gr(Res(X)|C')$ contains $Res(alice)$ and $Res(eve)$ while $gr(Res(X)|C)$ also contains $Res(bob)$.

We use the basic blocks of logvars, PRVs, and constraints to form more complex structures that make up a model.

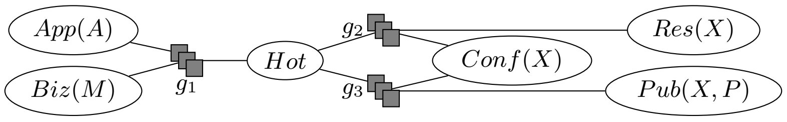

Fig. 1: Parfactor graph for G_{ex}

Definition 2. *A parametric factor* (parfactor) g *consists of a function mapping argument values to real values. 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 generalises over.* $\mathcal{A} = (A_1, \dots, A_n)$ *is a sequence of PRVs. Each PRV is built from* $\mathbf{R}$ *and possibly* $\mathbf{X}$*. We omit* $(\forall \mathbf{X} :)$ *if* $\mathbf{X} = lv(\mathcal{A})$*.* $\phi : \times_{i=1}^{n} range(A_i) \mapsto \mathbb{R}^+$ *is a function with name* $\phi \in \Phi$*, identical for all instances of* $\mathcal{A}$*. A full specification of* ϕ *requires listing all input-output values.* 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* $P_G = \frac{1}{Z} \prod_{f \in gr(G)} \phi_f(\mathcal{A}_f)$ *with* Z *as the normalisation constant.*

For our above example of *alice*, *eve*, and *bob* doing research and attending conferences influenced by a topic considered hot, we can build a parfactor that encodes this identical behaviour.

Example 2. With the above PRVs and a factor name ϕ, we build a parfactor $g = \phi(HoTpc, AttCnf(X), Res(X))|\top$. The mappings with randomly chosen potentials are (with $true = 1$ and $false = 0$:

$$\begin{aligned} &(0,0,0) \to 10, \quad (0,0,1) \to 3, \quad (0,1,0) \to 3, \quad (0,1,1) \to 7, \\ &(1,0,0) \to 6, \quad\ \ (1,0,1) \to 6, \quad (1,1,0) \to 5, \quad (1,1,1) \to 9 \end{aligned}$$

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 and for a person to publish in publications. 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 $\mathbf{R} = \{HoTpc, Biz, App, AttCnf, Res, Pub\}$. The domains for the logvars are $\mathcal{D}(A) = \{ml, nlp\}$, $\mathcal{D}(M) = \{itsec, ehealth\}$, $\mathcal{D}(P) = \{p_1, p_2\}$, and $\mathcal{D}(X) = \{alice, eve, bob\}$. In addition to $HoTpc$, $AttCnf(X)$, and $Res(X)$, we build the binary PRVs $Biz(M)$, $App(A)$, and $Pub(X, P)$. The model reads $G_{ex} = \{g_1, g_2, g_3\}$,

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

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 (a normal randvar) and $\mathbf{E}$ is a set of events (grounded PRVs with fixed range values). A query for G_{ex} is $P(Pub(eve, p_1)|AttCnf(eve) = true)$, with $AttCnf(eve) = true$ 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 of PRV given the remaining PRVs in a parfactor. 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 instances of a PRV A evaluate to $range(A)$. A counting randvar (CRV) encodes for n interchangeable randvars, i.e., instances of A, how many have a certain value.

Example 4. Consider $\phi(R_1, R_2, R_3)$ with mappings as follow:

$$(0,0,0) \to 1,\ \ (0,0,1) \to 2,\ \ (0,1,0) \to 2,\ \ (0,1,1) \to 3,$$
$$(1,0,0) \to 2,\ \ (1,0,1) \to 3,\ \ (1,1,0) \to 3,\ \ (1,1,1) \to 4$$

The potentials for $(0,0,1)$, $(0,1,0)$, and $(1,0,0)$ are identical, namely 2, and the argument values exhibit that two of them are $false$ and one is $true$. The same observation holds for two arguments being $true$ and one $false$, all mapping to the potential of 3. So, instead of using eight mappings, we use a histogram to encode how many of the R randvars have a specific value that maps to the corresponding potential (first position $R = 1$, second $R = 0$):

$$[0,3] \to 1,\ \ [1,2] \to 2,\ \ [2,1] \to 3,\ \ [3,0] \to 4$$

To refer to a set of randvars in this counted version, we use a CRV.

