Preventing Groundings and Handling Evidence 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. For inference in probabilistic formalisms with first-order constructs, lifted variable elimination (LVE) is one of the standard approaches for single queries. To handle multiple queries efficiently, the lifted junction tree algorithm (LJT) uses a specific representation of a first-order knowledge base and LVE in its computations. Unfortunately, LJT induces unnecessary groundings in cases where the standard LVE algorithm, GC-FOVE, has a fully lifted run. Additionally, LJT does not handle evidence explicitly. We extend LJT (i) to identify and prevent unnecessary *groundings* and (ii) to effectively handle *evidence* in a lifted manner. Given multiple queries, e.g., in machine learning applications, our extension computes answers faster than LJT and GC-FOVE.

Keywords: Probabilistic Logical Models, Reasoning, Lifting

1 Introduction

AI research and application areas such as machine learning (ML) need efficient inference algorithms. Modeling realistic scenarios results in large probabilistic knowledge bases (KBs) that require reasoning about sets of individuals. Lifting uses symmetries in a KB, also called model, to speed up reasoning with known objects. We study the problem of reasoning in large KBs with symmetries for answering multiple queries, a common scenario in ML. Answering queries reduces to computing marginal distributions. We aim to enhance the efficiency of these computations exploiting that a model remains constant under multiple queries.

In [3], we introduce a lifted junction tree algorithm (LJT) for multiple queries on models with first-order constructs. The algorithm combines the junction tree algorithm [12,18] and lifted variable elimination (LVE) [18]. LJT currently does not provide a lifted run for all models that have a lifted solution in LVE, requiring unnecessary groundings. This paper contributes the following: First, we identify when LJT induces unnecessary *groundings*. Second, we add a *fusion* step that aims at preventing these groundings for models with a lifted solution in LVE. Third, we add efficient *evidence* handling.

LJT imposes some static overhead for building a first-order junction tree (FO jtree) and for propagating knowledge in this tree. The fusion step slightly adds to the static overhead in exchange for faster knowledge propagation. Evidence handling does not affect FO jtree construction but inherently affects knowledge distribution and query answering. We significantly speed up runtime compared to LJT and LVE. Overall, we handle multiple queries more efficiently than existing approaches tailored for handling single queries.

The remainder of this paper is structured as follows: First, we look at related work on exact inference and lifting. Next, we introduce basic notations and recap LJT. Then, we present conditions for groundings and our extensions regarding fusion and evidence. A short evaluation shows the potential of our approach. Last, we give a conclusion and provide future work.

2 Related Work

For single queries given some evidence, researchers have sped up runtimes for inference significantly over the last two decades. For propositional formalisms, VE decomposes a model into subproblems to evaluate them in an efficient order [21]. We can represent such a decomposition using a dtree [6]. LVE, also called first-order VE (FOVE), first introduced in [14] and expanded in [15,13], exploits symmetries at a global level. LVE saves computations by reusing intermediate results for isomorphic subproblems. Its current standard form GC-FOVE generalizes counting and decouples lifting from constraint handling [18].

For multiple queries in a propositional setting, Lauritzen and Spiegelhalter [12] present jtrees along with a reasoning algorithm that uses a message passing scheme, known as probability propagation (PP). Well known PP schemes include [16,11] trading off runtime and storage differently. The connection between jtrees and VE lies in a dtree representing a VE: The clusters of a dtree form a jtree [7]. Taghipour *et al.* [19] introduce first-order dtrees (FO dtrees) and perform a theoretical analysis of lifted inference using the clusters of an FO dtree.

Many researchers apply lifting to various settings, e.g., continuous or dynamic KBs [5,20], logic programming [2], or theorem proving [10]. For example, van den Broeck [4] lifts weighted model counting and knowledge compilation. Lifted belief propagation combines PP and lifting, often using lifted representations [17,9,1], allowing for approximate inference. Das *et al.* [8] use graph data bases storing compiled models for scalability. To the best of our knowledge, none of them focus on multiple queries or changing evidence.

In [3], we lift jtrees introducing FO jtrees. Our reasoning algorithm induces additional groundings and does not handle evidence effectively. We widen its scope with our extensions regarding fusion and evidence.

3 Preliminaries

This section introduces basic notations, provides an overview of LVE and recaps LJT along with FO jtrees based on [18,3].

3.1 Parameterized Models

Parameterized models compactly represent models with first-order constructs. We first denote basic building blocks for constructing more complex structures.

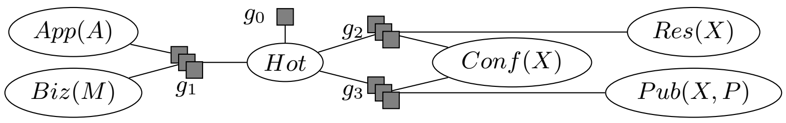

Fig. 1: Parfactor graph for G_{ex}

Definition 1. *Let* $\mathbf{L}$ *be a set of logical variable names (logvars),* Φ *a set of factor names, and* $\mathbf{R}$ *a set of random variable names (randvars). A* parameterized 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. Domain* $\mathcal{D}(L)$ *refers to the values a logvar* L *can take and* $range(A)$ *to the values a PRV* A *can take. A* constraint $(\mathbf{X}, C_{\mathbf{X}})$ *is a tuple of a sequence of logvars* $\mathbf{X} = (X_1, \dots, X_n)$ *and a set* $C_{\mathbf{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X_i)$ *restricting logvars to certain values. Symbol* $\top$ *marks that no restrictions apply and may be omitted.*

