Lifted Junction Tree Algorithm

Tanya Braun and Ralf Möller

Institute of Information Systems, Universität zu Lübeck, Lübeck
{braun,moeller}@ifis.uni-luebeck.de

Abstract. We look at probabilistic first-order formalisms where the domain objects are known. In these formalisms, the standard approach for inference is lifted variable elimination. To benefit from the advantages of the junction tree algorithm for inference in the first-order setting, we transfer the idea of lifting to the junction tree algorithm.
Our lifted junction tree algorithm aims at reducing computations by introducing first-order junction trees that compactly represent symmetries. First experiments show that we speed up the computation time compared to the propositional version. When querying for multiple marginals, the lifted junction tree algorithm performs better than using lifted VE to infer each marginal individually.

Keywords: Probabilistic Logical Models, Lifted Inference, Junction Tree, Belief Propagation

1 Introduction

New probabilistic logical representation formalisms support first-order logic, rather than just propositional logic, and one can reason about sets of individuals in a relational domain. To express patterns or symmetries in the relation between individuals, we combine random variables (randvars) with logical variables (logvars) to denote a whole set of randvars (parameterized randvars, PRVs). In an undirected formalism with known domain objects, the idea of lifting is to use these patterns and symmetries to infer knowledge faster.

A small example that serves as a running example for the upcoming sections is a knowledge base (KB) G_{ex} with PRVs $epidemic(D)$ and $sick(D, P)$. The PRV $epidemic(D)$, for example, could stand for two propositional randvars if logvar D had the two instantiations flu and $measles$.

In general, we study the inference task of computing marginal distributions. Many approaches and applications need optimizations to enhance efficiency. For propositional representation languages, variable elimination (VE) [19] speeds up computation. VE decomposes a KB into parts that we can solve faster. In the first-order context, lifted VE [12] aims at answering queries more efficiently by exploiting symmetries captured in PRVs. More specifically, with PRVs in a KB, we have parameterized factors (potential functions), called parfactors for short, that have PRVs as arguments. A parfactor represents a set of factors with an identical potential function, e.g., a probability distribution. Lifted VE uses the symmetries in the potential functions to reduce the number of computations carried out.

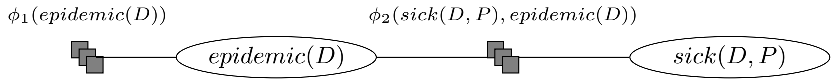

Fig. 1. Parfactor Graph for G_{ex}

Figure 1 shows a graphical representation of G_{ex} with PRVs $epidemic(D)$ and $sick(D, P)$ and parfactors $\phi_1(epidemic(D))$ and $\phi_2(sick(D, P), epidemic(D))$. The graph consists of two variable nodes, one for each randvar in G_{ex}, and two factor nodes for the two parfactors. The factor nodes have edges to the nodes of the randvars involved. E.g., factor $\phi_1(epidemic(D))$ denotes that all randvars for which $epidemic(D)$ stands have the same potential function ϕ_1, e.g., a prior probability for some epidemic to occur.

When asking multiple queries in the propositional case, an optimization is the junction tree algorithm [7]. It allows to compute all marginal distributions efficiently instead of answering queries individually with VE. The junction tree algorithm is designed for query answering with respect to KBs specified with undirected formalisms. We can transfer directed formalisms into undirected ones by moralizing the underlying graphs or by building decomposition trees (dtrees) [7]. Dtrees are tree representations of the decomposition of a KB during VE. The junction tree algorithm supports exact reasoning through message passing where we basically apply VE in all directions at a time. In the context of junction trees, message passing distributes "knowledge" in a graph. It does not approximate in itself. With symmetries present, many unnecessary messages are sent. We transfer the idea of lifting to the junction tree algorithm to optimize the junction tree representation and message handling. We illustrate our findings in the evaluation with an extended example where we show that the advantages of a junction tree transfer from the propositional to the first-order setting.

This paper contributes the following: We propose a lifted junction tree algorithm for inference in probabilistic logical KBs. We lift the algorithm by building a lifted (first-order) junction tree (FO jtree). To this end, we introduce parameterised clusters (parclusters) that, similar to parfactors, support logvars to capture symmetries. We modify the message passing scheme to operate on FO jtrees. When calculating messages and results to queries, we integrate lifted VE.

The representation language and lifted VE operators we use heavily rely on Taghipour's work[17] (and the papers cited therein). Taghipour also introduces lifted (first-order) dtrees (FO dtrees) based on [6] and gives a simple algorithm to find one for a given KB. We use FO dtrees to build FO jtrees.

