Adaptive Inference on Probabilistic Relational Models

Tanya Braun and Ralf Möller

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

Abstract. Standard approaches for inference in probabilistic relational models include lifted variable elimination (LVE) for single queries. To efficiently handle multiple queries, the lifted junction tree algorithm (LJT) uses a first-order cluster representation of a model, employing LVE as a subroutine in its steps. Adaptive inference concerns efficient inference under changes in a model. If the model changes, LJT restarts, possibly unnecessarily dumping information. The purpose of this paper is twofold, (i) to adapt the cluster representation to incremental changes, and (ii) to transform LJT into an adaptive version, enabling LJT to preserve as much computations as possible. Adaptive LJT fast reaches the point of answering queries again after changes, which is especially important for time-critical applications or online query answering.

1 Introduction

A common task in many applications is repeated inference on variations of a model. Variations range from conditioning on a new set of observed events to updating a probability distribution given observations or adapting a model structure while optimising a model representation. Applications include risk analysis where most likely explanations are of interest with changing sets of events coming in regularly [14]. When learning a model structure given data, one approach, called structural expectation-maximisation, alternates between minimally changing a model structure and updating distributions in a model to optimise the representation of the given data. The approach involves changing a model w.r.t. structure and distributions as well as repeated inference when computing the probability of the observed data in the altered model [11].

In a naive way, one incorporates the changes in a model or evidence and performs inference. Adaptive inference, however, aims at performing inference more efficiently when changes in a model or evidence occur. Research exists for adaptive inference on propositional models [10,1]. But, modelling realistic scenarios yields large probabilistic relational models, requiring exact and efficient reasoning about sets of individuals.

Research in the field of lifted inference has lead to efficient algorithms for relational models. Lifted variable elimination (LVE), first introduced in [16] and expanded in [17,13,20], saves computations by reusing intermediate results for isomorphic subproblems when answering a query. The lifted junction tree algorithm (LJT) sets up a first-order junction tree (FO jtree) to handle multiple queries efficiently [4] using LVE as a subroutine. Van den Broeck et al. apply lifting to weighted model counting and knowledge compilation [8], with newer work on asymmetrical models [7]. To scale lifting, Das et al. use graph databases storing compiled models to count faster [9]. Lifted belief propagation (BP) provides approximate solutions to queries, often using lifted representations, e.g. [2]. But, to the best of our knowledge, research for adaptive inference on relational models is limited. In relational models, changes can also affect the sets of individuals over which one reasons or on which one conditions on. How to handle such incremental changes correctly and efficiently is not obvious.

Nath and Domingos as well as Ahmadi et al. provide approximate algorithms based on BP for lifted, adaptive inference for changing evidence [15,3]. They reuse results from previous algorithm runs and propagate messages only in affected regions or adapt their lifted representations to the changed evidence. We focus on *exact* inference for multiple queries and present an efficient algorithm for *adaptive inference* based on LJT, called aLJT, handling changes in model and evidence. This paper includes two main contributions, (i) procedures for adapting an FO jtree to incremental changes for its underlying model and (ii) an algorithm, aLJT, preserving as much computations as possible under changes in a model. aLJT handles changes ranging from new evidence to extending a model with new factors. aLJT fast reaches the point of answering queries again, which is especially important for time-critical or online query answering.

The remainder of this paper is structured as follows: First, we introduce basic notations and recap LJT. Then, we show how to adapt an FO jtree to changes and present aLJT, followed by a discussion. We conclude with upcoming work.

2 Preliminaries

This section specifies notations and recaps LJT. Based on [17], a running example models the interplay of natural or man-made disasters, an epidemic, and people being sick, travelling, and being treated. Parameters represent disasters, people, and treatments.

2.1 Parameterised Probabilistic Models

Parameterised models compactly represent models by using logical variables (logvars) to parameterise randvars, abbreviated PRVs.

Definition 1. *Let $\mathbf{L}$, Φ, and $\mathbf{R}$ be sets of logvar, factor, and randvar names respectively. A* PRV *$R(L_1, \dots, L_n)$, $n \geq 0$, is a syntactical construct with $R \in \mathbf{R}$ and $L_1, \dots, L_n \in \mathbf{L}$ to represent a set of randvars. For PRV A, the term $range(A)$ denotes possible values. A logvar L has a domain $\mathcal{D}(L)$. 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 $C_{\mathbf{X}} \subseteq \times_{i=1}^{n} \mathcal{D}(X_i)$ restricting logvars to values. The symbol $\top$ marks that no restrictions apply and may be omitted. For some P, the term $lv(P)$ refers to its logvars, the term $rv(P)$ to its PRVs with constraints, and the term $gr(P)$ to all instances of P, i.e. P grounded w.r.t. constraints.*

For the epidemic scenario, we build the boolean PRVs $Epid$, $Sick(X)$, and $Travel(X)$ from $\mathbf{R} = \{Epid, Sick, Travel\}$ and $\mathbf{L} = \{X\}$, $\mathcal{D}(X) = \{alice, eve, bob\}$. $Epid$ holds if an epidemic occurs. $Sick(X)$ holds if a person X is sick, $Travel(X)$ holds if X travels. With $C = (X, \{eve, bob\})$, $gr(Sick(X)_{|C}) = \{Sick(eve), Sick(bob)\}$. $gr(Sick(X)_{|\top})$ also contains $Sick(alice)$. Parametric factors (parfactors) combine PRVs. A parfactor describes a function, identical for all argument groundings, mapping argument values to real values (potentials), of which at least one is non-zero.