Definition 3. *We denote a* CRV *by* $\#_{X \in C}[P(\mathbf{X})]$ *for a PRV* $P(\mathbf{X})$ *and constraint* C*, where* $lv(\mathbf{X}) = \{X\}$ *(meaning all other inputs are constant). The range of a CRV is the space of possible histograms. Since counting binds logvar* X*,* $lv(\#_{X \in C}[P(\mathbf{X})]) = \mathbf{X} \setminus \{X\}$*. A histogram* h *is a set of tuples* $\{(v_i, n_i)\}_{i=1}^m$*,* $m = |range(P(\mathbf{X}))|$*,* $n_i \in \mathbb{N}$*, and* $\sum_i n_i = |gr(P(\mathbf{X})|C)|$*. A shorthand notation is* $[n_1, \ldots, n_m]$*.* $h(v_i)$ *returns* n_i*. If* $\{X\} \subset lv(\mathbf{X})$*, the CRV is a parameterised CRV* (PCRV) *and represents a set of CRVs. We* count-convert *a logvar* X *in a PRV* $A_i \in \mathcal{A}$ *in a parfactor* $\mathbf{L} : \phi(\mathcal{A})|C$ *leading to a CRV* A'_i*. In the new parfactor,* ϕ' *has a histogram* h *as input for* A'_i*.* $\phi'(\ldots, a_{i-1}, h, a_{i+1}, \ldots)$ *maps to* $\prod_{a_i \in range(A_i)} \phi(\ldots, a_{i-1}, a_i, a_{i+1}, \ldots)^{h(a_i)}$*.*

The techniques have preconditions [26], e.g., to sum out PRV A in parfactor g, $lv(A) = lv(g)$. To count-convert logvar X in g, only one input in g contains X. Counting binds X, i.e., $lv(\#_{X \in C}[P(\mathbf{X})]) = \mathbf{X} \setminus \{X\}$, 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. 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. Let us apply LVE to $g_1 \in G_{ex}$.

Example 5. In $\phi_1(HoTpc, App(A), Biz(M))$, we cannot sum out any PRV as neither includes both logvars. One may ground M, leading to $|gr(M)|$ parfactors of the form $\phi(HoTpc, App(A), Biz(m))$ for all $m \in gr(M)$. To eliminate $App(A)$, one multiplies all new parfactors into one with $HoTpc$, $App(A)$, and all instances of $Biz(M)$ as arguments. All randvars represented by $Biz(M)$ lead to true or false and we can count them as in Example 4. We count convert to avoid the grounding step. We can rewrite $Biz(M)$ into $\#_M[Biz(M)]$ and g_1 into $g_1' = \phi'(HoTpc, App(A), \#_M[Biz(M)])|C_1$. 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 sum out $App(A)$ using standard VE and exponentiate the result with $|gr(A)| = 2$.

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 [26]). In an FO dtree, DPG nodes represent such DPGs.

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* $\mathbf{x}$*.*

With a means to represent isomorphic instances and as such, lifted summing out in a dtree, we define FO dtrees for decompositions of a model during LVE.

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*

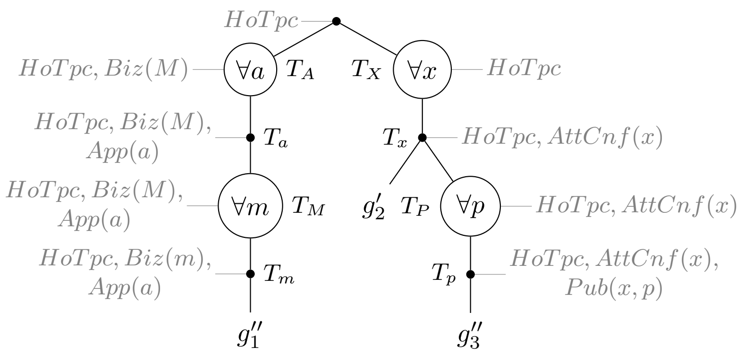

Fig. 2: FO dtree for G_{ex} (clusters for inner nodes in gray)