In terms of performance, the lifted junction tree algorithm imposes some static overhead due to the junction tree construction and message passing. But, with multiple queries or varying evidence where the tree is reusable, the overhead amortizes and becomes more and more negligible compared to the junction tree speed-up. According to our experiments, we significantly speed up run times in the presence of symmetries compared to the grounded version. Additionally, we speed up inference compared to lifted VE if asking multiple queries.

The remainder of this paper starts with related work on lifted inference and belief propagation followed by background information on the junction tree algorithm, parameterized KBs, and FO dtrees. Next, we introduce our lifted junction tree algorithm. Additionally, we present an evaluation of our algorithm with promising results. We conclude the paper by looking at future work.

2 Related Work

We present related work in the area of probabilistic (first-order) formalisms, focusing on the junction tree algorithm and lifted inference.

Basic junction tree algorithms, specifically, their message passing schemes, use one of two architectures. Shafer and Shenoy [13] propose the first architecture under the name probability propagation, often called Shafer-Shenoy. Jensen et al. [9] introduce the second architecture, nowadays referred to as Hugin. Both architectures have a *collect* and a *distribute* phase but vary with respect to what they store and how they compute messages. On the one hand, Shafer-Shenoy is more space-efficient than Hugin. On the other hand, Hugin usually is faster. Hugin saves time by doing fewer computations per message but requires more space to store larger intermediate results. We adapt the ideas of both architectures to pass and process messages in our lifted algorithm.

Darwiche [7] provides the foundation for the dtrees as we use them and the connection between dtrees and junction trees. His work on recursive conditioning [6] and local symmetry (the latter together with Chavira [5]) provides ideas on how to further utilize first-order structures in different ways.

Lifted inference has been the focus of research for some years now. The first formalizations of lifted inference go back to [12], named FOVE for first-order VE. The research presented in [3, 10, 17] extends the formalism to the standard form GC-FOVE of lifted VE with generalized counting. We use the lifting operators in GC-FOVE for internal lifted calculations in our algorithm.

Parallel to lifted VE, weighted first-order model counting emerges using the lifting idea applied to weighted model counting for inference [4]. Another branch, lifted belief propagation (BP), picks up the idea of probability propagation and combines it with lifting. Often, the work on belief propagation is accompanied with lifted representations. The work of Singla and his colleagues includes BP on a lifted network, using hypercube-based representations, and an approximate lifted BP, to approximate lifting in presence of noise [14–16]. Gogate uses hypercubes as well for a lifted representation [8]. Ahmadi et al. [1] provide a counting BP using a coloring algorithm including an extension to dynamic Bayesian networks. Though lifted BP uses belief propagation similar as we do, none of the approaches given uses a lifted version of junction trees.

The junction tree algorithm provides an efficient alternative to inference if confronted with the need to answer multiple queries or queries under varying evidence. Lifting provides an idea to further optimize inference by handling symmetries in an efficient way. We take the propositional version and adapt it to a first-order setting by modifying the underlying tree structure. Additionally, we revise the propositional algorithm to deal with first-order constructs efficiently. We alter the computing instructions for delivering results to queries as well as assembling messages to incorporate lifted VE instead of ground VE. Overall, we propose a lifted algorithm that compactly represents clusters in a KB and efficiently handles inference on them.

3 Background

This section presents the standard junction tree algorithm and introduces definitions for parameterized KBs and FO dtrees. The first subsection is based on [7], the second on [17]. Taghipour [17] calls the KB we work on a model. We use the term model for the remainder of this paper. We assume familiarity with common notions such as dtrees and its properties cutsets, contexts, and clusters (for an introduction, see also the appendix in [2]).

3.1 Junction Tree Algorithm

In inference, we query models, e.g., a factor graph, given some evidence. For one query, VE is the standard approach. With multiple queries, we look for a data structure that allows to pre-compute recurring calculations for faster query answering. Junction trees (jtrees) serves as such a data structure. Jtree nodes represent sets of variable nodes of the underlying model, called clusters. One algorithm run distributes knowledge in the underlying jtree. At the end, a node holds all information to compute marginal probabilities for its variables.

Intuitively, clusters consist of elements (i.e., randvars) that share close relations, through factors, otherwise not present in the model. Randvars that contribute to various clusters inform the structure of the jtree. All clusters that share a randvar build a subgraph to ensure that if local changes in one cluster influence a randvar, the effect is communicated to the other clusters. To construct a jtree, we build a dtree and compute its clusters. A dtree represents a decomposition of a model during VE. The dtree structure and its clusters associated with each node form a jtree.

A factor is associated with a cluster that includes the factor's arguments. Evidence influences arguments. If we enter evidence in the graph at one end, we propagate that information to other parts using messages. After propagating all information (factors are information as well), we answer queries by looking at clusters that contain the query variables. Starting from evidence, we compute aggregations of factors by propagating information from node to node. We reuse the jtree with new evidence.