Parametric factor (parfactor) g *has a function mapping inputs to real values. We specify* g *with* $\forall \mathbf{X} : \phi(\mathcal{A}) \mid C$*.* $\mathbf{X}$ *is a set of logvars that* g *generalizes over.* $\mathcal{A} = (A_1, \dots, A_n)$ *is a sequence of PRVs, each PRV built from* $\mathbf{R}$ *and possibly* $\mathbf{L}$*. We omit* $(\forall \mathbf{X} :)$ *if* $\mathbf{X} = logvars(\mathcal{A})$*.* C *is a constraint on* $\mathbf{X}$*.* $\phi : \times_{i=1}^{n} range(A_i) \mapsto \mathbb{R}^+$ *is a potential function with name* $\phi \in \Phi$*.* ϕ *is identical for all randvars represented by the logvars in* $\mathcal{A}$ *w.r.t.* C*. A full specification of* ϕ *includes values for each combination of input values. A set of parfactors forms a* model $G := \{g_i\}_{i=1}^{n}$ *representing the probability distribution* $P_G = \frac{1}{Z} \prod_{f \in gr(G)} \phi_f(\mathcal{A}_f)$*. Term* $gr(G)$ *denotes a set of instances with all logvars in* G *grounded.*

The terms $logvars(P)$ and $randvars(P)$ denote the logvars and randvars in input P, e.g., a parfactor or model. We specify model G_{ex} for publications on some topic. We model that the topic may be hot, serves business markets and application areas, people do research, attend conferences, and publish in publications.

Example 1. Let $\mathbf{L} = \{A, M, P, X\}$, $\Phi = \{\phi_0, \phi_1, \phi_2, \phi_3\}$, and $\mathbf{R} = \{Hot, Biz, App, Res, Conf, Pub\}$. We build five binary PRVs with $n > 0$ and one with $n = 0$: Hot, $Biz(M)$, $App(A)$, $Conf(X)$, $Res(X)$, $Pub(X,P)$. The model reads $G_{ex} = \{g_0, g_1, g_2, g_3\}$ where $g_0 = \phi_0(Hot)$, $g_1 = \phi_1(Hot, App(A), Biz(M))|C_1$, $g_2 = \phi_2(Hot, Conf(X), Res(X))|C_2$, and $g_3 = \phi_3(Hot, Conf(X), Pub(X,P))|C_3$. We omit concrete functions for ϕ_0 to ϕ_3. We exemplarily define constraint $C_3 = ((P,X), C_{(P,X)})$. Let $\mathcal{D}(P) = \{p_1, p_2\}$ and $\mathcal{D}(X) = \{alice, eve, bob\}$. We define $C_{(W,X)}$ as $\{(p_1, eve), (p_1, bob), (p_2, alice), (p_2, eve)\}$. ϕ_3 applies to all tuples in C_3. Figure 1 depicts G_{ex} as a graph with six variable nodes for the PRVs and four factor nodes for g_0 to g_3 with edges to the PRVs involved.

The semantics of a model is given by grounding w.r.t. constraints and building a full joint distribution. Query answering (QA) asks for a probability distribution of a randvar w.r.t. a model's joint distribution and fixed events (evidence). A grounded PRV Q and a set of events $\mathbf{E}$ (grounded PRVs with values) build a query $P(Q|\mathbf{E})$. For G_{ex}, $P(pub(eve, p_1)|conf(eve))$ forms a query. Next, we look at QA algorithms seeking to avoid grounding and building a joint distribution.

3.2 Lifted Variable Elimination

LVE employs two main techniques for QA, (i) decomposition into isomorphic subproblems and (ii) counting of domain values leading to a certain range value. The first one refers to lifted summing out. The idea is to compute VE for one case and exponentiate the result for isomorphic instances. The second one exploits that all randvars of a PRV A evaluate to $range(A)$, forming a histogram by counting for each $v \in range(A)$ how many instances of $gr(A)$ evaluate to v.

Definition 2. *$\#_{X \in C}[P(\mathbf{X})]$ denotes a counting randvar* (CRV) *with PRV $P(\mathbf{X})$ and constraint C, where $logvars(\mathbf{X}) = \{X\}$. Its range is the space of possible histograms. If $\{X\} \subset logvars(\mathbf{X})$, the CRV is a* parameterized CRV (PCRV) *representing a set of CRVs. Since counting binds logvar X, $logvars(\#_{X \in C}[P(\mathbf{X})]) = \mathbf{X} \setminus \{X\}$. We count-convert a logvar X in a parfactor $g = \mathbf{L} : \phi(\mathcal{A})|C$ by turning a PRV $A_i \in \mathcal{A}, X \in logvars(A_i)$, into a CRV A'_i. In the new parfactor g', the input for A'_i is a histogram h. Let $h(a_i)$ denote the count of a_i in h. Then, $\phi'(\dots, a_{i-1}, h, a_{i+1}, \dots)$ maps to $\prod_{a_i \in range(A_i)} \phi(\dots, a_{i-1}, a_i, a_{i+1}, \dots)^{h(a_i)}$.*

For both techniques, preconditions exist, see [18]. E.g., to sum out PRV A from parfactor g, $logvars(A) = logvars(g)$. To count-convert logvar X in g, only one input in g contains X. Let us apply LVE to parfactor $g_1 \in G_{ex}$.