$T_{\mathbf{X}}$, $\mathbf{X} = \{X_1, \dots X_k\}$, $T_{\mathbf{x}}$ 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 form 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 6. 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 $HoTpc$ while the other PRVs appear in only one of them. T_A has a child T_a with model $\{g'_1 = \phi'_1(HoTpc, App(a), Biz(M))\}$, representative object a replacing A. Child node $T_M = (M, m, \top)$ has a child T_m with model $\{g''_1 = \phi''_1(HoTpc, App(a), Biz(m))\}$. g''_1 is ground so we have a leaf node. T_X has a child T_x with the model $\{g'_2 = \phi'_2(HoTpc, AttCnf(x), Res(x)),\ g'_3 = \phi'_3(HoTpc, AttCnf(x), Pub(x, P))\}$. The children are a leaf node for g'_2 and a node $T_P = (P, p, \top)$ with child T_p and model $\{g''_3 = \phi''_3(HoTpc, AttCnf(x), Pub(x, p))\}$. g'_2 includes randvar $Res(x)$ 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. As M appears in the cluster of T_M, it is count-convertible.

3.4 FO Jtrees

LJT runs on FO jtrees using logvars to encode symmetries in FO dtree clusters. We define a parameterised cluster (parcluster), i.e., a set of PRVs.

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

Next, we define FO jtrees, analogous to propositional jtrees, with parclusters replacing clusters and parfactors replacing factors.

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 $\mathcal{J}$. Parameterised 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 minimise, we merge neighbouring nodes if one parcluster is a subset of the other.

3.5 Lifted Junction Tree Algorithm

LJT provides an efficient way for answering a set of queries $\{Q_i\}_{i=1}^m$ given a model G. The main workflow is: (i) Construct an FO jtree for G. (ii) Pass messages. (iii) Compute answers for $\{Q_i\}_{i=1}^m$. For details regarding evidence, see [4].

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 [17]. LJT uses LVE to calculate the content of a message based on separators. If a node has received messages from all neighbours but one, it sends a message to the remaining neighbour (*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 provides an outline of LJT.

Algorithm 1 Lifted Junction Tree Algorithm

```
function FOJT(Model $G$, Queries $\{\mathbf{Q}_i\}_{i=1}^m$)
    FO jtree $\mathcal{J}$ = FO-JTREE($G$)
    PASSMESSAGES($\mathcal{J}$)
    GETANSWERS($\mathcal{J}$,$\{\mathbf{Q}_i\}_{i=1}^m$)
```

4.1 Construction

LJT constructs an FO jtree using FO dtree clusters. The FO dtree is constructed using a naive algorithm proposed in [24] that splits a model based on logvars if no DPG is possible. [24] also provides how to calculate clusters. We formalise how we convert the clusters into parclusters and when and how merging proceeds.

A cluster $\mathbf{A}_i$ of an FO dtree node i forms a parcluster $\forall \mathbf{L} : \mathbf{A}|C$ with

- $\mathbf{A} = \mathbf{A}_i$,
- $\mathbf{L} = lv(\mathbf{A}_i)$, if i is a DPG node $(\mathbf{X}, \mathbf{x}, C_{\mathbf{X}})$, then $\mathbf{L} = lv(\mathbf{A}_T) \cup \mathbf{X}$,
- $C = \emptyset$, if i is a DPG node $(\mathbf{X}, \mathbf{x}, C_{\mathbf{X}})$, then $C = C_{\mathbf{X}}$, and
- $G_i = \emptyset$, if i is a leaf node with factor g, then $G_i = \{g\}$.

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 componentwise. Other relations and operations are defined analogously.