Next, we formalize the dtree and jtree data structure and the junction tree algorithm. A dtree for a graph G is a tree whose leaf nodes correspond to the factors in G. An inner node represents a decomposition of its factors into partitions, one for each child, containing the factors in the child's subtree. The cluster of a dtree node N is the union of its cutset and context. The cutset of N is the set of randvars shared between any pair of its child nodes. In case of N not being the root, we subtract the randvars appearing in its ancestor cutsets. The context of N is the intersection of its randvars and those in its ancestor cutsets.

A jtree for a graph G is a pair $(\mathcal{J}, \mathbf{C})$ where $\mathcal{J}$ is a tree (the structure of the jtree) and $\mathbf{C}$ is a function that maps each node i in $\mathcal{J}$ to a label $\mathbf{C}_i$ called a cluster. The mapping function effectively makes the clusters and nodes of $\mathcal{J}$ interchangeable. A jtree must satisfy three properties: (i) A cluster $\mathbf{C}_i$ is a set of nodes from G. (ii) For every edge X—Y in G, variables X and Y appear in some cluster $\mathbf{C}_i$. (iii) If a node from G appears in clusters $\mathbf{C}_i$ and $\mathbf{C}_j$, it must appear in every cluster $\mathbf{C}_k$ on the path between nodes i and j in $\mathcal{J}$. $\mathbf{S}_{ij}$, called the separator of edge i—j, holds those randvars shared by clusters $\mathbf{C}_i$ and $\mathbf{C}_j$ and is given by $\mathbf{C}_i \cap \mathbf{C}_j$.

A jtree is minimal if by removing a variable from any cluster, the jtree stops being a jtree The clusters of a dtree fulfil the jtree properties. But, the resulting jtree is seldom minimal. We merge two neighboring clusters if the randvars in one of them is a subset of the randvars in the other.

The main workflow to answer queries is to construct the jtree, pass messages, and then answer queries. We modify the junction tree algorithm from [7] using potential functions instead of conditional probability tables (CPTs). The algorithm consists of a preparation phase and the actual algorithm. The preparations incorporate three steps: (i) Construct a jtree. (ii) Assign each potential function ϕ to a cluster that contains its randvars. (iii) Assign for each randvar X an evidence indicator λ_X to a cluster that contains X. We use evidence indicators to assign an observed value of a randvar. By multiplying an indicator with a potential function, we incorporate the evidence into the model. For each cluster, we multiply the assigned factors.

The algorithm itself is short: We enter evidence $\mathbf{e}$ through the indicators. Then, the algorithm sends messages to distribute knowledge. Message M_{ij} from node i to node j holds new information for j encoded in a readable way: M_{ij} is a product of the factor assigned to i and the messages received from other nodes but j projected onto $\mathbf{S}_{ij}$ by summing out $\mathbf{C}_i \setminus \mathbf{S}_{ij}$. Message computation is a form of VE, summing out variables unknown at the receiver node. We can interpret a message as a factor with the separator variables as arguments. After passing two messages per edge, we can compute, e.g., marginal $P(\mathbf{C}, \mathbf{e})$ for every cluster $\mathbf{C}$. To answer a query, we can now use any cluster that contains the query variables and sum out all other variables.

3.2 Parameterized Models and FO Dtrees

Parameterized models provide a compact way to specify KBs using first-order constructs allowing lifted VE. To enable a compact dtree representation, we need a first-order version, which we use for constructing FO jtrees.

First, we define a few useful shortcuts. Consider the PRV $epidemic(D)$ representing a set of randvars depending on the instantiations of D. We call the possible values of D its domain, denoted $\mathcal{D}(D)$. Assuming $epidemic(D)$ is binary, the range of each randvar represented by $epidemic(D)$ is $true$ and $false$: $range(epidemic(D)) = \{true, false\}$. The term $logvar(P)$ denotes the logvars in a set or sequence of randvars P (e.g., $logvar(epidemic(D)) = \{D\}$). Domains constrain the instantiations of a PRV to a given set by specifying the values of its logvars. As PRVs are arguments to parfactors, parfactors are subject to constraints as well. We introduce a constraint C to specify instantiations of logvars. Taghipour defines a constraint C as a tuple $(\mathbf{X}, C_{\mathbf{X}})$, with $C_{\mathbf{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X_i)$ where $\mathbf{X} = (X_1, \dots, X_n)$ is a tuple of logvars.

To specify a parfactor g, we need the potential function with its arguments, i.e. PRVs, and a constraint on the logvars in g. Formally, g is given by

$$g := \forall \mathbf{L} \ : \ \phi(\mathcal{A}) \mid C$$

where $\mathbf{L}$ is a set of logvars that the factor generalizes over. $\mathcal{A} = (A_1, \dots, A_n)$ is a sequence of randvars. If $\mathbf{L} = logvar(\mathcal{A})$, we omit $\forall \mathbf{L}$ in the parfactor. $\phi = \times_{i=1}^{n} range(A_i) \rightarrow \mathbb{R}^+$ is a potential function with values of $\mathcal{A}$ as input.