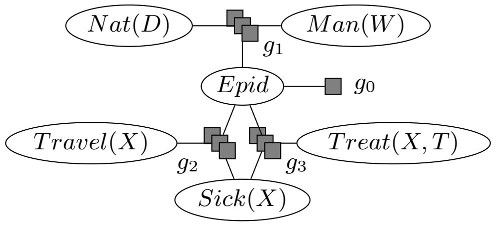
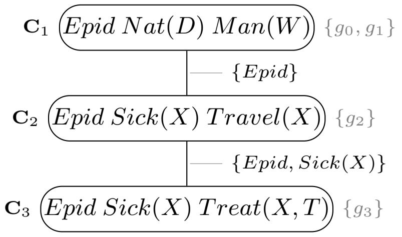


Fig. 1: Parfactor graph for G_{ex}



$\mathbf{C}_1$ $Epid\ Nat(D)\ Man(W)$ $\{g_0, g_1\}$
$\{Epid\}$
$\mathbf{C}_2$ $Epid\ Sick(X)\ Travel(X)$ $\{g_2\}$
$\{Epid, Sick(X)\}$
$\mathbf{C}_3$ $Epid\ Sick(X)\ Treat(X,T)$ $\{g_3\}$

Fig. 2: FO jtree for G_{ex}

Definition 2. *Let* $\mathbf{X} \subseteq \mathbf{L}$ *be a set of logvars,* $\mathcal{A} = (A_1, \dots, A_n)$ *a sequence of PRVs, built from* $\mathbf{R}$ *and* $\mathbf{X}$*,* C *a constraint on* $\mathbf{X}$*, and* $\phi : \times_{i=1}^{n} range(A_i) \mapsto \mathbb{R}^+$ *a function with name* $\phi \in \Phi$*, identical for all* $gr(\mathcal{A}_{|C})$*. We denote a* parfactor g *by* $\forall \mathbf{X} : \phi(\mathcal{A})_{|C}$*. We omit* $(\forall \mathbf{X} :)$ *if* $\mathbf{X} = lv(\mathcal{A})$ *and* $|\top$*. A set of parfactors forms a* model $G := \{g_i\}_{i=1}^{n}$*.*

We define a model G_{ex} as our running example. Let $\mathbf{L} = \{D, W, M, X\}$, $\Phi = \{\phi_0, \phi_1, \phi_2, \phi_3\}$, and $\mathbf{R} = \{Epid, Nat, Man, Sick, Travel, Treat\}$. We build three more boolean PRVs. $Nat(D)$ holds if a natural disaster D occurs, $Man(W)$ if a man-made disaster W occurs. $Treat(X, T)$ holds if a person X is treated with treatment T. The other domains are $\mathcal{D}(D) = \{earthquake, flood\}$, $\mathcal{D}(W) = \{virus, war\}$, and $\mathcal{D}(T) = \{vaccine, tablet\}$. The model reads $G_{ex} = \{g_i\}_{i=0}^{3}$, $g_0 = \phi_0(Epid)$, $g_1 = \phi_1(Epid, Nat(D), Man(W))_{|\top}$, $g_2 = \phi_2(Epid, Sick(X), Travel(X))_{|\top}$, and $g_3 = \phi_3(Epid, Sick(X), Treat(X, T))_{|\top}$. Parfactors g_1 to g_3 have eight input-output pairs, g_0 has two (omitted here). Figure 1 depicts G_{ex} as a graph with six variable nodes for the PRVs and four factor nodes for the parfactors with edges to arguments.

Evidence displays symmetries if observing the same value for n instances of a PRV [20]. In a parfactor $g_E = \phi_E(P(\mathbf{X}))_{|C_E}$, a potential function ϕ_E and constraint C_E encode the observed values and instances for PRV $P(\mathbf{X})$. Assume we observe the value $true$ for ten randvars of the PRV $Sick(X)$. The corresponding parfactor is $\phi_E(Sick(X))_{|C_E}$. C_E represents the domain of X restricted to the 10 instances and $\phi_E(true) = 1$ and $\phi_E(false) = 0$. A technical remark: To *absorb* evidence, we split all parfactors g_i that cover $P(X)$, called shattering [17], restricting C_i to those tuples that contain $gr(P(X)_{|C_E})$ and a duplicate of g_i to the rest. g_i absorbs g_E (cf. [20]).

The *semantics* of a model G is given by grounding and building a full joint distribution P_G. With Z as the normalisation constant, G represents $P_G = \frac{1}{Z} \prod_{f \in gr(G)} f$. The query answering (QA) problem asks for a marginal distribution of a set of randvars or a conditional distribution given events, which boils down to computing marginals w.r.t. a model's joint distribution, eliminating non-query terms. Formally, $P(\mathbf{Q}|\mathbf{E})$ denotes a query with $\mathbf{Q}$ a set of grounded PRVs and $\mathbf{E} = \{E_i = e_i\}_{i=1}^{n}$ a set of events. An example query for G_{ex} is $P(Epid|Sick(eve) = true)$. Next, we look at LJT, a lifted QA algorithm, which seeks to avoid grounding and building a full joint distribution.

2.2 Lifted Junction Tree Algorithm

LJT answers queries for probability distributions. It uses an FO jtree to efficiently answer a set of queries, with LVE as a subroutine. We briefly recap LJT.

LJT answers a set of queries $\{\mathbf{Q}_i\}_{i=1}^m$ given a model G and evidence $\mathbf{E}$. The main workflow is: (i) Construct an FO jtree J for G. (ii) Enter $\mathbf{E}$ into J. (iii) Pass messages in J. (iv) Compute answers for $\{\mathbf{Q}_i\}_{i=1}^m$. LJT first constructs a minimal FO jtree with parameterised clusters (parclusters) as nodes, which are sets of PRVs connected by parfactors, both defined as follows.

Definition 3. *Let* $\mathbf{X}$ *be a set of logvars,* $\mathbf{A}$ *a set of PRVs with* $lv(\mathbf{A}) \subseteq \mathbf{X}$*, and* C *a constraint on* $\mathbf{X}$*. Then,* $\forall \mathbf{X}{:}\mathbf{A}_{|C}$ *denotes a* parcluster. *We omit* $(\forall \mathbf{X}{:})$ *if* $\mathbf{X} = lv(\mathbf{A})$ *and* $|\top$*. An* FO jtree *for a model* G *is a cycle-free graph* $J = (V, E)$*, where* V *is the set of nodes, i.e., parclusters, and* E *the set of edges.* J *must satisfy three properties: (i)* $\forall \mathbf{C}_i \in V$*:* $\mathbf{C}_i \subseteq rv(G)$*. (ii)* $\forall g \in G$*:* $\exists \mathbf{C}_i \in V$ *s.t.* $rv(g) \subseteq \mathbf{C}_i$*. (iii) If* $\exists A \in rv(G)$ *s.t.* $A \in \mathbf{C}_i \wedge A \in \mathbf{C}_j$*, then* $\forall \mathbf{C}_k$ *on the path between* $\mathbf{C}_i$ *and* $\mathbf{C}_j$*:* $A \in \mathbf{C}_k$ *(running intersection property). An FO jtree is* minimal *if by removing a PRV from any parcluster, the FO jtree ceases to be an FO jtree, i.e., it no longer fulfils at least one of the three properties. The parameterised set* $\mathbf{S}_{ij}$*, called* separator *of edge* $\{i, j\} \in E$*, is defined by* $\mathbf{C}_i \cap \mathbf{C}_j$*. The term* $nbs(i)$ *refers to the neighbours of node i, defined as* $\{j | \{i, j\} \in E\}$*. Each* $\mathbf{C}_i \in V$ *has a* local model G_i *and* $\forall g \in G_i$*:* $rv(g) \subseteq \mathbf{C}_i$*. The* G_i*'s partition* G*.*