Parclusters $\mathbf{C}_i$ and $\mathbf{C}_j$ with local models G_i and G_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 local model G_k are given by $\mathbf{C}_k = \mathbf{C}_i \cup \mathbf{C}_j$ and $G_k = G_i \cup G_j$. The new node k takes over all neighbours of i and j. If we merge two parcluster, one with logvars $\mathbf{X}$ and one with representative objects $\mathbf{x}$, we first perform the inverse of 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 7. After converting the clusters in Fig. 2 into parclusters, we look at the leaf node with local model $\{g_3''\}$. As the neighbouring 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 neighbour 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 $Res(x)$ (but applies $x \mapsto X$). The same procedure iteratively merges the nodes containing PRVs $HoTpc$, $App(A)$, and $Biz(M)$. Fig. 3 shows the final result with three parclusters,

- $\mathbf{C}_1 = \forall A, M : \{HoTpc, App(A), Biz(M)\}|\top$,
- $\mathbf{C}_2 = \forall X : \{HoTpc, AttCnf(X), Res(X)\}|\top$, and
- $\mathbf{C}_3 = \forall X, P : \{HoTpc, AttCnf(X), Pub(X, P)\}|\top$.

$\mathbf{S}_{12} = \mathbf{S}_{21} = \{HoTpc\}$ and $\mathbf{S}_{23} = \mathbf{S}_{32} = \{HoTpc, AttCnf(X)\}$ are the separators. Each local model consists of one parfactor, which is not a common scenario.

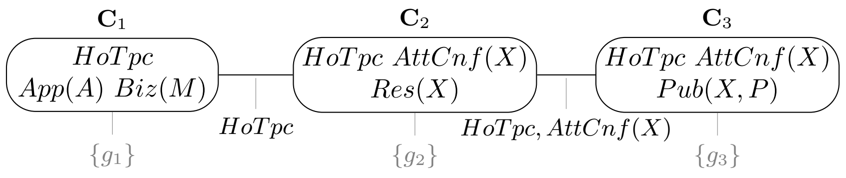

Fig. 3: FO jtree for G_{ex} (parcluster models in gray)

Regarding the extensions, *conjunctive queries* do not have any influence on construction. *Counting* affects construction w.r.t. PCRVs. We allow PCRVs in the input model, increasing its expressivity. PCRVs facilitate specifying counting behaviour explicitly in the model description. Construction handles PCRVs along with PRVs, becoming part of parclusters and possibly separators. Though we can identify logvars for count conversion in the FO dtree, we do not use this feature as explained in the next subsection.

Given an FO jtree for an input model, the next step in LJT is message passing, which we discuss next.

4.2 Message Passing

Message passing starts at the periphery, moves inwards, and then in the opposite direction to distribute all local information through the whole FO jtree. We define a message, discuss the effects of the extensions on messages, and as a consequence of counting, adapt the LVE heuristic selecting the next operation for calculating a message.

For a message from node i to node j, LJT encodes information present at i in 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 G_i to j is a set of parfactors, each with a subset of $\mathbf{S}_{ij}$ as arguments. To calculate m_{ij}, LJT eliminates all PRVs not in $\mathbf{S}_{ij}$ from G_i and the messages from all other neighbours using LVE, as described by

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

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

Example 8. In the FO jtree for G_{ex} as depicted in Fig. 3, messages flow from nodes 1 and 3 to node 2 and back. Messages between nodes 1 and 2 have the argument $HoTpc$, messages between nodes 2 and 3 the arguments $HoTpc$ and $AttCnf(X)$. Inbound, the messages are m_{12} and m_{32}. For m_{12}, LJT eliminates $\mathbf{E}_1 = \{App(A), Biz(M)\}$ from $G' = G_1$ as in Example 5. For m_{32}, LJT eliminates $\mathbf{E}_3 = \{Pub(X,P)\}$ from $G' = G_3$ 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}, LJT sums out $\mathbf{E}_2 = \{AttCnf(X), Res(X)\}$ from $G' = F_2 \cup \{m_{32}\}$ and for m_{23}, $\mathbf{E}_2 = \{Res(X)\}$ from $G' = F_2 \cup \{m_{12}\}$.

The extensions again only influence LJT on behalf of counting since *conjunctive queries* do not affect message passing which is independent of any queries. As mentioned above, *counting* appears in the form of PCRVs in an input model. As part of a model, PCRVs appear during message passing in a separator or in the set of PRVs to eliminate and are handled accordingly.