A model G is given by a set of parfactors $\{g_i\}_{i=1}^{n}$. Model G_{ex} becomes the parfactor model $G_{ex} = \{g_1, g_2\}$ with $g_1 = \phi_1(epidemic(D))|C_1$, $g_2 = \phi_2(sick(D, P),$ $epidemic(D))|C_2$. Let $\mathcal{D}(D) = \{flu, measles\}$ and $\mathcal{D}(P) = \{alice, eve, bob\}$. Then we could define C_2 by $((D, P), C_{(D,P)})$ and $C_{(D,P)} = \{(flu, eve), (flu, bob),$ $(flu, alice), (measles, eve), (measles, bob), (measles, alice)\}$.

Lifting uses the fact that we can decompose a model into isomorphic subproblems and solve only one representative. For our algorithm, we use the lifting operators defined by Taghipour (for full definitions including pre- and postconditions, see [17]). For example, we use lifted absorption when entering evidence in the FO jtree or lifted summing-out when computing messages.

First-Order Dtrees We now define a compact representation of the decomposition of a model. Logvars allow us to ground models partially by grounding a subset of the logvars and work with representatives of the grounded logvars. If a model is in a certain normal form, we can decompose it into partial groundings isomorphic up to permutations of inputs. For details on the normal form and decomposition into partial groundings (DPG), see also the appendix in [2].

Isomorphic decomposition (ID) nodes represent isomorphic partial groundings in the FO dtree. An ID node $T_{\mathbf{X}}$ is given by a triplet $(\mathbf{X}, \mathbf{x}, C)$ where $\mathbf{X} = \{X_1, \dots X_k\}$ is a set of logvars all of the same domain $\mathcal{D}_{\mathbf{X}}$, $\mathbf{x} = \{x_1, \dots x_k\}$ is a set of *symbolic constants* from $\mathcal{D}_{\mathbf{X}}$, and C is a constraint on $\mathbf{x}$, such that for $i, j : x_i \neq x_j \in C$. We denote $T_{\mathbf{X}}$ by $\forall \mathbf{x} : C$ in the FO dtree. $T_{\mathbf{X}}$ has one child named $T_{\mathbf{x}}$. The model under $T_{\mathbf{x}}$ is a representative instance of $T_{\mathbf{X}}$.

An FO dtree represents a decomposition of a parfactor model during lifted VE with parfactors in its leaves and ID nodes to model representative instances. Formally, an FO dtree is a tree that can have ID nodes and in which (i) each leaf contains a factor (possibly with symbolic constants), (ii) each leaf with symbolic constant x is the descendent of exactly one ID node $T_{\mathbf{X}}$ such that $x \in \mathbf{x}$, (iii) each leaf that is a descendent of ID node $T_{\mathbf{X}}$ has all symbolic constants $\mathbf{x}$ in its factor, and (iv) for each ID node $T_{\mathbf{X}}$, $\mathbf{X} = \{X_1, \dots X_k\}$, $T_{\mathbf{x}}$ has $k!$ children $\{T_i\}_{i=1}^{k!}$, which are isomorphic up to a permutation of symbolic constants $\mathbf{x}$.

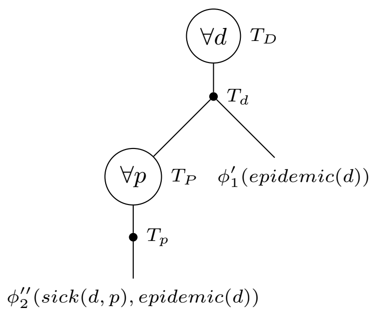

Fig. 2. FO Dtree for G_{ex}

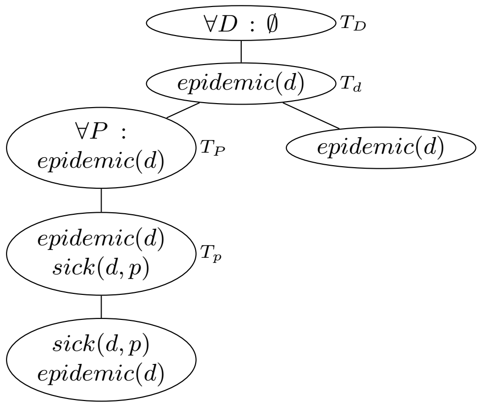
$\forall D : \emptyset$
T_D
$epidemic(d)$
T_d
$\forall P :$ $epidemic(d)$
T_P
$epidemic(d)$
$epidemic(d)$ $sick(d,p)$
T_p
$sick(d,p)$ $epidemic(d)$

Fig. 3. FO Jtree for the FO Dtree in Fig. 2

For an FO dtree, one can compute a cluster for each node analogously to computing clusters for propositional dtrees. The appendix in [2] shows how to build an FO dtree for a model and how to compute clusters for FO dtrees.