In a minimal FO jtree, no parcluster is a subset of another parcluster. Figure 2 shows a minimal FO jtree for G_{ex} with parclusters $\mathbf{C}_1 = \{Epid, Nat(D), Man(W)\}$, $\mathbf{C}_2 = \{Epid, Sick(X), Travel(X)\}$, and $\mathbf{C}_3 = \{Epid, Sick(X), Treat(X, T)\}$. $\mathbf{S}_{12} = \{Epid\}$ and $\mathbf{S}_{23} = \{Epid, Sick(X)\}$ are the separators. Parfactor g_0 appears at $\mathbf{C}_1$ but could be in any local model as $rv(g_0) = \{Epid\} \subset \mathbf{C}_i \; \forall \, i \in \{1, 2, 3\}$.

During construction, LJT assigns the parfactors in G to local models (cf. [4]). LJT enters $\mathbf{E}$ into each parcluster $\mathbf{C}_i$ where $rv(\mathbf{E}) \subseteq \mathbf{C}_i$. Local model G_i at $\mathbf{C}_i$ absorbs $\mathbf{E}$ as described above. Message passing distributes local information within the FO jtree. Two passes from the periphery to the center and back suffice [12]. If a node has received messages from all neighbours but one, it sends a message to the remaining neighbour (*inward* pass). In the *outward* pass, messages flow in the opposite direction. Formally, a *message* m_{ij} from node i to node j is a set of parfactors, with arguments from $\mathbf{S}_{ij}$. LJT computes m_{ij} by eliminating $\mathbf{C}_i \setminus \mathbf{S}_{ij}$ from G_i and the messages of all other neighbours with LVE. A minimal FO jtree enhances the efficiency of message passing. Otherwise, messages unnecessarily copy information between parclusters. To answer a query $\mathbf{Q}_i$, LJT finds a subtree J' covering $\mathbf{Q}_i$, compiles a submodel G' of local models in J' and messages from outside J', and sums out all non-query terms in G' using LVE.

Currently, LJT partially handles *adaptive inference*. LJT assumes a constant G for which it builds an FO jtree J, reusing J for varying $\mathbf{E}$ and $\mathbf{Q}$. If G or $\mathbf{E}$ change, LJT restarts with construction or evidence entering. However, changes do not necessarily mean a completely new model or evidence set. LJT may preserve J, local models, or messages in parts. Before presenting aLJT, we show how to adapt an FO jtree.

3 Adapting an FO Jtree to Model Changes

Changes may yield a structure change in a model G, which may cause a structure change in an FO jtree J. All actions towards adapting J need to ensure that J continues to be a minimal FO jtree and that local models still partition G. This section looks at adding, deleting, or replacing a parfactor and ends with an example.

Algorithm 1 Adapting an FO jtree $J = (V, E)$

```
procedure ADD(FO jtree $J$, parfactor $g'$)
    Let $\mathbf{A}^{old}$ known, $\mathbf{A}^{new}$ new PRVs in $g'$
    ADJUST($J$, $\mathbf{A}^{old}$) to get $\mathbf{C}_i$ with $\mathbf{A}^{old} \subseteq \mathbf{C}_i$
    if $\mathbf{A}^{new} = \emptyset$ then
        $G_i \leftarrow G_i \cup \{g'\}$, mark $\mathbf{C}_i$
    else if $\mathbf{A}^{old} = \mathbf{C}_i$ then
        $\mathbf{C}_i \leftarrow \mathbf{C}_i \cup rv(g'), G_i \leftarrow G_i \cup \{g'\}$, mark $\mathbf{C}_i$
    else
        New $\mathbf{C}_k \leftarrow rv(g')$, $G_k \leftarrow \{g'\}$, mark $\mathbf{C}_k$
        Add $\{i, k\}$ to $E$

procedure DELETE(FO jtree $J$, parfactor $g$)
    Get $\mathbf{C}_i \in V$ where $g \in G_i$
    $G_i \leftarrow G_i \setminus \{g\}$
    MIN($J$, $\mathbf{C}_i$, $rv(g) \setminus rv(G_i)$), mark $\mathbf{C}_i$

procedure MIN(FO jtree $J$, node $\mathbf{C}_i$, PRVs $\mathbf{A}$)
    for PRV $A \in \mathbf{A}$ do
        if $\forall j, k \in nbs(i) : A \notin \mathbf{S}_{ij} \wedge A \notin \mathbf{S}_{ik}$ then
            $\mathbf{C}_i \leftarrow \mathbf{C}_i \setminus \{A\}$, mark $\mathbf{C}_i$
    if $\mathbf{C}_i$ marked $\wedge\ \exists j \in nbs(i) : \mathbf{C}_i \subseteq \mathbf{C}_j$ then
        MERGE($J$, $\mathbf{C}_i$, $\mathbf{C}_j$)

procedure ADJUST(FO jtree $J$, PRVs $\mathbf{A}$)
    Extract set of nodes $N$ s.t. $\mathbf{A} \subseteq rv(N)$
    while $|N| > 1$ do
        Get $\mathbf{C}_i, \mathbf{C}_j \in N$
        $P \leftarrow$ path betw. $i, j$ without $i, j$, mark $P$
        $\mathbf{C}' := \mathbf{C}_i, \mathbf{C}'' := \mathbf{C}_j, lst \leftarrow |P| - 1$
        MERGE($J$, $\mathbf{C}_i$, $\mathbf{C}_j$), remove $\mathbf{C}_j$ from $N$
        while $lst > 0$ do
            if $\exists k, l \in P : \mathbf{S}_{kl} \subseteq \mathbf{C}' \wedge \mathbf{S}_{kl} \subseteq P[lst]$
                        $\vee \mathbf{S}_{kl} \subseteq \mathbf{C}'' \wedge \mathbf{S}_{kl} \subseteq P[0]$ then
                Remove $\{k, l\}$ from $E$
                break
            $\mathbf{C}' := P[0], \mathbf{C}'' := P[lst]$
            MERGE($J$, $P[0]$, $P[lst]$), update $N$
            $P \leftarrow P[1 \dots lst - 1], lst \leftarrow |P| - 1$

procedure MERGE(FO jtree $J$, nodes $\mathbf{C}_i, \mathbf{C}_j$)
    $\mathbf{C}_i \leftarrow \mathbf{C}_i \cup \mathbf{C}_j$, $G_i \leftarrow G_i \cup G_j$
    Remove $\mathbf{C}_j$ from $V$
    for each $k \in nbs(j)$ do
        Remove $\{j, k\}$, add $\{i, k\}$, $k \neq i$, in $E$
```