Example 2. In $g_1 = \phi_1(Hot, App(A), Biz(M))|C_1$, we cannot sum out any PRV as neither includes both logvars. To sum out $App(a)$ of some a in the propositional case, we would multiply all factors that include $App(a)$ into a factor with inputs Hot, $App(a)$, and $Biz(m_1), \dots Biz(m_n)$ for each market in C_1. All $Biz(m_i)$ lead to true or false, making $Biz(M)$ a CRV. We 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 $v \in range(Biz(M))$ how many grounded PRVs evaluate to v. The mappings $(h, a, true) \mapsto x$ and $(h, a, false) \mapsto y$ in ϕ become $(h, a, [n_1, n_2]) \mapsto x^{n_1} y^{n_2}$ in ϕ'. We can now sum out $App(A)$.

Evidence shows symmetries as well, exhibiting the same value for n ground randvars of a PRV. Evidence parfactor $\phi_E(P(\mathbf{X}))|C_E$ holds evidence for PRV $P(\mathbf{X})$. Potential function ϕ_E and constraint C_E encode the observed values and randvars. For each evidence parfactor g_E, LVE tests each parfactor $g \in G$ if $C_E \cap C \neq \emptyset$. If true, it splits g for lifted absorption: We add a duplicate g' and restrict C to tuples where a component receives evidence through g_E and C' to tuples unaffected by evidence. Then, g absorbs g_E, eliminating P in g.

Example 3. We observe that $Conf(x_1), \dots, Conf(x_{10})$ are true. In evidence parfactor $g_E = \phi_E(Conf(X))|C_E$, C_E restricts X to $x_1, \dots, x_{10}$. $\phi_E(true) = 1$ and $\phi_E(false) = 0$. g_E affects g_2 and g_3. After splitting, g_2 and g_3 absorb g_E.

3.3 Lifted Junction Tree Algorithm

LJT builds an FO jtree for faster QA. We first define a parameterized cluster (parcluster) and FO jtrees, analogous to propositional jtrees, before diving into the algorithm. Compared to [3], we assign a set of parfactors F, i.e., a local model, to a parcluster instead of one parfactor due to evidence splitting.

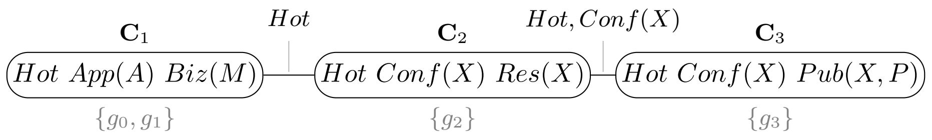

Fig. 2: FO jtree for G_{ex} (local parcluster models in gray)

Definition 3. *A parcluster* $\mathbf{C}$ *is denoted by* $\mathbf{C} := \forall \mathbf{L} : \mathcal{A} \mid C$ *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* $\mathbf{L}$*. Each parcluster has a possibly empty set of parfactors* F *assigned. A parfactor* $g_{\mathbf{C}} = \phi(\mathcal{A}_\phi)|C_\phi$ *assigned to* $\mathbf{C}$ *fulfills (i)* $\mathcal{A}_\phi \subseteq \mathcal{A}$*, (ii)* $logvars(\mathcal{A}_\phi) \subseteq \mathbf{L}$*, and (iii)* $C_\phi \subseteq C$*.*

Definition 4. *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* $\mathbf{C}_j$*.*

By way of construction, LJT assigns each parfactor in G to exactly one parcluster in $(\mathcal{J}, f_{\mathbf{C}})$, by adding them to local models F_i at nodes i.

Example 4. For G_{ex}, Fig. 2 shows its FO jtree with three parclusters, $\mathbf{C}_1 = \forall A, M : \{Hot, App(A), Biz(M)\}|\top$, $\mathbf{C}_2 = \forall X : \{Hot, Conf(X), Res(X)\}|\top$, and $\mathbf{C}_3 = \forall X, P : \{Hot, Conf(X), Pub(X,P)\}|\top$. Separators are $\mathbf{S}_{12} = \mathbf{S}_{21} = \{Hot\}$ and $\mathbf{S}_{23} = \mathbf{S}_{32} = \{Hot, Conf(X)\}$. Each parcluster has one or two parfactors in its local model (g_0 could have been assigned to any of them).

LJT answers 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 $\mathbf{Q}$. For construction, see [3]. Message passing spreads information among nodes. Two passes propagating information from peripheral to inner nodes and back suffice [12]. A *message* m_{ij} from node i to node j is a parfactor with the PRVs in $\mathbf{S}_{ij}$ as inputs. To compute m_{ij}, we sum out $\mathcal{A}_i \setminus \mathbf{S}_{ij}$ from F_i and the messages from all other neighbors. 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. To answer a query, we take a parcluster covering the query terms and sum out all non-query terms in its model and received messages.