Figure 2 shows an FO dtree for G_{ex}. For readability purposes, we only write the element for singleton sets and omit $\top$ constraints. The root is an ID node $T_D = (D, d, \top)$ (logvar D allows a DPG). T_D has a child T_d with the model $G' = \{g'_1 = \phi'(epidemic(d)), g'_2 = \phi'(sick(d,P), epidemic(d))\}$. G' does not allow a DPG (g'_1 has no logvar). Hence, G' is split based on the occurrence of P. Thus, T_d gets two children. The right child with the model $\{g'_1 = \phi'(epidemic(d))\}$ is ground. It has only one factor in its model which results in a leaf node with factor $\phi'(epidemic(d))$. The left child with the model $\{g'_2 = \phi'(sick(d,P), epidemic(d))\}$ has a logvar, P, that permits a DPG. Hence, the child is an ID node $T_P = (P, p, \top)$ with a child T_p with the model $\{g''_2 = \phi''(sick(d,p), epidemic(d))\}$. g''_2 is ground and only consists of one factor as well, so the child is a leaf node with factor $\phi''(sick(d,p), epidemic(d))$ contained in it.

4 Lifted Junction Tree Algorithm

This section presents our lifted version of the junction tree algorithm including FO jtrees and parclusters.

4.1 First-Order Junction Trees

FO jtrees follow the idea of ground jtrees. Clusters combine PRVs with close relations and message passing distributes knowledge to enable efficient query answering of many queries. A ground jtree in the presence of symmetries has many nodes with identical factors where messages propagate information that basically is already present. We allow parameterized randvars to capture symmetries and parameterize the notion of a cluster to represent a subgraph of grounded clusters with identical potential functions.

Algorithm 1 Constructing an FO Jtree for a Model G Using an FO Dtree

```
function FO-JTREE($G$)
    FO dtree $T$ = FO-DTREE($G$)
    Compute clusters for $T$
    Construct FO jtree J
    Minimize J
    return J
```

Parclusters Intuitively, parclusters are the nodes of an FO jtree formed by FO dtree clusters. A parcluster describes the set of randvars in a cluster and can have factors assigned. It generalize over logvars if ID nodes are involved.

Formally, a parcluster $\mathbf{C} = \forall \mathbf{L} : \mathcal{A}|C$ is a set of randvars $\mathcal{A}$. The parameters of $\mathbf{C}$ are the set of logvars $\mathbf{L}$ and $logvar(\mathcal{A}) \subseteq \mathbf{L}$. The constraint C puts limitations onto logvars and symbolic constants. A factor $\phi(\mathcal{A}_\phi)|C_\phi$ assigned to $\mathbf{C}$ has to fulfil (i) $\mathcal{A}_\phi \subseteq \mathcal{A}$, (ii) $logvar(\mathcal{A}_\phi) \subseteq \mathbf{L}$, and (iii) $C_\phi \subseteq C$.

As jtrees built from dtrees often are non-minimal, we define a merge operation for parclusters. Parclusters $\mathbf{C}_i$ and $\mathbf{C}_j$ with possibly assigned factors ϕ_i and ϕ_j can merge if $\mathcal{A}_i \subseteq \mathcal{A}_j \vee \mathcal{A}_j \subseteq \mathcal{A}_i$ holds. The merged parcluster $\mathbf{C}_k$ and its assigned factor ϕ_k are determined by

- $\mathcal{A}_k = \mathcal{A}_i \cup \mathcal{A}_j$,
- $\mathbf{L}_k = \mathbf{L}_j \cup \mathbf{L}_i$,
- $C_k = C_i \cup C_j$, and
- $\phi_k = \begin{cases} \phi_i & \text{if only } \phi_i \text{ exists} \\ \phi_j & \text{if only } \phi_j \text{ exists} \\ \phi_i \otimes \phi_j & \text{if both exist} \\ undefined & \text{otherwise.} \end{cases}$

The new node k takes over all neighbours of i and j. To merge a parcluster with logvars and another with corresponding symbolic constants, we first perform an inverse substitution from symbolic constants to logvars and then merge.

FO Jtrees An FO jtree for a model G is a graph with parclusters as nodes. It must also satisfy the three properties introduced for ground jtrees. Grounding an FO jtree leads to a jtree that could have been built by converting a ground dtree into a jtree. The set of factors in the grounded version of the FO jtree is identical to the set of factors in the ground jtree. Algorithm 1 shows pseudo code for constructing an FO jtree for a model G. First, it constructs an FO dtree and computes its clusters. Then, it builds an FO jtree using the clusters. Finally, the FO jtree is minimized by merging parclusters.