Algorithm 2 Conjunctive Query Answering

```
function getAnswers(FO Jtree $J$, Queries $\{\mathbf{Q}_i\}_{i=1}^m$)
    for $\mathbf{Q} \in \{Q_i\}_{i=1}^m$ do
        Subtree $J^{\mathbf{Q}} \leftarrow$ getSubtree($J$, $\mathbf{Q}$)
        Model $G^{\mathbf{Q}} \leftarrow$ getModel($J^{\mathbf{Q}}$)
        LVE($G^{\mathbf{Q}}$,$\mathbf{Q}$)
```

We do not count-convert logvars identified for count 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 count conversion of logvar M superfluous. Since we cannot always determine from the clusters in the FO dtree if count conversion is reasonable for message passing, we do not count-convert in the FO dtree.

Example 8 references another use of counting, namely, as a means to enable a sum-out operation after count conversion when calculating a message. Without counting, the algorithm would need to ground a logvar. After count conversion, the new PCRV becomes part of the model that is used for further calculating the message. Then, the scenario plays out as described above when PCRVs are part of the model itself. The new PCRV either needs to be eliminated or becomes part of the message if the original PRV is part of the separator. If the new PCRV is part of the message, it becomes part of message calculations at the receiver.

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 M has 10. LVE would count-convert M as it leads to a smaller parfactor than count-converting A. After the count conversion, it still cannot sum out $\#_M[Biz(M)]$. So, it count-converts A to finally sum out $\#_M[Biz(M)]$, making the first count conversion unnecessary. For LJT, we require the heuristic to consider the PRVs 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.

After receiving messages from each neighbour, the parclusters hold in their local models all information to answer queries on its PRVs.

4.3 Query Answering

While *conjunctive queries* so far do not change LJT, query answering changes as we allow for multiple grounded PRVs $\mathbf{Q}$ in a query. Since we do not discuss evidence, a query has the form $P(\mathbf{Q})$ with a set of grounded PRVs $\mathbf{Q}$. LJT has as input a set of queries that now can each be a set of grounded PRVs $\mathbf{Q}_i$ instead of a single grounded PRV Q_i.

Query answering so far has meant to find a parcluster that contains the query randvar Q_i and eliminate all non-query PRVs from its local model using LVE. With a set of grounded PRVs in a query, we may 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 [16], which extracts 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 $\mathbf{Q}$. From the parclusters in the subtree, we extract a model to answer $\mathbf{Q}$ with LVE handling multiple query randvars. A more detailed description of each step follows, starting with identifying a subtree.

Subtree Identification The goal is to find a subtree of the FO jtree where the subtree parclusters cover all query randvars $\mathbf{Q}$. Since the subtree is the basis for model extraction, the subtree should result in the smallest model possible in terms of number of PRVs. In a straight forward way, LJT finds a first node that covers at least part of $\mathbf{Q}$ and uses it as the first node in the subtree $J^{\mathbf{Q}}$. Then, it adds further nodes that cover still missing query randvars closest to the current $J^{\mathbf{Q}}$. Future work includes ways of finding a reasonably small subtree efficiently.

From the subtree covering all query randvars, LJT needs to extract a model to actually answer the query.

Model Extraction We build a model $G^{\mathbf{Q}}$ from subtree $J^{\mathbf{Q}}$. The extracted model may not contain any duplicate information, meaning we cannot simply use all local models and messages within $J^{\mathbf{Q}}$. Instead, we use the local models at the nodes in $J^{\mathbf{Q}}$ and the messages that the nodes at the borders of $J^{\mathbf{Q}}$ received from outside $J^{\mathbf{Q}}$. 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.

The remaining step in answering a query is to let LVE answer the query using the extracted model.

Query Answering Using the model $G^{\mathbf{Q}}$ built during model extraction, LJT performs LVE to answer a query over the randvars $\mathbf{Q}^{\mathbf{Q}}$. Though LVE as described by [26] 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 $\mathbf{Q}$, we shatter $G^{\mathbf{Q}}$ on $\mathbf{Q}$. We use LVE to compute a joint probability for $\mathbf{Q}$ and normalise. LJT still works for a singleton query of one randvar Q as we find a node k that covers Q, extract a model, namely the local model G_k and all messages to k, and perform LVE.