Example 5. In the FO jtree in Fig. 2, messages flow from nodes 1 and 3 to node 2 and back with the corresponding separators as inputs. E.g., messages between nodes 1 and 2 have the argument Hot. For m_{12}, we sum out $App(A)$ and $Biz(M)$ from F_1. For m_{21}, we sum out $Conf(X)$ and $Res(X)$ from F_2 and message m_{32} from node 3. After message passing, we can answer, e.g., query $P(Conf(x_1))$ at node 3 by summing out Hot and $Pub(X,P)$ from $F_3 \cup \{m_{23}\}$.

4 Algorithm-induced Groundings

A lifted solution to a query given a model means that we compute an answer without grounding a part of the model. Not all models have a lifted solution as LVE requires certain conditions to hold to be applicable. Computing a solution to queries based on these models involves groundings with any exact lifted inference algorithm. But additionally to inherent groundings, LJT may induce unnecessary groundings during message passing as the separators may impede a reasonable elimination order. Grounding a logvar is expensive and, during message passing, may propagate through all nodes, forcing even more groundings in a worst case. This section examines when algorithm-induced groundings occur and derives conditions for messages that allow lifted solutions if possible.

Within this section, we use examples displayed in Fig. 3. Each example is a node with two PRVs in its parcluster and an edge with a separator consisting of one of the PRVs. The local model has one parfactor with both PRVs as inputs. We use the labels $\mathbf{L} = \{X, Y, Z\}$ and $\mathbf{R} = \{P, Q, R\}$ to build PRVs.

Informally, LJT does not induce groundings due to message calculations if it can sum out the PRVs in a separator last. Figure 3a shows an example without groundings. The parcluster contains $P(X)$ and $Q(X, Y)$. For the message, we have to eliminate $Q(X, Y)$ from the local parfactor. $Q(X, Y)$ fulfills all preconditions for lifted summing out. We can sum out $P(X)$ last. No groundings occur.

Formally, for message m_{ij} from node i to j with parcluster $\mathbf{C}_i = \mathcal{A}_i|C_i$, local model F_i, and separator $\mathbf{S}_{ij}$, we eliminate the parcluster PRVs not part of the separator, i.e., $\mathbf{E}_{ij} := \mathcal{A}_i \setminus \mathbf{S}_{ij}$, from the local model and all messages received from other nodes than j, i.e., $F' := F_i \cup \{m_{il}\}_{l \neq j}$. To eliminate $E \in \mathbf{E}_{ij}$ by lifted summing out from F', we replace all parfactors $g \in F'$ that include E with a parfactor $g^E = \phi(\mathcal{A}^E)|C^E$ that is the lifted product of these parfactors g. Let $\mathbf{S}^E_{ij} := \mathbf{S}_{ij} \cap \mathcal{A}^E$ be the set of randvars in the separator that occur in g^E. For lifted message calculation, it necessarily has to hold $\forall S \in \mathbf{S}^E_{ij}$,

$$logvars(S) \subseteq logvars(E). \qquad (1)$$

Otherwise, E does not include all logvars in g^E. We may induce Eq. (1) for a particular S by count conversion if S has an additional, count-convertible logvar:

$$logvars(S) \setminus logvars(E) = \{L\}, L \text{ count-convertible in } g^E. \qquad (2)$$

If Eq. (2) holds, we count-convert L, yielding a (P)CRV in m_{ij}, else, we ground.

Figure 3b shows a parcluster with $E = P(X)$ and $S = Q(X, Y, Z)$ where we ground. As $logvars(Q(X, Y, Z)) \nsubseteq logvars(P(X))$ and $Q(X, Y, Z)$ has two logvars not in $P(X)$, Eqs. (1) and (2) do not hold. We can count-convert Y (or Z), still leaving us with Z (or Y) to ground. In Fig. 3c, the separator PRV, $Q(X, Y)$, has one logvar, Y, more than $P(X)$. Y is count-convertible so Eq. (2) holds. We count-convert Y, building $\#_Y[Q(X, Y)]$. Now, X is the only logvar and we sum out $P(X)$. The same holds if P has more logvars, e.g., $P(X, Z)$. Y would not be count-convertible if the parcluster and parfactor contain, for instance, a PRV $R(Y)$ as Y appears in two PRVs, leading to a grounding of Y.

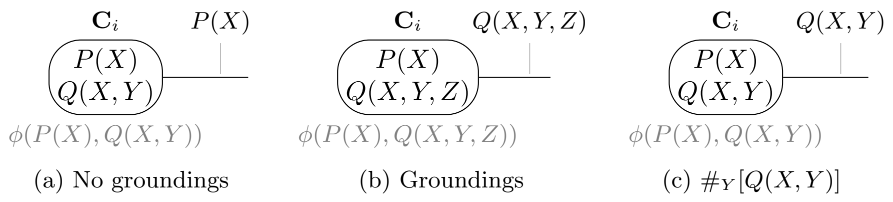

Fig. 3: Conceptual examples with liftable and non-liftable message calculation

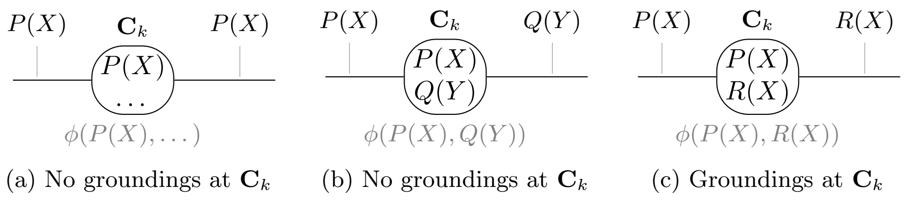

Fig. 4: Conceptual examples with $\#_X[P(X)]$ in incoming message

Count conversion of a logvar L may prevent groundings at node i but the (P)CRV of the affected PRV S may cause problems at node j. As S appears in F_j, we have S present as a PRV and a (P)CRV. If we do not need to sum out S or if L is count-convertible in g^S, the (P)CRV does not lead to groundings. But if we have to eliminate S and L is not count-convertible, we need to ground all occurrences of the counted logvar in the affected parfactors. Hence, count conversion only helps in preventing a grounding if all following messages can handle the resulting (P)CRV. Formally, for each node k receiving S as a (P)CRV with counted logvar L, it has to hold for each neighbor n of k that