The clusters of the FO dtree in Fig. 2 lead to the FO jtree in Fig. 3. The labels next to the nodes indicate from which node a cluster came. The FO jtree shown is not minimal. Iteratively merging leaf parclusters with their neighbor leads to a single parcluster $\mathbf{C} = \forall D, P : \{epidemic(D), sick(P, D)\}$ with ϕ_1 and ϕ_2 assigned to it.

Algorithm 2 Lifted Junction Tree Alg. for Model G, Queries Q, Evidence E

```
function FOJT($G$, $Q$, $E$)
    FO jtree $J$ = FO-JTREE($G$)
    $J$.ENTEREVIDENCE($E$)
    $J$.PASSMESSAGES
    $J$.GETANSWERS($Q$)
```

4.2 Algorithm Description

Our lifted junction tree algorithm has the following workflow: (i) Construct an FO jtree for a given model G. (ii) Enter evidence E into the tree. (iii) Pass messages. (iv) Compute answers. Algorithm 2 shows a corresponding pseudo code description, which uses Algorithm 1.

FO jtree construction uses the clusters of an FO dtree for model G. After the FO jtree construction, we enter evidence E, a set of evidence parfactors. We assign an evidence parfactor to a parcluster if the represented set of the parcluster randvars includes the randvars of the evidence parfactor.

Message passing on FO jtrees proceeds analogously to the one on grounded jtrees. A message from node i to node j is a factor over the randvars in separator $\mathbf{S}_{ij}$ where all other randvars in parcluster $\mathbf{C}_i$ are summed out. The messages and factors include PRVs which allow us to use lifted VE for computations. Message passing starts from the leaves and moves inward (*collect* phase). When all neighbors but one have sent messages to a node, the node itself sends a message to the remaining neighbor. Sending messages in such a way leads to one or two nodes in the center of the jtree to have received messages from all neighbors. Then, the *distribute* phase begins. The one or two nodes send messages to all its neighbors. If now a node receives a message (from the node to which it sent a message during the *collect* phase), it sends messages to all other neighbors. With new evidence, we repeat message passing.

To answer a query, we identify a cluster with the query terms in its domain and sum out all other terms in its parfactor and messages received. For answering queries (or handling evidence), we need the lifted VE operator splitting to rewrite the model to permit lifted summing out. (A parfactor is split into one parfactor for the query term (or the terms for which we have evidence) and one for the other instances of the logvar.)

Compared to the ground version, we do not change the algorithm structure much to lift it. We enter evidence and pass messages. The preparation phase and how we handle evidence, messages, and queries vary. In a dtree, the leaves hold the factors and every factor appears in exactly one leaf. Therefore, clusters have assigned factors and our merge operation maintains them. Additionally, we do not assign evidence indicators as we use evidence parfactors to handle evidence in a lifted manner. So, the preparation phase (construction, assign factors, assign indicators) melts down to construction. For message and query computation, we incorporate lifted VE operators to further optimize calculations.

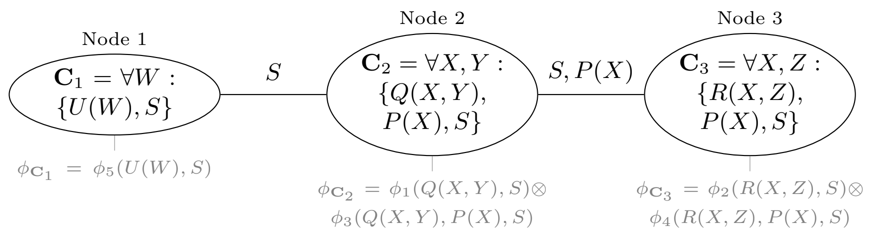
Node 1
$\mathbf{C}_1 = \forall W : \{U(W), S\}$
$\phi_{\mathbf{C}_1} = \phi_5(U(W), S)$
S
Node 2
$\mathbf{C}_2 = \forall X, Y : \{Q(X,Y), P(X), S\}$
$\phi_{\mathbf{C}_2} = \phi_1(Q(X,Y), S) \otimes \phi_3(Q(X,Y), P(X), S)$
$S, P(X)$
Node 3
$\mathbf{C}_3 = \forall X, Z : \{R(X,Z), P(X), S\}$
$\phi_{\mathbf{C}_3} = \phi_2(R(X,Z), S) \otimes \phi_4(R(X,Z), P(X), S)$

Fig. 4. FO Jtree for G_{ex2}

5 Evaluation

We compare our lifted algorithm with GC-FOVE and a propositional version of the junction tree algorithm. We have implemented our lifted junction tree algorithm with Shafer-Shenoy message passing as a Java program that builds on GC-FOVE [18] which is an extension of C-FOVE [10] in BLOG [11].