Example 9. After passing messages, LJT can answer, e.g., the queries $P(Res(eve), Pub(eve, p_1))$ and $P(AttCnf(eve))$. For $\mathbf{Q}_1 = \{Res(eve), Pub(eve, p_1)\}$, nodes 2 and 3 cover the query randvars. The extracted model $G^{\mathbf{Q}_1}$ consists of G_2, G_3, and m_{12}. Shattering $G^{\mathbf{Q}_1}$ w.r.t. $\mathbf{Q}_1$ leads to five parfactors. LJT sums 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 $HoTpc$ and $AttCnf(X)$, $X \neq eve$. Next, it sums out $Res(X)$, $X \neq eve$, from the g_2 duplicate without eve, resulting in a parfactor g'' with arguments $HoTpc$ and $AttCnf(X)$, $X \neq eve$. Summing out $AttCnf(X)$, $X \neq eve$, from the product of g' and g'' yields a parfactor g''' with argument $HoTpc$. Summing out $AttCnf(eve)$ from the product of g_2 and g_3 where $X = eve$ and $P = p_1$ yields a parfactor $\hat{g}$ with arguments $HoTpc$, $Res(eve)$, and $Pub(eve, p_1)$. Last, LJT multiplies m_{12}, g''', and $\hat{g}$, sums out $HoTpc$, and normalises, leading to the queried distribution.
For $\mathbf{Q}_2 = \{AttCnf(eve)\}$, LJT can use node 2. It sums out $Res(X)$, $HoTpc$, and $AttCnf(X)$ where $X \neq eve$ from $G_2 \cup \{m_{12}, m_{32}\}$ after shattering.

Regarding the extensions to LJT, query answering changes substantially with *conjunctive queries* since the models for answering a query first need to be compiled as just described. *Counting* affects query answering in the sense that extracted models may contain PCRVs. LVE for query answering in LJT uses count conversion as well and can sum out PCRVs. Prior to a short empirical evaluation, we look at the extended LJT from a more theoretical viewpoint.

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 original LJT and PCRVs and their handling in LVE and FO dtrees are sound. We first look at LJT with counting and then at LJT for conjunctive queries.

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

Proof sketch. To compute answers to queries at a node with information present through a PP scheme, Shenoy and Shafer present three axioms for the operations marginalisation and combination on potential functions in a jtree [22]. Our definition of potential functions and PP scheme coincide with [22] with lifted summing out and lifted multiplication in the roles of marginalisation and combination fulfilling the axioms for local computations. The original LJT constructs

a valid jtree in the form of an FO jtree with nodes containing randvars and jtree properties fulfilled.
The extended LJT still constructs a valid FO jtree as FO jtree construction is unchanged. PCRVs are handled during FO dtree construction and appear in parclusters accordingly. Since we assume a sound LVE with sound handling of count conversions and PCRVs, lifted summing out and lifted multiplication remain sound. Thus, message passing is still allowed and produces sound results: With sound LVE operations and a valid FO jtree, the algorithm carries out sound computations at the local models, sending sound information from one node to another. The same holds for query answering: With sound information at the nodes, LJT computes a correct answer for a query.

Next, we look at LJT with counting and conjunctive queries.

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

Proof sketch. Given that LJT with counting 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 as argued above. Given that LVE is sound, LJT computes a correct answer for a query on the extracted submodel.

Next, we look at best and worst case scenarios for LJT including what characteristics influence LJT runtimes.

Best and Worst Case Scenario The extended LJT allows for efficient query answering given multiple queries. It imposes some static overhead due to FO jtree construction and message passing. After these steps, it answers queries based on typically smaller models compared to the input model G. If G changes, LJT ha to construct a new FO jtree.

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.

In a worst case scenario (the same holds for LVE), LJT needs to ground all logvars in the model and perform inference at a propositional level to calculate correct results. In such a case, it 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. Simplified, 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 PRVs in a parcluster automatically results in at least a counting conversion and, in the worst case, groundings. Since 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. For a detailed discussion, see [4].

For singleton queries, we gain the most if the model permits an FO jtree with few PRVs per parcluster. With a clever access function, LJT quickly identifies a parcluster for the query and sum out the few non-query PRVs. For conjunctive queries, the best case is if the nodes that cover all query PRVs are adjacent and form a submodel with few PRVs to eliminate. Needing the whole tree represents the worst case as LJT builds a submodel equal to the original model and do standard LVE, adding overhead without payoff. Over many queries, LJT offsets queries requiring a large model with queries using a small model.

After these theoretical considerations, we look at an empirical evaluation for our running example G_{ex}.

