

Fusing First-order Knowledge Compilation and the Lifted Junction Tree Algorithm

Tanya Braun and Ralf Möller

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

Abstract. Standard approaches for inference in probabilistic formalisms with first-order constructs include lifted variable elimination (LVE) for single queries as well as first-order knowledge compilation (FOKC) based on weighted model counting. To handle multiple queries efficiently, the lifted junction tree algorithm (LJT) uses a first-order cluster representation of a model and LVE as a subroutine in its computations. For certain inputs, the implementation of LVE and, as a result, LJT ground parts of a model where FOKC runs without groundings. The purpose of this paper is to prepare LJT as a backbone for lifted query answering and to use any exact inference algorithm as subroutine. Fusing LJT and FOKC, by setting FOKC as a subroutine, allows us to compute answers faster than FOKC alone and LJT with LVE for certain inputs.

Keywords: Lifting · Probabilistic logical models · Variable elimination · Weighted model counting.

1 Introduction

AI areas such as natural language understanding and machine learning need efficient inference algorithms. Modeling realistic scenarios yields large probabilistic models, requiring reasoning about sets of individuals. Lifting uses symmetries in a model to speed up reasoning with known domain objects. We study probabilistic inference in large models that exhibit symmetries with queries for probability distributions of random variables (randvars).

In the last two decades, researchers have advanced probabilistic inference significantly. Propositional formalisms benefit from variable elimination (VE), which decomposes a model into subproblems and evaluates them in an efficient order [28]. Lifted VE (LVE), introduced in [21] and expanded in [22,19,25], saves computations by reusing intermediate results for isomorphic subproblems. Taghipour et al. formalise LVE by defining lifting operators while decoupling the constraint language from the operators [26]. 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. LJT is based on the propositional junction tree algorithm [18], which includes a junction tree (jtree) and a reasoning algorithm for efficient handling of multiple queries. Approximate lifted inference often uses lifting in conjunction with belief propagation [24,15,1]. To scale

lifting, Das et al. use graph databases storing compiled models to count faster [14]. Other areas incorporate lifting to enhance efficiency, e.g., in continuous or dynamic models [12,27], logic programming [3], and theorem proving [16].

Logical methods for probabilistic inference are often based on weighted model counting (WMC) [11]. Propositional knowledge compilation (KC) compiles a weighted model into a deterministic decomposable negation normal form (d-DNNF) circuit for probabilistic inference [13]. Chavira and Darwiche combine VE and KC as well as algebraic decision diagrams for local symmetries to further optimise inference runtimes [10]. Van den Broeck et al. apply lifting to KC and WMC, introducing weighted first-order model counting (WFOMC) and a first-order d-DNNF [9,7], with newer work on asymmetrical models [8].

For certain inputs, LVE, LJT, and FOKC start to struggle either due to model structure or size. The implementations of LVE and, as a consequence, LJT ground parts of a model if randvars of the form $Q(X), Q(Y), X \neq Y$ appear, where parameters X and Y have the same domain, even though in theory, LVE handles those occurrences of just-different randvars [2]. While FOKC does not ground in the presence of such constructs in general, it can struggle if the model size increases. The purpose of this paper is to prepare LJT as a backbone for lifted query answering (QA) to use any exact inference algorithm as a subroutine. Using FOKC and LVE as subroutines, we fuse LJT, LVE, and FOKC to compute answers faster than LJT, LVE, and FOKC alone for the inputs described above.

The remainder of this paper is structured as follows: First, we introduce notations and FO jtrees and recap LJT. Then, we present conditions for subroutines of LJT, discuss how LVE works in this context and FOKC as a candidate, before fusing LJT, LVE, and FOKC. We conclude with future work.

2 Preliminaries

This section introduces notations and recap LJT. We specify a version of the smokers example (e.g., [9]), where two friends are more likely to both smoke and smokers are more likely to have cancer or asthma. Parameters allow for representing people, avoiding explicit randvars for each individual.

Parameterised Models To compactly represent models with first-order constructs, parameterised models use logical variables (logvars) to parameterise randvars, abbreviated PRVs. They are based on work by Poole [20].

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 $\text{range}(A)$ 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 given values. The symbol \top marks that no restrictions apply and may be omitted. For some P , the term $\text{lv}(P)$ refers to its logvars, $\text{rv}(P)$ to its PRVs with constraints, and $\text{gr}(P)$ to all instances of P grounded w.r.t. its constraints.*

For the smoker example, let $\mathbf{L} = \{X, Y\}$ and $\mathbf{R} = \{Smokes, Friends\}$ to build boolean PRVs $Smokes(X)$, $Smokes(Y)$, and $Friends(X, Y)$. We denote $A = true$ by a and $A = false$ by $\neg a$. Both logvar domains are $\{alice, eve, bob\}$. An inequality $X \neq Y$ yields a constraint $C = ((X, Y), \{(alice, eve), (alice, bob), (eve, alice), (eve, bob), (bob, alice), (bob, eve)\})$. $gr(Friends(X, Y)|C)$ refers to all propositional randvars that result from replacing X, Y with the tuples in C . Parametric factors (parfactors) combine PRVs as arguments. A parfactor describes a function, identical for all argument groundings, that maps argument values to the reals (potentials), of which at least one is non-zero.