First experiments exhibit promising results. Since lifting is relevant in the presence of symmetries, the experiments focus on models with symmetries. Additionally, inference benefits from our approach particularly if asking several queries. Although GC-FOVE has to eliminate all non-query randvars in the model, it is usually faster than our algorithm if asking only one query as it does not have to construct an FO jtree. With multiple queries, though FO jtree construction and message passing impose some static overhead, our algorithm needs considerably fewer computations to answer queries. If evidence changes, we can reuse the FO jtree and only add the overhead of a message passing run.

We illustrate our findings with an extended model $G_{ex2} = \{g_1, g_2, g_3, g_4, g_5\}$. The parfactors are defined as follows:

- $g_1 = \phi_1(Q(X,Y), S)$
- $g_2 = \phi_2(R(X,Z), S)$
- $g_3 = \phi_3(Q(X,Y), P(X), S)$
- $g_4 = \phi_4(R(X,Z), P(X), S)$
- $g_5 = \phi_5(U(W), S)$

The domains of the four logvars are of size four. The randvars are binary and the potential functions are CPTs with random entries. We have no evidence and the queries are $P(S)$, $P(U(w_1))$, $P(P(x_1))$, $P(Q(x_1, y_1))$, and $P(R(x_1, z_1))$. Figure 4 displays the minimized FO jtree for G_{ex2} consisting of three nodes with the associated parfactors as labels. The two parfactors of parclusters $\mathbf{C}_2$ and $\mathbf{C}_3$ are multiplied into one parfactor using lifted multiplication during merging.

Comparison with GC-FOVE We feed our algorithm with the model and the five queries as input and receive as answers five probability distributions. We run GC-FOVE with the same model and each query individually. Asking

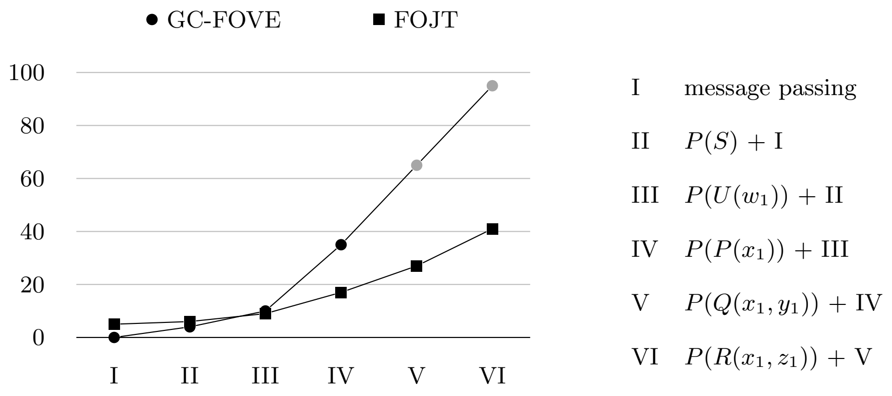

Fig. 5. Accumulated Number of Calls to Lifted VE Operators During Query Answering by GC-FOVE and FOJT (The lines between data points are only there for readibility. The gray dots denote that the counts include queries that GC-FOVE aborted.)

multiple queries in GC-FOVE may lead to dependencies between query terms which causes GC-FOVE to abort.

For the queries $P(S)$, $P(U(w_1))$, and $P(P(x_1))$, we get three probability distributions identical to the ones from our algorithm. The query $P(Q(x_1, y_1))$ leads GC-FOVE to terminate and prompt the following error message: `Fatal error: Parfactor[qe(x1, y1), qe($318, y1)] :{constraint} still contains a non-query term`. When GC-FOVE aborts, it has already performed 30 split and sum-out operations to no avail. Query $P(R(x_1, z_1))$ causes the same problem.

Figure 5 shows the accumulated number of calls to lifted VE operators during answering the five queries for GC-FOVE and our algorithm called FOJT in the figure. We initialize the accumulated counts with the number of calls to lifted VE operators during message passing for FOJT. The order of the added counts is from the shortest to the longest query in terms of operator calls. GC-FOVE has to always eliminate all randvars in the model except for the query randvar. In contrast, after message passing, FOJT only has to eliminate all non-query randvars in a parcluster. E.g., for $P(U(w_1))$, we take the final parfactor at $\mathbf{C}_1$, $\phi_{\mathbf{C}_1}$, and sum out $U(W), W \neq w_1$ (with lifted VE) and S. GC-FOVE has to additionally sum out $P(X)$, $Q(X, Y)$, and $R(X, Z)$.