$$S \in \mathbf{S}_{kn} \vee L \text{ count-convertible in } g^S \tag{3}$$

Let us look at some examples for clarification. Figure 4 shows example nodes as in Fig. 3 with another edge and $\#_X[P(X)]$ in an incoming message. In Fig. 4a, $\#_X[P(X)]$ does not lead to groundings as $P(X)$ is in the next separator, i.e., the first disjunct in Eq. (3) holds. However, as $\#_X[P(X)]$ becomes part of the next message, we have to check if $\#_X[P(X)]$ causes groundings at the receiving node. Figure 4b shows a case where $\#_X[P(X)]$ is not in the next separator but X is count-convertible, i.e., the second disjunct in Eq. (3) holds. As an aside, to actually sum out $\#_X[P(X)]$, we need to count-convert logvar Y in $Q(Y)$. But that conversion is not due to $\#_X[P(X)]$. In Fig. 4c, neither disjunct holds for $P(X)$. $P(X)$ is not in the separator and X is not count-convertible as X appears in $R(X)$ as well. Because X is not count-convertible, we cannot combine the counted $P(X)$ with the local $P(X)$ for summing out. Instead, we need to ground X and sum out each $P(x)$, $x \in \mathcal{D}(X)$, individually.

In the next section, we derive a test for unnecessary groundings at nodes based on Eqs. (1) to (3) without messages being sent.

5 Extended Lifted Junction Tree Algorithm

We extend LJT to prevent unnecessary groundings by adding a *fusion* step and to fully support evidence in an efficient manner.

5.1 Fusion: Preventing Groundings

The main idea of fusion is to merge nodes if message calculation needs groundings. We first set up a grounding test. Then, we present fusion using the test to decide mergings. Last, we analyze the effects on data structure and workload.

Test for Groundings Checking message m_{ij}, PRV E to eliminate, and separator PRV S, the test strings together Eqs. (1) to (3). Testing E only needs $\mathcal{A}^E$ of g^E, making it easier to build. But, we need to track changes from quasi-eliminating E for the next PRV E'. Our test runs before message passing. Since we do not have the actual messages for F', we assume that a message covers the separator. This slight over-approximation may result in a larger $\mathcal{A}^E$ which may lead to more PRVs in $\mathbf{S}^E_{ij}$ that have to fulfill Eqs. (1) to (3). Thus, our test may identify a grounding that does not occur. The test outcomes for S given m_{ij} and E are:

Eq. (1) holds	$\rightarrow$ No groundings. Check next S.
Eq. (1) does not hold	$\rightarrow$ Check Eq. (2).
Eq. (2) holds	$\rightarrow$ Check Eq. (3) for each node receiving S.
Eq. (2) does not hold	$\rightarrow$ Groundings.
Eq. (3) holds	$\rightarrow$ No groundings. Check next S.
Eq. (3) does not hold	$\rightarrow$ Groundings.

Extension We use the test to decide if we merge two nodes. Merging nodes i and j means building their union in terms of parclusters and neighbors. Formally, the union of parclusters $\mathbf{C}_i$ and $\mathbf{C}_j$, denoted by $\mathbf{C}_i \cup \mathbf{C}_j$, is given by $gr(\mathbf{C}_i) \cup gr(\mathbf{C}_j)$. Exploiting that parclusters have certain properties by way of construction, we need not ground but can build the union component-wise. For the local models, we build $F_i \cup F_j$. Regarding graph structure, the merged node k with parcluster $\mathbf{C}_k = \mathbf{C}_i \cup \mathbf{C}_j$ takes over all neighbors from i and j.

Algorithm 1 shows pseudo code for fusion combining the grounding test with merging given an FO jtree J. For each node i in J, we merge i with a neighboring node j until Eqs. (1) to (3) hold if applicable. Algorithm 2 shows LJT with the new step in line 3 after construction. The other steps remain the same.

Algorithm 1 Fusion of FO jtree J to prevent groundings

```
1: function FUSE(FO jtree $J$)
2:     for node $i$ in $J$ do
3:         while $\exists$ node $j \in neighbors(i), E \in \mathbf{E}_{ij}, S \in \mathbf{S}^E_{ij}$ : (Eq. (1) does not hold $\wedge$
                (Eq. (2) does not hold $\vee$ Eq. (3) does not hold)) do
4:             MERGE($i$, $j$)
```

Algorithm 2 Extended Lifted Junction Tree Algorithm

```
1: function FUSEDLJT(Model $G$, Queries $Q$, Evidence $E$)
2:     FO jtree $J$ = FO-JTREE($G$)
3:     FUSE($J$)
4:     ENTEREVIDENCE($J$,$E$)
5:     PASSMESSAGES($J$)
6:     GETANSWERS($J$,$Q$)
```

Theorem 1. *FusedLJT is sound, i.e., produces the same result as LJT.*

Proof sketch. The proof relies on LJT being sound. Fusion alters an underlying FO jtree with merging, preserving the FO jtree properties. Given LJT is sound, LJT works with a valid FO jtree after fusion and produces sound results.