Adding a parfactor g' to G requires adding g' to a local model to partition $G \cup \{g'\}$. Algorithm 1 includes pseudocode for adding g' to $J = (V, E)$. It contains marking instructions relevant for aLJT. We assume that g' contains at least one PRV from V to yield one FO jtree. If the arguments in g' appear in a parcluster $\mathbf{C}_i$, we add g' to G_i. But, if g' contains new PRVs $\mathbf{A}^{new}$ or if the old, known PRVs in g', $\mathbf{A}^{old} \leftarrow rv(g') \cap rv(V)$, do not appear in a single parcluster, there is no parcluster $\mathbf{C}_i$ s.t. $rv(g') \subseteq \mathbf{C}_i$. Thus, we adjust J until $\mathbf{A}^{old} \subseteq \mathbf{C}_i$ for some i and handle $\mathbf{A}^{new}$ appropriately.

Procedure ADJUST in Alg. 1 arranges that $\mathbf{A}^{old} \subseteq \mathbf{C}_i$ for some i in J. ADJUST finds a set of parclusters N that cover the PRVs in $\mathbf{A}^{old}$ and merges N into a single parcluster to fulfil $\mathbf{A}^{old} \subseteq \mathbf{C}_i$ by successively merging parclusters $\mathbf{C}_i, \mathbf{C}_j \in N$. Merging is a union of parclusters, local models, and neighbours. Since J is acyclic, there exists a unique path P from $\mathbf{C}_i$ to $\mathbf{C}_j$ without i and j, which forms a cycle if $|P| > 1$, which ADJUST resolves: It searches for a separator $\mathbf{S}_{kl}$ of two parclusters $\mathbf{C}_k, \mathbf{C}_l$ on P s.t.

$$\mathbf{S}_{kl} \subseteq \mathbf{C}' \wedge \mathbf{S}_{kl} \subseteq P[lst] \vee \mathbf{S}_{kl} \subseteq \mathbf{C}'' \wedge \mathbf{S}_{kl} \subseteq P[0] \tag{1}$$

where $\mathbf{C}'$ and $\mathbf{C}''$ are $\mathbf{C}_i$ and $\mathbf{C}_j$ in the beginning, i.e., information on $\mathbf{S}_{kl}$ reaches $\mathbf{C}_k$ from one end and $\mathbf{C}_l$ from the other end. If $\mathbf{S}_{kl}$ exists, ADJUST deletes the edge $\{k, l\}$ to break the cycle, which keeps the parclusters on P small. Otherwise, it continues along P, merging parclusters at the path ends if the search for a separator fulfilling Eq. (1) fails. For details, see Alg. 1.

After adjusting J, there is a parcluster $\mathbf{C}_i$ s.t. $\mathbf{A}^{old} \subseteq \mathbf{C}_i$. If g' contains only $\mathbf{A}^{old}$, procedure ADD adds g' to local model G_i at $\mathbf{C}_i$. If g' contains new PRVs, it distinguishes between $\mathbf{A}^{old} \subset \mathbf{C}_i$ and $\mathbf{A}^{old} = \mathbf{C}_i$. In the former case, PRVs in $\mathbf{C}_i$ do not

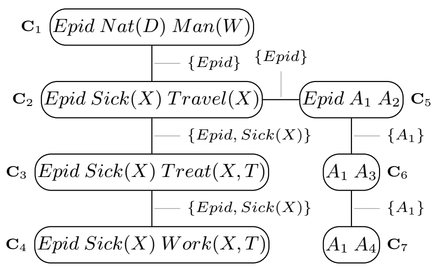

$\mathbf{C}_1$ $Epid\ Nat(D)\ Man(W)$
$\{Epid\}$
$\{Epid\}$
$\mathbf{C}_2$ $Epid\ Sick(X)\ Travel(X)$
$Epid\ A_1\ A_2$ $\mathbf{C}_5$
$\{Epid, Sick(X)\}$
$\{A_1\}$
$\mathbf{C}_3$ $Epid\ Sick(X)\ Treat(X,T)$
$A_1\ A_3$ $\mathbf{C}_6$
$\{Epid, Sick(X)\}$
$\{A_1\}$
$\mathbf{C}_4$ $Epid\ Sick(X)\ Work(X,T)$
$A_1\ A_4$ $\mathbf{C}_7$

Fig. 3: Adapted and Extended FO jtree

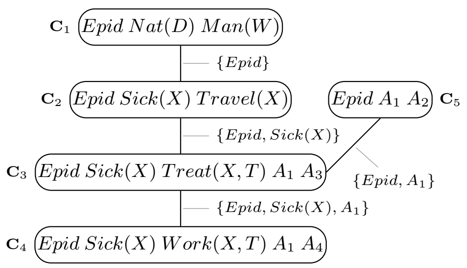

$\mathbf{C}_1$ $Epid\ Nat(D)\ Man(W)$
$\{Epid\}$
$\mathbf{C}_2$ $Epid\ Sick(X)\ Travel(X)$
$Epid\ A_1\ A_2$ $\mathbf{C}_5$
$\{Epid, Sick(X)\}$
$\mathbf{C}_3$ $Epid\ Sick(X)\ Treat(X,T)\ A_1\ A_3$
$\{Epid, A_1\}$
$\{Epid, Sick(X), A_1\}$
$\mathbf{C}_4$ $Epid\ Sick(X)\ Work(X,T)\ A_1\ A_4$

Fig. 4: Adjusted FO jtree

appear in $rv(g)$ and vice versa. ADD adds a new node $\mathbf{C}_k \leftarrow rv(g')$ with $G_k \leftarrow \{g'\}$ as a neighbour to i. In the latter case, $\mathbf{C}_i$ is a strict subset of the PRVs in g. ADD adds the new PRVs to $\mathbf{C}_i$ and g' to G_i. Now, the local models partition G'.