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 use the `gcfove` operators in `exfojt`. We also implemented a propositional junction tree algorithm, named `jt`, as a reference point.

Standard lifting examples such as the smokers model are too simple, leading to an FO jtree with one node. Runtimes of `exfojt` on the standard examples compared to `gcfove` are slightly higher due to the static overhead for constructing an FO jtree (FO dtree construction, cluster calculation, parcluster conversion, merging). The resulting node carries the original model as a local model. Message passing does not apply with one node. Query answering takes the same time for each query as both carry out the same operations on the original model.

We use G_{ex} as input. We vary the domain sizes, yielding grounded model sizes $|gr(G_{ex})|$ between 3 and 241,000. We query each PRV once with one grounding, resulting in 6 queries,

– $HoTpc$,
– $Biz(m_1)$,
– $App(a_1)$,
– $Res(x_1)$,
– $AttCnf(x_1)$, and
– $Pub(x_1, p_1)$.

Which grounding we use is irrelevant for the calculations and the answer for the algorithms given a current model size, since the instances are interchangeable.

We compare runtimes for inference accumulated over the given queries, averaged over several runs. `exfojt` constructs an FO jtree comparable to the FO jtree in Fig. 3 with three nodes and passes messages (four messages). 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. The propositional

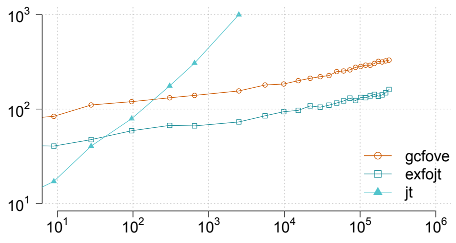

Fig. 4: Runtimes [ms] with $|gr(G_{ex})|$ ranging from 3 to 241,000 on log scales accumulated over 6 queries

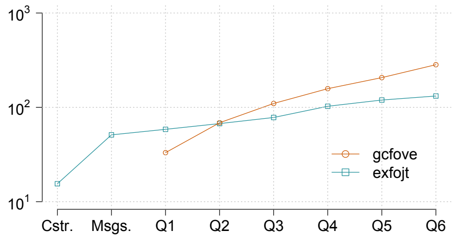

Fig. 5: Runtimes [ms] on log scale with $|gr(G_{ex})| = 102{,}050$ accumulating over 6 queries

jtrees have an increasing number of nodes with the largest cluster containing four randvars. While message passing takes longer in jtrees, QA is faster than QA in FO jtrees since only up to three randvars need to be eliminated without any accounting for logvars necessary. `gcfove` eliminates all non-query randvars from G_{ex} for each query. We do not compare against the original LJT version since our example model leads to groundings without counting. Its runtimes come close to the runtimes of `jt`.

Figure 4 shows runtimes for inference in miliseconds 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 of $|gr(G_{ex})|$, 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 its FO jtree and messages at each node.

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 $|gr(G_{ex})| = 102{,}050$. The shape of the curves is identical over the different groundings with higher or lower runtimes. We ordered the given queries by increasing runtimes for `gcfove`. With the second query, `gcfove` needs marginally more time. With each passing query, `exfojt` saves more time compared to `gcfove` as it is able to answer queries based on one node.

Conjunctive queries that `exfojt` answers using one parcluster have similar runtimes compared to the singleton queries from above. With more complex queries that require more than one parcluster, runtimes increase since subtree identification takes longer and the models become larger. We do not compare runtimes for conjunctive queries as `gcfove` only supports singleton queries.

In summary, even in our small example model and only a prototype implementation, spending effort on an FO jtree 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 other 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 allows for a lifted run, i.e., without groundings, 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 parallelisation, 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