FusedLJT does not induce any unnecessary groundings. As the grounding test over-approximates, we may merge two nodes whose messages do not need groundings. But, no node remains that grounds due to message calculation.

We add a parameter α to encode how many steps our grounding test should follow. If $\alpha = 0$, we do not execute the fusion step. If $\alpha = 1$, we only check Eq. (1) at a node i. It saves work on checking Eqs. (2) and (3) concerning (P)CRVs which may inhibit smaller local models for faster query answering. If $\alpha = 2$, we additionally check Eq. (2) and move on to the next PRV in $\mathbf{S}_{ij}^E$ if it holds. If $\alpha = 3$, we also check Eq. (3) at j. With $\alpha > 3$, we check Eq. (3) at all nodes receiving a (P)CRV with a path length of $\alpha - 3$ starting from j.

Effects We look at the effects of fusion on LJT in terms of data structure and workload. Regarding data structure, effects can range from no change to a collapse into one node. Without a change, we add work for all checks without any merging (no or only model-inherent groundings). Collapsing into one node with the input model in its local model is a worst case scenario: We add overhead for construction and fusion without a payoff as query answering compares to LVE.

Regarding workload, fusion adds to it for checking Eqs. (1) to (3). At a node i, most work occurs if Eq. (1) does not hold for each neighbor j, PRV $E \in \mathbf{E}_{ij}$, and PRV $S \in \mathbf{S}_{ij}^E$. Then, we check for each S that Eq. (2) holds and for each neighbor at nodes k reached through j receiving S that Eq. (3) holds. Let c_1, c_2, and c_3 denote the workload of checking Eqs. (1) to (3), respectively. Let d_k denote the number of neighbors at a node k. Then, a workload amounts of

$$T^{Fusion}(\mathcal{J}) = \sum_i \sum_j |\mathbf{E}_{ij}| \cdot |\mathbf{S}_{ij}^E| \cdot \left(c_1 + c_2 + \sum_k (d_k - 1) \cdot c_3 \right) \tag{4}$$

where i covers the nodes in $\mathcal{J}$, j the neighbors of i, and k the nodes reached through j. If $\alpha = 0$, we do not add work, save for an if-condition check. If $\alpha = 1$ or 2, Eq. (4) ends after c_1 or c_2. With increasing α, we reach more nodes k.

In a worst case, we have the most checks if $\mathbf{E}_{ij}$ and $\mathbf{S}_{ij}$ have the same size, i.e., $\frac{1}{2}|\mathcal{A}_i|$. We replace $\mathcal{A}_i$ with $\mathcal{A}^{max}$ denoting the largest parcluster, meaning the one with the most PRVs. As we cover each edge twice checking each neighbor j of each node i, we rewrite $\sum_i \sum_j$ with $2 \cdot |E|$, E being the set of edges in $\mathcal{J}$. We reformulate $\sum_k$ as $|E| - 1$ since we may cover each edge except i—j. Combined, we have a complexity of $O(|E|^2 \cdot |\mathcal{A}^{max}|^2)$.

5.2 Evidence Handling

This section formalizes evidence handling, a central feature of any inference algorithm. Though we look at evidence in general, we have to keep in mind that computing conditional probabilities, i.e., marginals given evidence, is not liftable unless evidence consists of PRVs with at most one logvar [4].

Extension Entering evidence includes formalizing when we add evidence, how we distribute it, and how we absorb it. We add an evidence parfactor g_E with constraint C_E to local model F_i at node i with constraint C_i in its parcluster iff

$$C_i \cap C_E \neq \emptyset \tag{5}$$

Unlike LVE, we avoid testing all parfactors in G using the third FO jtree property. The property states that if a PRV appears in parclusters $\mathbf{C}_i$ and $\mathbf{C}_j$, it must appear in every parcluster on the path between nodes i and j. To distribute g_E, we find a first node with a parcluster that meets Eq. (5) and add g_E. If adding g_E at a node i, we add g_E to each neighboring node j if C_i projected onto the PRVs in separator $\mathbf{S}_{ij}$ fulfills Eq. (5). After distributing all evidence, we split the parfactors in the local models accordingly and use lifted absorption. (We store the original model for new evidence.)

In the following example, we add the evidence from Example 3 ($Conf(X)$ is true for 10 people) to the FO jtree of G_{ex}.

Example 6. $Conf(X)$ appears in parclusters $\mathbf{C}_2$ and $\mathbf{C}_3$. For lifted absorption in $\mathbf{C}_2$, we split the parfactor in F_2 into $g_2 = \phi_2(Hot, Conf(X), Res(X))|(C_2 \setminus C_E)$ and $g_2' = \phi_2(Hot, Conf(X), Res(X))|C_E$. g_2' absorbs g_E by dropping the values where $\neg conf(X)$ and removing $Conf(X)$ from its arguments. F_2 is now $\{g_2, g_2' = \phi_2(Hot, Conf(X))|C_E\}$. Absorbing g_E in $\mathbf{C}_3$ proceeds analogously.