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

We define a model G_{ex} for the smoker example, adding the binary PRVs $Cancer(X)$ and $Asthma(X)$ to the ones above. The model reads $G_{ex} = \{g_i\}_{i=0}^5$, $g_0 = \phi_0(Friends(X, Y), Smokes(X), Smokes(Y))|C$, $g_1 = \phi_1(Friends(X, Y))|C$, $g_2 = \phi_2(Smokes(X))|\top$, $g_3 = \phi_3(Cancer(X))|\top$, $g_4 = \phi_4(Smokes(X), Asthma(X))|\top$, and $g_5 = \phi_5(Smokes(X), Cancer(X))|\top$. g_0 has eight, g_1 to g_3 have two, and g_4 and g_5 four input-output pairs (omitted here). Constraint C refers to the constraint given above. The other constraints are \top . Figure 1 depicts G_{ex} as a graph with five variable nodes and six factor nodes for the PRVs and parfactors with edges to arguments.

The *semantics* of a model G is given by grounding and building a full joint distribution. With Z as the normalisation constant, G represents the full joint probability distribution $P_G = \frac{1}{Z} \prod_{f \in gr(G)} f$. The QA problem asks for a likelihood of an event, a marginal distribution of some randvars, or a conditional distribution given events, all queries boiling down to computing marginals w.r.t. a model's joint distribution. Formally, $P(\mathbf{Q}|\mathbf{E})$ denotes a (conjunctive) query with \mathbf{Q} a set of grounded PRVs and $\mathbf{E} = \{E_k = e_k\}_k$ a set of events (grounded PRVs with range values). If $\mathbf{E} = \emptyset$, the query is for a conditional distribution. A query for G_{ex} is $P(Cancer(eve)|friends(eve, bob), smokes(bob))$. We call $\mathbf{Q} = \{Q\}$ a singleton query. Lifted QA algorithms seek to avoid grounding and building a full joint distribution. Before looking at lifted QA, we introduce FO jtrees.

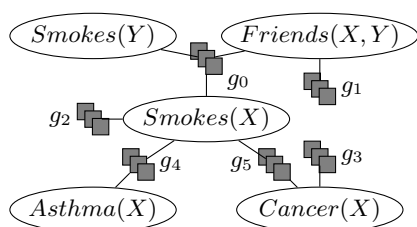


Fig. 1. Parfactor graph for G_{ex}

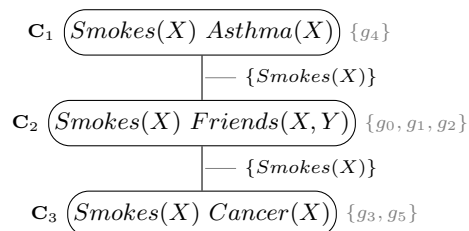


Fig. 2. FO jtree for G_{ex} (local models in grey)

Algorithm 1 Outline of the Lifted Junction Tree Algorithm

```

procedure LJT(Model  $G$ , Queries  $\{\mathbf{Q}_j\}_{j=1}^m$ , Evidence  $\mathbf{E}$ )
  Construct FO jtree  $J$  for  $G$ 
  Enter  $\mathbf{E}$  into  $J$ 
  Pass messages on  $J$ 
  for each query  $\mathbf{Q}_j$  do
    Find subtree  $J'$  for  $\mathbf{Q}_j$ 
    Extract submodel  $G'$  of local models in  $J'$  and outside messages into  $J'$ 
    Answer  $\mathbf{Q}_j$  on  $G'$ 

```

First-order Junction Trees LJT builds an FO jtree to cluster a model into submodels that contain all information for a query after propagating information. An FO jtree, defined as follows, constitutes a lifted version of a jtree. Its nodes are parameterised clusters (parclusters), i.e., sets of PRVs connected by parfactors.

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 $(\forall \mathbf{X}:\cdot)$ if $\mathbf{X} = lv(\mathbf{A})$. An FO jtree for a model G is a cycle-free graph $J = (V, E)$, where V is the set of nodes (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 $\mathbf{C}_j: A \in \mathbf{C}_k$. 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 . 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 .