1. Ahmadi, B., Kersting, K., Mladenov, M., Natarajan, S.: Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training. Machine Learning 92(1), 91–132 (2013)
2. Bellodi, E., Lamma, E., Riguzzi, F., Costa, V.S., Zese, R.: Lifted Variable Elimination for Probabilistic Logic Programming. Theory and Practice of Logic Programming 14(4–5), 681–695 (2014)
3. Braun, T., Möller, R.: Lifted Junction Tree Algorithm. In: Proceedings of KI 2016: Advances in Artificial Intelligence. pp. 30–42 (2016)
4. Braun, T., Möller, R.: Preventing Groundings and Handling Evidence in the Lifted Junction Tree Algorithm. In: Proceedings of KI 2017: Advances in Artificial Intelligence. pp. 85–98 (2017)
5. van den Broeck, G.: Lifted Inference and Learning in Statistical Relational Models. Ph.D. thesis, KU Leuven (2013)
6. van den Broeck, G., Niepert, M.: Lifted Probabilistic Inference for Asymmetric Graphical Models. In: AAAI-15 Proceedings of the 29th Conference on Artificial Intelligence. pp. 3599–3605 (2015)
7. Chavira, M., Darwiche, A.: Compiling Bayesian Networks Using Variable Elimination. In: IJCAI-07 Proceedings of the 20th International Joint Conference on Artificial Intelligence. pp. 2443–2449 (2007)
8. Choi, J., Amir, E., Hill, D.J.: Lifted Inference for Relational Continuous Models. In: UAI-10 Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence. pp. 13–18 (2010)
9. Darwiche, A.: Recursive Conditioning. Artificial Intelligence 2(1–2), 4–51 (2001)
10. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press (2009)
11. Das, M., Wu, Y., Khot, T., Kersting, K., Natarajan, S.: Scaling Lifted Probabilistic Inference and Learning Via Graph Databases. In: Proceedings of the SIAM International Conference on Data Mining. pp. 738–746 (2016)
12. Gogate, V., Domingos, P.: 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. pp. 19–25 (2010)

13. Gogate, V., Domingos, P.: Probabilistic Theorem Proving. In: UAI-11 Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence. pp. 256–265 (2011)
14. Jensen, F.V., Lauritzen, S.L., Olesen, K.G.: Bayesian Updating in Recursive Graphical Models by Local Computations. Computational Statistics Quarterly 4, 269–282 (1990)
15. Kersting, K., Ahmadi, B., Natarajan, S.: Counting Belief Propagation. In: UAI-09 Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence. pp. 277–284 (2009)
16. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. The MIT Press (2009)
17. Lauritzen, S.L., Spiegelhalter, D.J.: Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. Journal of the Royal Statistical Society. Series B: Methodological 50, 157–224 (1988)
18. Milch, B., Zettelmeyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted Probabilistic Inference with Counting Formulas. In: AAAI-08 Proceedings of the 23rd Conference on Artificial Intelligence. pp. 1062–1068 (2008)
19. Poole, D., Zhang, N.L.: Exploiting Contextual Independence in Probabilistic Inference. Jounal of Artificial Intelligence 18, 263–313 (2003)
20. de Salvo Braz, R., Amir, E., Roth, D.: Lifted First-order Probabilistic Inference. In: IJCAI-05 Proceedings of the 19th International Joint Conference on Artificial Intelligence (2005)
21. Shafer, G.R., Shenoy, P.P.: Probability Propagation. Annals of Mathematics and Artificial Intelligence 2(1), 327–351 (1990)
22. Shenoy, P.P., Shafer, G.R.: Axioms for Probability and Belief-Function Propagation. Uncertainty in Artificial Intelligence 4 9, 169–198 (1990)
23. Singla, P., Domingos, P.: Lifted First-order Belief Propagation. In: AAAI-08 Proceedings of the 23rd Conference on Artificial Intelligence. pp. 1094–1099 (2008)
24. Taghipour, N.: Lifted Probabilistic Inference by Variable Elimination. Ph.D. thesis, KU Leuven (2013)
25. Taghipour, N., Davis, J., Blockeel, H.: First-order Decomposition Trees. In: Advances in Neural Information Processing Systems 26, pp. 1052–1060. Curran Associates, Inc. (2013),
26. Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. Journal of Artificial Intelligence Research 47(1), 393–439 (2013)
27. Vlasselaer, J., Meert, W., van den Broeck, G., Raedt, L.D.: Exploiting Local and Repeated Structure in Dynamic Baysian Networks. Artificial Intelligence 232, 43–53 (2016)
28. Zhang, N.L., Poole, D.: A Simple Approach to Bayesian Network Computations. In: Proceedings of the 10th Canadian Conference on Artificial Intelligence. pp. 171–178 (1994)