With new evidence, LJT enters evidence and passes messages again. We can save work on both if evidence changes only incrementally: We only enter changed evidence. After entering evidence, leaf nodes calculate a new message if evidence changed. Without a change, an empty message is sent. Inner nodes calculate a message if their own evidence changed or a non-empty message arrived from any neighbor. Otherwise, they send an empty message.

Theorem 2. *Evidence handling in LJT is sound, i.e., is equivalent to handling evidence in the ground version.*

Proof sketch. The proof relies on the LVE operations, specifically lifted absorption and splitting, and LJT to be sound. Since lifted absorption drops the affected PRVs, we enter evidence at each node that includes evidence randvars. With a correct split and absorption in the local model of a node, the node absorbs evidence correctly. Given LJT is sound, all following computations are sound.

Effects Evidence has an effect on message passing and query answering since the local models change with absorption of evidence. The effect is inherent to handling evidence. Message passing starts after lifted absorption. In case evidence affects sender and receiver, i.e., the evidence PRVs are part of the separator, the message covers those PRVs without evidence since the part with evidence is already absorbed at both parclusters. In case evidence only affects the sender but $logvars(\mathbf{S}_{ij}) \cap logvars(g_E) \neq \emptyset$, the message consists of two parts, one for the part without evidence and one for the part with evidence as F_i is shattered w.r.t. C_E splitting up all occurrences of $logvars(g_E)$. In all other cases, evidence is hidden from the other nodes through summing out.

Entering evidence means checking Eq. (5) for each evidence parfactor $g_E \in \mathbf{E}$ at each node and each parfactor in a local model absorbing g_E in a worst case scenario, leading to a worst case complexity of $O(|N| \cdot |\mathbf{E}| \cdot |F^{max}|)$, N denoting the set of nodes in $\mathcal{J}$ and F^{max} the largest local model. With more evidence, the size of intermediate results decreases and consequently, runtimes fall.

If a change in evidence leads to changes in all nodes, a full message passing run is necessary. With changes only in one part of the model, we save calculating inbound messages from the unchanged part and outbound messages distributing the information from the unchanged part to the remaining model.

6 Empirical Evaluation

We have implemented a prototype of the LJT extended with fusion and evidence handling, named `exfojt`. Taghipour provides a baseline implementation of GC-FOVE including its operators (available at `https://dtai.cs.kuleuven.be/software/gcfove`), named `gcfove` in this test. We include the `gcfove` operators in `exfojt`. We test our implementation against `gcfove`.

We have also implemented a propositional junction tree algorithm as a reference point, named `jt`. `jt` requires substantially more time and memory and therefore, is not part of the discussion. We compare runtimes for inference summed up over queries answered, averaged over several executions per setup.

Fusion We use a variation of G_{ex} as input whose FO jtree has four nodes and requires groundings with $\alpha = 0$ and $\alpha = 2$. If $\alpha = 1$, its FO jtree has two nodes with five PRVs in its largest parcluster after fusion. If $\alpha = 3$, its FO jtree has three nodes with four PRVs in its largest parcluster after fusion. The probability entries are random. We query each PRV once with random groundings.

Figure 5 shows runtimes with increasing domain sizes for `jt` (filled triangle), `gcfove` (circle), and `exfojt` with $\alpha \in \{0, 1, 2, 3\}$ (squares). There is no strong difference noticeable for varying α and the depth of its checks in terms of runtime with this limited example. Between construction and message passing, the fusion step does not add significantly to the overhead.

The runs of `exfojt` with groundings (filled squares) have a runtime worse than the runtime of `jt` for small domains. Runtimes for `exfojt` without groundings (empty squares) and `gcfove` do not have the steep increase in runtime with larger

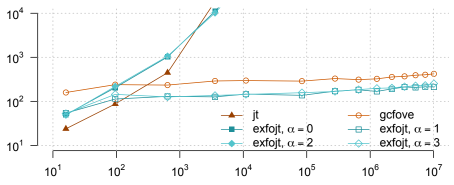

Fig. 5: Runtimes [ms], x-axis: $|gr(G_{ex})|$ from 16 to 10,100,000 (log scale)

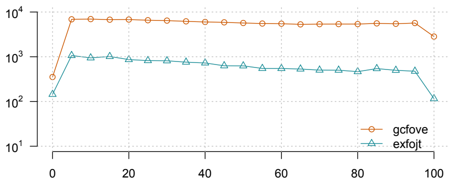

Fig. 6: Runtimes [ms] (log scale), x-axis: evidence from 0% to 100%

domains. `exfojt` ($\alpha = 1$, $\alpha = 3$) needs 30 to 60% of the time `gcfove` needs which it trades off with memory. It requires 1.06 to 1.16 times the memory of `gcfove`. The decrease in runtime is mirrored in the number of LVE operations performed, independent of the size of the grounded model: `exfojt` ($\alpha = 1$) performs 170 operations (181 with $\alpha = 3$). `gcfove` performs 368 operations.

Evidence We use G_{ex} as input with random probability entries and set $\alpha = 0$. We enter evidence on all PRVs except Hot and $Pub(X, P)$ ranging from 0% to 100% in 5% steps. We query each PRV once and $Pub(X, P)$ twice with random groundings. We fix the domain sizes, yielding $|gr(G_{ex})| = 111,000$.