Deleting a parfactor g from G requires removing g from the local model G_i in which g appears. Afterwards, the local models partition $G \setminus \{g\}$. Algorithm 1 contains pseudocode for deleting g from J. After removing g from G_i, it minimises $\mathbf{C}_i$ w.r.t. $\mathbf{A}^{del} \leftarrow rv(g) \setminus rv(G_i)$. The procedure deletes a PRV $A \in \mathbf{A}^{del}$ from $\mathbf{C}_i$ if no two separators contain A, i.e., $\forall j,k \in nbs(i) : A \notin \mathbf{S}_{ij} \wedge A \notin \mathbf{S}_{ik}$. If now $\mathbf{C}_i \subseteq \mathbf{C}_j$ for a neighbour $\mathbf{C}_j$, MIN merges $\mathbf{C}_i$ and $\mathbf{C}_j$ to keep J minimal.

Replacing a parfactor g with a parfactor g' in G boils down to adding g' and then deleting g. If $rv(g) = rv(g')$, adding g' and deleting g does not touch J. If $rv(g) \subseteq rv(g')$, adding g' yields J', followed by deleting g from J', which does not change J'. First deleting g may lead to removing PRVs and superfluously merging parclusters. If $rv(g') \subseteq rv(g)$, adding g' before deleting g uses that there exists a parcluster $\mathbf{C}_i$ with $rv(g') \subseteq \mathbf{C}_i$ as $rv(g) \subseteq \mathbf{C}_i$. If the arguments of g and g' overlap otherwise, first adding g' and then deleting g avoids unnecessarily deleting PRVs and merging parclusters for the overlap PRVs. If both parfactors do not share any PRVs, replacing g with g' naturally decomposes into adding g' and deleting g.

To illustrate adaption, consider the FO jtree in Fig. 2. We add the parfactor $g_4 = \phi_4(Epid, Sick(X), Work(X))$ to G_{ex}, where PRV $Work(X)$ holds if a person X works. For g_4, the known PRVs are $Epid$ and $Sick(X)$ which appear in $\mathbf{C}_2$ and $\mathbf{C}_3$. Assume Alg. 1 chooses $\mathbf{C}_3$, which contains a PRV not in g_4, $Treat(X,T)$, while g_4 contains a new PRV, $Work(X)$. Thus, Alg. 1 adds a parcluster $\mathbf{C}_4 = \{Epid, Sick(X), Work(X)\}$, $G_4 = \{g_4\}$. The left column of parclusters in Fig. 3 shows the result .

Next, we replace g_2 with a parfactor $g_2' = \phi_2'(Travel(X), Sick(X))$ in G_{ex}, which means adding g_2' to $\mathbf{C}_2$ and deleting g_2. After removing g_2 from G_2, $Epid$ no longer appears in G_2. But, $Epid$ appears in both its separators and as such, has to remain in $\mathbf{C}_2$ to connect the appearance of $Epid$ from $\mathbf{C}_1$ to $\mathbf{C}_3$. If $g_2' = \phi_2'(Epid, Travel(X))$, Alg. 1 would delete $Sick(X)$ as $Sick(X)$ appears only in one separator. If $g_2' = \phi_2'(Epid, Sick(X))$, Alg. 1 would delete $Travel(X)$ and merge $\mathbf{C}_2$ with $\mathbf{C}_3$.

To illustrate adjusting an FO jtree, let the adapted FO jtree have three more parclusters with PRVs A_1, A_2, A_3, and A_4, shown in Fig. 3. We add a parfactor $g' = \phi'(A_4, Work(X))$. ADJUST merges $\mathbf{C}_4$ and $\mathbf{C}_7$ into $\mathbf{C}_4'$, causing a cycle. $P[0]$ and $P[lst]$ are $\mathbf{C}_3$ and $\mathbf{C}_6$, i.e., the neighbours of 4 and 7 on the cycle/path. No separator appears in $\mathbf{C}_4$ and $\mathbf{C}_6$ or $\mathbf{C}_7$ and $\mathbf{C}_3$ (Eq. (1) not fulfilled). ADJUST merges $\mathbf{C}_3$ and $\mathbf{C}_6$ into $\mathbf{C}'_3$. Separator $\mathbf{S}_{25} = \{Epid\}$ appears in $\mathbf{C}'_3$ and $\mathbf{C}_5$. ADJUST deletes edge $\{2, 5\}$, forming an acyclic FO jtree as seen in Fig. 4. At $\mathbf{C}'_4$, Alg. 1 adds g' to the local model.

4 LJT for Adaptive Inference

The extended algorithm aLJT performs adaptive inference for more efficient QA than by restarting from scratch. aLJT basically still consists of the steps construction, evidence entering, and message passing before it answers queries. Each step proceeds in an adaptive manner w.r.t. changes in input model G or in evidence $\mathbf{E}$ given an FO jtree J. Without an FO jtree, the steps are identical to the LJT steps.

Algorithm 2 shows a description of aLJT for J, referring to the changes in G and $\mathbf{E}$ by Δ_G and $\Delta_{\mathbf{E}}$. Line 1 contains the adaptive construction step, which adapts J to Δ_G according to Alg. 1. To track changes, aLJT marks a parcluster $\mathbf{C}_i$ if a local model changes s.t. the messages become invalid. Based on the marks and $\Delta_{\mathbf{E}}$, aLJT performs adaptive evidence entering and message passing, answering queries as before. Lines 2 to 4 show adaptive evidence entering and lines 5 to 9 adaptive message passing. Lines 10 to 12 contain the steps to answer a query $\mathbf{Q}_i$ from a set of queries $\{\mathbf{Q}_i\}_{i=1}^m$, as in LJT. Next, we look at the adaptive steps, followed by an example.