Figure 2 shows an FO jtree for G_{ex} with the following parclusters, $\mathbf{C}_1 = \forall X: \{Smokes(X), Asthma(X)\}|\top$, $\mathbf{C}_2 = \forall X, Y: \{Smokes(X), Friends(X, Y)\}|C$, and $\mathbf{C}_3 = \forall X: \{Smokes(X), Cancer(X)\}|\top$. Separators are $\mathbf{S}_{12} = \mathbf{S}_{23} = \{Smokes(X)\}$. As $Smokes(X)$ and $Smokes(Y)$ model the same randvars, \mathbf{C}_2 names only one. Parfactor g_2 appears at \mathbf{C}_2 but could be in any local model as $rv(g_2) = \{Smokes(X)\} \subset \mathbf{C}_i \forall i \in \{1, 2, 3\}$. [4] details building FO jtrees.

Lifted Junction Tree Algorithm LJT answers a set of queries efficiently by answering queries on smaller submodels. Algorithm 1 outlines LJT for a set of queries (cf. [4] for details). LJT starts with constructing an FO jtree. It enters evidence for a local model to absorb whenever the evidence randvars appear in a parcluster. Message passing propagates local information through the FO jtree in two passes: LJT sends messages from the periphery towards the center and then back. A message is a set of parfactors over separator PRVs. For a message m_{ij} from node i to neighbour j , LJT eliminates all PRVs not in separator \mathbf{S}_{ij} from G_i and the messages from other neighbours using LVE. Afterwards, each parcluster holds all information of the model in its local model and received messages. LJT answers a query by finding a subtree whose parclusters cover the query randvars, extracting a submodel of local models and outside messages, and answering the query on the submodel. In the original LJT, LJT eliminates randvars for messages and queries using LVE.

3 LJT as a Backbone for Lifted Inference

LJT provides general steps for efficient QA given a set of queries. It constructs an FO jtree and uses a subroutine to propagate information and answer queries. To ensure a lifted algorithm run without groundings, evidence entering and message passing impose some requirements on the algorithm used as a subroutine. After presenting those requirements, we analyse how LVE matches the requirements and to what extent FOKC can provide the same service.

Requirements LJT has a domain-lifted complexity, meaning that if a model allows for computing a solution without grounding part of a model, LJT is able to compute the solution without groundings, i.e., has a complexity linear in the domain size of the logvars. Given a model that allows for computing solutions without grounding part of a model, the subroutine must be able to handle message passing and query answering without grounding to maintain the domain-lifted complexity of LJT.

Evidence displays symmetries if observing the same value for n instances of a PRV [26]. Thus, for evidence handling, the algorithm needs to be able to handle a set of observations for some instances of a single PRV in a lifted way. Calculating messages entails that the algorithm is able to calculate a form of parameterised, conjunctive query over the PRVs in the separator. In summary, LJT requires the following:

1. Given evidence in the form of a set of observations for some instances of a single PRV, the subroutine must be able to absorb the evidence independent of the size of the set.
2. Given a parcluster with its local model, messages, and a separator, the subroutine must be able to eliminate all PRVs in the parcluster that do not appear in the separator in a domain-lifted way.

The subroutine also establishes which kind of queries LJT can answer. The expressiveness of the query language for LJT follows from the expressiveness of the inference algorithm used. If an algorithm answers queries of single randvar, LJT answers this type of query. If an algorithm answers maximum a posteriori (MAP) queries, the most likely assignment to a set of randvars, LJT answers MAP queries. Next, we look at how LVE fits into LJT.

Lifted Variable Elimination First, we take a closer look at LVE before analysing it w.r.t. the requirements of LJT. To answer a query, LVE eliminates all non-query randvars. In the process, it computes VE for one case and exponentiates its result for isomorphic instances (lifted summing out). Taghipour implements LVE through an operator suite (see [26] for details). Algorithm 2 shows an outline. All operators have pre- and postconditions to ensure computing a result equivalent to one for $gr(G)$. Its main operator *sum-out* realises lifted summing out. An operator *absorb* handles evidence in a lifted way. The remaining operators (*count-convert*, *split*, *expand*, *count-normalise*, *multiply*, *ground-logvar*) aim at enabling lifted summing out, transforming part of a model.