Figure 6 shows runtimes with increasing evidence coverage. On all evidence settings, `exfojt` (triangles) outperforms `gcfove` (circles). Entering evidence increases runtimes since handling evidence costs time. With more evidence, runtimes decrease for both programs as a larger part of the model is fixed with evidence. Apart from the settings with 0% and 100% evidence, `exfojt` needs 8 to 16% of the time `gcfove` needs. In terms of VE operations, `exfojt` needs 128 operations (including message passing) against 475 by `gcfove`. `exfojt` trades off runtimes with memory. It requires 1.2 to 1.4 times the memory of `gcfove`.

With its static overhead, `exfojt` outperforms `gcfove` with the second query at 0% evidence. In all other cases, `exfojt` is faster with the first query.

In summary, spending effort on building an FO jtree and passing messages pays off. Even with little evidence, `exfojt` runs faster after the first query.

7 Conclusion

We present extensions to LJT to answer multiple queries efficiently in the presence of symmetries in a model. We identify when LJT induces unnecessary groundings during message passing. To remedy this effect, we add a step to LJT that merges parclusters. Additionally, we formalize how LJT handles evidence. We speed up runtimes significantly, especially with evidence, for answering multiple queries compared to the current version of LJT and GC-FOVE.

We currently work on adapting LJT to incrementally changing knowledge bases. 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

1. Ahmadi, B., Kersting, K., Mladenov, M., Natarajan, S.: Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training. In: Machine Learning, vol. 92, pp. 91–132. Kluwer Academic Publishers, Hingham (2013)
2. Bellodi, E., Lamma, E., Riguzzi, F., Santos Costa, V., Zese, R.: Lifted Variable Elimination for Probabilistic Logic Programming. In: Theory and Practice of Logic Programming, vol. 14(4–5), pp. 681–695. Cambridge University Press, Cambridge (2014)
3. Braun, T., Möller, R.: Lifted Junction Tree Algorithm. In: KI 2016: Advances in Artificial Intelligence - 39th Annual German Conference on AI, pp. 30–42. Springer, Cham (2016)
4. Van den Broeck, G.: Lifted Inference and Learning in Statistical Relational Models. PhD Thesis, KU Leuven (2013)
5. Choi, J., Amir, E., Hill, D. J.: Lifted Inference for Relational Continuous Models. In: Proceedings of the 26th Conference on Artificial Intelligence. The AAAI Press, Menlo Park (2012)
6. Darwiche, A.: Recursive Conditioning. In: Artificial Intelligence, vol. 2, pp. 4–41. Elsevier Science Publishers, Essex (2001)
7. Darwiche, A.: Modeling and Reasoning with Bayesian Networks. Cambridge University Press, Cambridge (2009)
8. 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. Society for Industrial and Applied Mathematics, Philadelphia (2016)
9. 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. The AAAI Press, Menlo Park (2010)
10. Gogate, V., Domingos, P.: Probabilistic Theorem Proving. In: Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence, pp. 256–265. AUAI Press, Arlington (2011)
11. Jensen, F.V., Lauritzen, S.L., Oleson, K. G.: Bayesian Updating in Recursive Graphical Models by Local Computations. In: Computational Statistics Quarterly, vol. 4, pp. 269–282. Physica-Verlag, Vienna (1990)
12. Lauritzen, S. L., Spiegelhalter, D. J.: Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. In: Journal of the Royal Statistical Society: Series B (Statistical Methodology), vol. 50, pp. 157–224. Wiley-Blackwell, Oxford (1988)
13. Milch, B., Zettlemoyer, L.S., Kersting, K., Haimes, M., Pack Kaelbling, L.: Lifted Probabilistic Inference with Counting Formulas. In: Proceedings of the 23rd Conference on Artificial Intelligence, pp. 1062–1068. The AAAI Press, Menlo Park (2008)
14. Poole, D.: First-Order Probabilistic Inference. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence, pp. 985–991. Morgan Kaufman Publishers Inc., San Francisco (2003)
15. de Salvo Braz, R.: Lifted First-Order Probabilistic Inference. PhD Thesis, University of Illinois at Urbana-Champaign (2007)
16. Shafer, G.R., Shenoy, P.P.: Probability Propagation. In: Annals of Mathematics and Artificial Intelligence, vol. 2, pp. 327–351. Springer, Heidelberg (1989)

17. Singla, P., Domingos, P.: Lifted First-Order Belief Propagation. In: Proceedings of the 23rd Conference on Artificial Intelligence, pp. 1094–1099. The AAAI Press, Menlo Park (2008)
18. Taghipour, N.: Lifted Probabilistic Inference by Variable Elimination. PhD Thesis, KU Leuven (2013)
19. Taghipour, N., Davis, J., Blockeel, H.: First-order Decomposition Trees, In: Advances in Neural Information Processing Systems 26, pp. 1052–1060. Curran Associates, Red Hook (2013)
20. Vlasselaer, J., Meert, W., van den Broeck, G., de Raedt, L.: Exploiting Local and Repeated Structure in Dynamic Baysian Networks. In: Artificial Intelligence, vol. 232, pp. 43–53. Elsevier, Amsterdam (2016)
21. 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. Morgan Kaufman Publishers, San Francisco (1994)