Construction: aLJT handles changes Δ_G as in Alg. 1 with J as input and Δ_G referring to parfactors to add, delete, or replace. When adding a parfactor g, aLJT marks the parcluster $\mathbf{C}_i$ that receives g. If adjusting J for known PRVs, aLJT marks all parclusters on the cycle between two parclusters $\mathbf{C}_i, \mathbf{C}_j$ that it merges. The merged parcluster $\mathbf{C}'_i$ has two messages m_{xi}, m_{yj} from its neighbours on the cycle with both information about the parclusters on the cycle and with information from G_i (in m_{yj}) and G_j (in m_{xi}), which is already contained in $G'_i \leftarrow G_i \cup G_j$. A similar situation occurs for all cycle parclusters, requiring new messages. Merging adjacent parclusters does not require a mark since messages between them are no longer considered and all other messages remain valid. When deleting a parfactor from the local model of $\mathbf{C}_i$, aLJT marks $\mathbf{C}_i$. aLJT replaces a parfactor by adding and deleting, which includes marks.

For changes in potentials, ranges, or constraints, aLJT replaces parfactors. For domain changes of a logvar X, aLJT marks a parcluster $\mathbf{C}_i$ if $X \in lv(\mathbf{C}_i)$ and its constraint w.r.t. X is $\top$. After incorporating all changes, parclusters are properly marked.

Evidence Entering: Adaptive entering deals with evidence at marked parclusters and changes $\Delta_{\mathbf{E}}$ in evidence. In the first case, marked parcluster only need evidence entering if new parfactors or domain changes affect it. If evidence does not change, only new parfactors or parfactors affected by domain changes need to absorb evidence.

In the second case, aLJT enters evidence at all parclusters $\mathbf{C}_i$ affected by $\Delta_{\mathbf{E}}$, which refers to changes in the form of additional or retracted evidence or new observed values. For additional evidence, aLJT uses the current local model G_i and enters the additional evidence. For retracted evidence, aLJT resets parfactors where the evidence no longer appears, which may require reentering evidence if evidence for a PRV is partially retracted. For new values, aLJT resets parfactors that have absorbed the original evidence. These parfactors absorb the new values. If $\Delta_{\mathbf{E}}$ leads to changes in G_i, aLJT marks $\mathbf{C}_i$.

Algorithm 2 LJT for adaptive inference answering queries $\{\mathbf{Q}_i\}_{i=1}^m$ given an FO jtree J and changes Δ_G for model G and $\Delta_\mathbf{E}$ for evidence $\mathbf{E}$

```
 1: Adapt $J$ to $\Delta_G$ according to Alg. 1                    ▷ marks parclusters
 2: for each parcluster $\mathbf{C}_i$ in $J$ do
 3:     if $\mathbf{C}_i$ marked or affected by $\Delta_\mathbf{E}$ then
 4:         Handle evidence at $\mathbf{C}_i$, mark $\mathbf{C}_i$
 5: while $\exists\ \mathbf{C}_i$ ready to send message $m_{ij}$ to $\mathbf{C}_j$ in $J$ do
 6:     if $\mathbf{C}_i$ marked or has marked message then
 7:         Send newly computed $m_{ij}$, mark $m_{ij}$ at $\mathbf{C}_j$ as changed
 8:     else
 9:         Send empty message, mark $m_{ij}$ at $\mathbf{C}_j$ as unchanged
10: for each query $\mathbf{Q}_i$ do
11:     Extract submodel $G'$ from subtree $J'$ that covers $\mathbf{Q}_i$
12:     Answer $\mathbf{Q}_i$ on $G'$ using LVE
```

Message Passing: aLJT maintains the same two-pass scheme starting at the periphery going inward and returning to the periphery outward. Inward, if a parcluster has received messages from all neighbours but one, it sends a message to the remaining neighbour. Outward, after a parcluster has received a message from the remaining neighbour, it sends messages to all other neighbours. The scheme preserves the ability for an automatic execution. After message passing, aLJT starts answering queries.

The adaptive part occurs during message calculation. A parcluster $\mathbf{C}_i$ calculates a new message if messages have become invalid during adjusting or if $\mathbf{C}_i$ has to distribute changes in its local model or received messages, else, it sends an empty message. The receiver replaces the old message with the new message and marks it changed (if not empty) or marks the old message as unchanged. Formally, $\mathbf{C}_i$ calculates a message m_{ij} for neighbour $\mathbf{C}_j$ if $\mathbf{C}_i$ itself is marked or if a message from a neighbour is marked as changed. Then, $\mathbf{C}_i$ computes m_{ij} using LVE with $G' \leftarrow G_i \cup \bigcup_{k \in nbs(i), k \neq j} m_{ki}$ as model (messages irregardless of whether they are marked changed) and $\mathbf{S}_{ij}$ as query.

As an example, consider the FO jtree in Fig. 4 with all its changes. All parclusters are marked except $\mathbf{C}_1$. Thus, the only empty message is m_{12}. After message passing, aLJT can answer queries for any randvar in $gr(rv(G))$. Next, assume we add evidence about $Nat(D)$ at $\mathbf{C}_1$, which leads aLJT to mark $\mathbf{C}_1$. With no further changes, aLJT only needs to distribute the updated information in G_1. Thus, messages m_{53} and m_{43} from $\mathbf{C}_5$ and $\mathbf{C}_4$ to $\mathbf{C}_3$ are empty as well as the messages from $\mathbf{C}_3$ over $\mathbf{C}_2$ to $\mathbf{C}_1$ as no change occurs in local models. Message m_{12} from $\mathbf{C}_1$ to $\mathbf{C}_2$ is new. The new message received by $\mathbf{C}_2$ leads to new messages from $\mathbf{C}_2$ to $\mathbf{C}_3$ and from $\mathbf{C}_3$ back to the leaf nodes $\mathbf{C}_4$ and $\mathbf{C}_5$. After sending all messages, aLJT can answer queries again.

Theoretical Discussion: aLJT and LJT have a *runtime complexity* linear in domain sizes, which also holds for other lifted algorithms [6,19]. The speedup comes in form of a factor as aLJT can avoid handling evidence for up to all parclusters and save calculating up to half of the messages after a change. Next, we argue why aLJT is sound.

Theorem 1. *aLJT is sound, i.e., computes a correct result for a query* $\mathbf{Q}$ *on an FO jtree* J *after adapting to changes in input model* G *and evidence* $\mathbf{E}$.