With the second query, FOJT needs fewer overall calls. Considering that we also need to construct the FO jtree, the test run supports our statement that with only very few queries to answer, the overhead of our lifted junction tree algorithm does not amortize. But with an increasing number of queries, our approach becomes more and more efficient.

Overall, GC-FOVE calls the splitting operator over three times more and the summing-out operator over one and a half times more than our algorithm as it has to eliminate all non-query terms for each query. We pay the savings in calls with time spent on constructing the FO jtree and passing messages. The message passing, which involves four messages being sent, takes another five calls to the summing-out operator.

A brief look at runtimes shows that FOJT appears to be competitive. GC-FOVE has an accumulated runtime of ~ 55 ms for model G_{ex2} and the three queries it can answer. The runtime of the implementation of our algorithm is ~ 40 ms which ~ 15 ms faster and answers two more queries. With only the three queries GC-FOVE can answer, our algorithm needs ~ 24 ms.

Comparison with Ground Version The ground jtree for G_{ex2} has 36 clusters, 4 clusters of form $\{U(w_i), S\}$, 16 clusters of the form $\{Q(x_i, y_j), P(x_i), S\}$, and 16 clusters of the form $\{R(x_i, z_j), P(x_i), S\}$. If we merge the clusters of the dtree to have one virtual root, e.g., $\{Q(x_1, y_1), P(x_1), S\}$, with 35 neighbors, we send 70 messages instead of 4. In the *collect* phase, we perform 35 grounded sum-out operations. For each message in the *distribute* phase, we calculate a product with 35 multiplicands and sum out one or two ground randvars. At the end, each leaf node has to multiply the received message into its factor and the virtual root has to multiply all its messages and its factor. To answer queries, we have less work as we only need to sum out one or two ground randvars.

For the grounded-out version, the Hugin architecture is advantageous. We would not calculate 35 products with 35 multiplicands but multiply each incoming message into the stored factor once (leading to overall 35 multiplications) and divide the stored factor by the received message if sending the return message (35 divisions). Using Hugin in our lifted algorithm does not have a huge effect on the number of computations given the FO jtree for G_{ex2}.

6 Conclusion

Most applications need efficient inference algorithms. As first experiments with our lifted junction tree algorithm show, our proposal performs inference more efficiently when dealing with multiple queries in the presence of symmetries. The junction tree construction imposes overhead but only once per fixed knowledge base. If the evidence changes, message passing is repeated. For different queries for a model and given evidence, the algorithm only needs to run once.

Currently, we are fleshing out our algorithm and implementation to fully handle evidence. Additionally, we intend to optimize the basic implementation of the algorithm and extend it to include message passing based on the Hugin architecture. We plan to thoroughly evaluate different settings and analyze the behavior of our algorithm in terms of growing knowledge bases. In a broader scope, we investigate ideas to further use the data structures with respect to other theoretical constructs as well as practical applications that could benefit from our algorithm. Symmetries within a factor present an area of interest to potentially refine the data structures and increase efficiency. Dynamic structures to incorporate temporal constructs are another branch of work to look at.

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. Braun, T.: Lifted Junction Tree Algorithm. Technical Report, Universität zu Lübeck (2016)
3. de Salvo Braz, R.: Lifted First-Order Probabilistic Inference. PhD Thesis, University of Illinois at Urbana-Champaign (2007)
4. Van den Broeck, G.: Lifted Inference and Learning in Statistical Relational Models. PhD Thesis, KU Leuven (2013)
5. Chavira, M., Darwiche, A.: Compiling Bayesian Networks Using Variable Elimination. In: Proceedings of the 20th International Joint Conference on Artificial Intelligence. Morgan Kaufman, San Francisco (2007)
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. 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)
9. 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)
10. 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)
11. Milch, B., Li L.: Bayesian Logic Programming Language, `https://bayesianlogic.github.io`
12. 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)
13. Shafer, G.R., Shenoy, P.P.: Probability Propagation. In: Annals of Mathematics and Artificial Intelligence, vol. 2, pp. 327–351. Springer, Heidelberg (1989)
14. 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)
15. Singla, P., Nath, A., Domingos, P.: Approximate Lifted Belief Propagation. In: Working Note of the Workshop on Statistical Relational Artificial Intelligence at the 24th Conference on Artificial Intelligence. The AAAI Press, Menlo Park (2010)
16. Singla, P., Nath, A., Domingos, P.: Approximate Lifting Techniques for Belief Propagation. In: Proceedings of the 28th Conference on Artificial Intelligence, pp. 2497–2504. The AAAI Press, Menlo Park (2014)
17. Taghipour, N.: Lifted Probabilistic Inference by Variable Elimination. PhD Thesis, KU Leuven (2013)
18. Taghipour, N.: GC-FOVE, `https://dtai.cs.kuleuven.be/software/gcfove`
19. 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)