Proof sketch. We assume that LJT is correct, yielding an FO jtree J, fulfilling the FO jtree properties, which allows for local computations [18]. Further, we assume that LVE is correct, ensuring correct local computations during evidence entering, message passing, and query answering. aLJT first adapts J, which consists of adding, deleting and replacing parfactors. We briefly sketch how to prove that adapting J outputs an FO jtree again: We follow the changes in J showing that J remains an FO jtree. For the changes regarding adding, extending, or deleting a parcluster, it is straightforward to see that J' still fulfils the properties. The main part concerns the ADJUST procedure, which relies on J being acyclic and thus, causing at most one cycle between two parclusters. Breaking the cycle then ensures the FO jtree properties. Thus, adaptive construction outputs an FO jtree with marked parclusters. Adaptive evidence entering enters the new evidence version at all parclusters covering evidence and re-enters evidence at parclusters with changed local models, ensuring a correct evidence handling at all parclusters. Adaptive message passing distributes updated information whenever changed information arrives or local information has changed. With messages and local models updated, aLJT uses local models and messages to correctly answer **Q** using LVE. □

5 Empirical Evaluation

We have implemented prototypes of (a)LJT, named `ljt` and `aljt` here. Taghipour provides an LVE implementation (https://dtai.cs.kuleuven.be/software/lve), named `lve`. We fixed some lines in `lve` for queries with more than one grounded logvar. We do not include ground algorithms as we have already shown the speed-up by lifting (e.g., [5]).

The evaluation has two parts. First, we look at runtimes for G_{ex} under changes, focussing on how fast the programs provide answers again after consecutive changes. Second, we look at runtimes for the individual steps of LJT and aLJT for varying models G of sizes $|G|$ ranging from 2 to 1024 under a model change (adding a parfactor) and an evidence change (adding new evidence).

5.1 Consecutive Changes

This first part concerns three consecutive changes and two queries each. As input, we use G_{ex} with random potentials. We set $|\mathcal{D}(X)| = 1{,}000$ and $|\mathcal{D}(.)| = 100$ for the other logvars, yielding $|gr(G_{ex})| = 111{,}001$. Evidence occurs for 200 instances of $Sick(X)$ with the value $true$. There are two queries, $Sick(x_{1000})$ and $Treat(x_1, t_1)$. The consecutive changes for G_{ex}, based on the adaption examples, are (i) adding parfactor $\phi(Epid, Sick(X), Work(X))$ (referred to as model G^1_{ex}), (ii) replacing g_2 with parfactor $\phi(Sick(X), Travel(X))$ (referred to as model G^2_{ex}), and (iii) adding as evidence $Work(X) = true$ for 100 instances of X (referred to as model G^3_{ex}). The X values are a subset of the X values in the $Sick(X)$ evidence. After each change, the programs answer both queries again. We compare runtimes for inference averaged over five runs. Runtimes for `ljt` and `aljt` include construction, evidence entering, message passing, and query answering. Runtimes for `lve` consist of query answering.

Figure 5 shows runtimes in seconds [s] accumulated over all four models for `lve` (square), `ljt` (triangle), and `aljt` (circle). The vertical lines indicate when the programs have answered both queries, after which `lve` and `ljt` proceed with the next

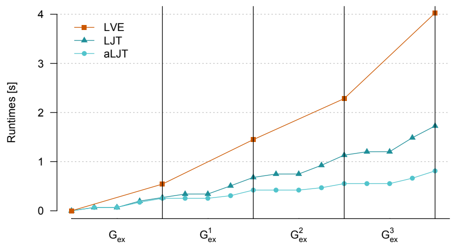


Fig. 5: Runtimes [s] accumulated over four models. Vertical lines mark the end of QA for the current model. Points on lines indicate the steps of (a)LJT.

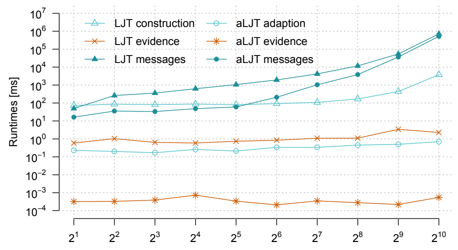


Fig. 6: Runtimes [ms] of the (a)LJT steps. X-axis: increasing $|G|$ from 2 to 1,024. Both axes appear on log scale. Points are connected for readability.

model, while `aljt` starts with adaption. For a model, the points on the `ljt` and `aljt` lines mark when an individual step is finished. `lve` takes longer than both LJT versions, showcasing the advantage of using an FO jtree. After only two queries, `ljt` and `aljt` have already offset their overhead and provide answers faster than `lve`.

For G_{ex}, `ljt` and `aljt` have the same runtimes since their runs are identical. As G_{ex} incrementally changes, `aljt` displays its advantage of adaptive steps in contrast to `ljt`. Starting with G_{ex}^1, `aljt` provides answers faster than `ljt`. Before `ljt` has completed message passing, `aljt` has already answered both queries. Especially message passing is faster as `aljt` does not need to compute half of the messages `ljt` computes. Construction is slightly faster. Evidence entering does not take long for both programs. But, evidence usually leads to longer runtimes for query answering compared to no evidence for LVE and LJT as the necessary splits lead to larger models. Since G_{ex}^3 contains more evidence, all runtimes increase compared to the previous models.

`aljt` fast reaches the point of answering queries again, providing answers more timely than the other two programs. As each change provides the possibility for `aljt` to save computations, leading to savings in runtime, the savings add up over a sequence of changes. Thus, performing adaptive inference pays off.

5.2 Step-wise Performance

This second part looks at runtimes of the individual steps of LJT and aLJT given models of varying size. The model sizes start at 2 and double until they reach $1{,}024$. The first model is $G_2 \cup G_3$ from the FO jtree of G_{ex}. The second model is G_{ex}. For the other models, we basically duplicated the current G, starting with G_{ex}, renamed the PRVs and logvars of the duplicate, and connected the original part with the copied part through a parfactor. The largest model has $1{,}024$ parfactors and logvars and $3,072$ PRVs, resulting in an FO jtree with 770 parclusters. The largest parcluster contains 256 PRVs. Technical remark: The maximum parcluster size is larger than need be due to the heuristic the construction is based on. The largest parcluster contains all PRVs without parameters, because the heuristic leads the (a)LJT implementations to handle all parfactors without logvars separately at the beginning, resulting in one large parcluster as the parameterless PRVs also appear in all other parts of the model.

The domain sizes for all logvars are set to $1{,}000$, leading to grounded model sizes, ranging from $1{,}001{,}000$ to $513{,}256{,}256$. A part of the model receives evidence for 50% of the instances of one PRV. We compare runtimes of the corresponding LJT and aLJT step for the following settings: (i) Add a parfactor with a new PRV. (ii) Enter new evidence to an unchanged model. (iii) Pass messages after changes in a model. Reentering known evidence after changes in a model and passing messages after changes in evidence have shown similar runtimes to settings (ii) and (iii).

Figure 6 shows runtimes in milliseconds [ms] of `ljt` and `aljt` averaged over five runs for the three settings. The triangles and crosses mark `ljt`, while the circles and stars mark `aljt`. The hollow marks refer to construction/adaption, the cross and star marks to evidence entering, and the filled marks to message passing. In all three settings, `aljt` is faster than `ljt` and both performing similar given larger models. The curves have a similar shape but are on a different level if domain sizes are different to $1{,}000$.

For construction (hollow marks), `aljt` is two to three orders of magnitude faster than `ljt` (0.0024 in average). For evidence entering (cross/star marks), the savings are even higher: `aljt` is faster than `ljt` by more than three orders of magnitude (0.0004 in average). Evidence handling appears to be constant in this setup. Since LVE has to perform one split per evidence PRV independent of the domain sizes and the evidence is restricted to one part of the model, evidence handling does not depend on the model size. Message passing (filled marks) shows only a clear speedup for smaller models. The first half of the models allows for `aljt` to be one order of magnitude faster than `ljt` (0.0955 in average). For the larger models, the factor of the speedup lays between 0.25 and 0.79. Concerning providing an answer to a query after a change, runtimes are basically a sum of the previous steps plus the time for answering a query, which takes around 100 ms. Since message passing dominates in the overall performance of (a)LJT with only one query, the overall runtimes resemble the runtimes of message passing.

Overall, `aljt` runtimes are faster by a factor ranging from 0.003 and 0.5 for such models. In the first two steps, aLJT is two orders of magnitude faster with changes in evidence and model restricted to certain parts of an FO jtree. Considering the first part of the evaluation, savings add up given frequent changes. In summary, performing adaptive inference pays off as `aljt` is able to provide a faster online QA than `ljt`.

6 Conclusion

We present aLJT, an adaptive version of LJT, which incorporates incremental changes in its input model or evidence efficiently. We specify how to adapt an FO jtree when deleting, adding, or replacing parts of a model. We formalise under which conditions evidence entering and new messages are necessary. Given the adaptive steps, aLJT reduces its static overhead for construction, evidence entering, and message passing under gradual changes compared to LJT. aLJT allows for fast online inference for answering multiple queries, minimising the lag in query answering when inputs change.

We currently work on learning lifted models, where we use aLJT as a subroutine. Other interesting algorithm extensions include parallelisation, construction using hypergraph partitioning, and different message passing strategies. Additionally, we look into areas of application to see its performance on real-life scenarios.

References

1. Acar, U.A., Ihler, A.T., Mettu, R.R., Sümer, Ö.: Adaptive Inference on General Graphical Models. In: UAI-08 Proc. of the 24th Conference on Uncertainty in AI. pp. 1–8 (2008)
2. 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)
3. Ahmadi, B., Kersting, K., Sanner, S.: Multi-evidence Lifted Message Passing, with Application to Pagerank and the Kalman Filter. In: IJCAI-11 Proc. of the 22nd International Joint Conference on AI. pp. 1152–1158 (2011)
4. Braun, T., Möller, R.: Lifted Junction Tree Algorithm. In: Proc. of KI 2016: Advances in AI. pp. 30–42. Springer (2016)
5. Braun, T., Möller, R.: Counting and Conjunctive Queries in the Lifted Junction Tree Algorithm - Extended Version. In: Postproc. of the 5th Internat. GKR Workshop. Springer (2018)
6. van den Broeck, G.: On the Completeness of First-order Knowledge Compilation for Lifted Probabilistic Inference. In: Advances in Neural Information Processing Systems 24. pp. 1386–1394 (2011)
7. van den Broeck, G., Niepert, M.: Lifted Probabilistic Inference for Asymmetric Graphical Models. In: AAAI-15 Proc. of the 29th Conference on AI. pp. 3599–3605 (2015)
8. van den Broeck, G., Taghipour, N., Meert, W., Davis, J., Raedt, L.D.: Lifted Probabilistic Inference by First-order Knowledge Compilation. In: IJCAI-11 Proc. of the 22nd International Joint Conference on AI (2011)
9. Das, M., Wu, Y., Khot, T., Kersting, K., Natarajan, S.: Scaling Lifted Probabilistic Inference and Learning Via Graph Databases. In: Proc. of the SIAM International Conference on Data Mining. pp. 738–746 (2016)
10. Delcher, A.L., Grove, A.J., Kasif, S., Pearl, J.: Logarithmic-time Updates and Queries in Probabilistic Networks. In: UAI-95 Proc. of the 11th Conference on Uncertainty in AI. pp. 116–124 (1995)
11. Friedman, M.: The Bayesian Structural EM Algorithm. In: UAI-98 Proc. of the 14th Conference on Uncertainty in AI. pp. 129–138 (1998)
12. 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)
13. Milch, B., Zettelmoyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted Probabilistic Inference with Counting Formulas. In: AAAI-08 Proc. of the 23rd Conference on AI. pp. 1062–1068 (2008)
14. Muñoz-González, L., Sgandurra, D., Barrère, M., Lupu, E.C.: Exact Inference Techniques for the Analysis of Bayesian Attack Graphs. IEEE Transactions on Dependable and Secure Computing **PP(99)**, 1–14 (2017)
15. Nath, A., Domingos, P.: Efficient Lifting for Online Probabilistic Inference. In: Proc. of the 24th AAAI Conference on AI (2010)
16. Poole, D.: First-order Probabilistic Inference. In: IJCAI-03 Proc. of the 18th International Joint Conference on AI (2003)
17. de Salvo Braz, R., Amir, E., Roth, D.: Lifted First-order Probabilistic Inference. In: IJCAI-05 Proc. of the 19th International Joint Conference on AI (2005)
18. Shenoy, P.P., Shafer, G.R.: Axioms for Probability and Belief-Function Propagation. Uncertainty in AI 4 **9**, 169–198 (1990)
19. Taghipour, N., Fierens, D., van den Broeck, G., Davis, J., Blockeel, H.: Completeness Results for Lifted Variable Elimination. In: Proc. of the 16th International Conference on AI and Statistics. pp. 572–580 (2013)
20. Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. Journal of AI Research **47(1)**, 393–439 (2013)