Algorithm 2 Outlines of Lifted QA Algorithms

```

function LVE(Model  $G$ , Query  $\mathbf{Q}$ , Evidence  $\mathbf{E}$ )
  Absorb  $\mathbf{E}$  in  $G$ 
  while  $G$  has non-query PRVs do
    if PRV  $A$  fulfils sum-out preconditions then
      Eliminate  $A$  using sum-out
    else
      Apply transformer
  return Multiply parfactors in  $G$  ▷  $\alpha$ -normalise

```

```

procedure FOKC(Model  $G$ , Queries  $\{Q_j\}_{j=1}^m$ , Evidence  $\mathbf{E}$ )
  Reduce  $G$  to WFOMC problem with  $\Delta, w_T, w_F$ 
  Compile a circuit  $\mathcal{C}_e$  for  $\Delta, \mathbf{E}$ 
  for each query  $Q_j$  do
    Compile a circuit  $\mathcal{C}_{q_e}$  for  $\Delta, Q_j, \mathbf{E}$ 
    Compute  $P(Q_j|\mathbf{E})$  through WFOMCs in  $\mathcal{C}_{q_e}, \mathcal{C}_e$ 

```

LVE as a subroutine provides lifted absorption for evidence handling. Lifted absorption splits a parfactor into one part, for which evidence exists, and one part without evidence. The part with evidence then absorbs the evidence by absorbing it once and exponentiating the result for all isomorphic instances. For messages, a relaxed QA routine computes answers to parameterised queries without making all instances of query logvars explicit. LVE answers queries for a likelihood of an event, a marginal distribution of a set of randvars, and a conditional distribution of a set of randvars given events. LJT with LVE as a subroutine answers the same queries. Extensions to LJT or LVE enable even more query types, such as queries for a most probable explanation or MAP [5].

First-order Knowledge Compilation FOKC aims at solving a WFOMC problem by building FO d-DNNF circuits given a query and evidence and computing WFOMCs on the circuits. Of course, different compilation flavours exist, e.g., compiling into a low-level language [17]. But, we focus on the basic version of FOKC. We briefly take a look at WFOMC problems, FO d-DNNF circuits, and QA with FOKC, before analysing FOKC w.r.t. the LJT requirements. See [9] for details.

Let Δ be a theory of constrained clauses and w_T a positive and w_F a negative weight function. Clauses follow standard notations of (function-free) first-order logic. A constraint expresses, e.g., an (in)equality of two logvars. w_T and w_F assign weights to predicates in Δ . A *WFOMC problem* consists of computing

$$\sum_{I \models \Delta} \prod_{a \in I} w_T(pred(a)) \prod_{a \in HB(T) \setminus I} w_F(pred(a))$$

where I is an interpretation of Δ that satisfies Δ , $HB(T)$ is the Herbrand base and $pred$ maps atoms to their predicate. See [6] for a description of how to transform parfactor models into WFOMC problems.

FOKC converts Δ to be in FO d-DNNF, where all conjunctions are decomposable (all pairs of conjuncts independent) and all disjunctions are deterministic (only one disjunct true at a time). The normal form allows for efficient reasoning as computing the probability of a conjunction decomposes into a product of the probabilities of its conjuncts and computing the probability of a disjunction follows from the sum of probabilities of its disjuncts. An *FO d-DNNF circuit* represents such a theory as a directed acyclic graph. Inner nodes are labelled with \vee and \wedge . Additionally, set-disjunction and set-conjunction represent isomorphic parts in Δ . Leaf nodes contain atoms from Δ . The process of forming a circuit is called compilation.

Now, we look at how FOKC answers queries. Algorithm 2 shows an outline with input model G , a set of query randvars $\{Q_i\}_{i=1}^m$, and evidence \mathbf{E} . FOKC starts with transforming G into a WFOMC problem Δ with weight functions w_T and w_F . It compiles a circuit \mathcal{C}_e for Δ including \mathbf{E} . For each query Q_i , FOKC compiles a circuit \mathcal{C}_{q_e} for Δ including \mathbf{E} and Q_i . It then computes

$$P(Q_i|\mathbf{E}) = \frac{WFOMC(\mathcal{C}_{q_e}, w_T, w_F)}{WFOMC(\mathcal{C}_e, w_T, w_F)} \quad (1)$$

by propagating WFOMCs in \mathcal{C}_{q_e} and \mathcal{C}_e based on w_T and w_F . FOKC can reuse the denominator WFOMC for all Q_i .

Regarding the potential of FOKC as a subroutine for LJT, FOKC does not fulfil all requirements. FOKC can handle evidence through conditioning [7]. But, a lifted message passing is not possible in a domain-lifted and exact way without restrictions. FOKC answers queries for a likelihood of an event, a marginal distribution of a single randvar, and a conditional distribution for a single randvar given events. Inherently, conjunctive queries are only possible if the conjuncts are probabilistically independent [13], which is rarely the case for separators. Otherwise, FOKC has to invest more effort to take into account that the probabilities overlap. Thus, the restricted query language means that LJT cannot use FOKC for message calculations in general. Given an FO jtree with singleton separators, message passing with FOKC as a subroutine may be possible. FOKC as such takes ground queries as input or computes answers for random groundings, so FOKC for message passing needs an extension to handle parameterised queries. FOKC may not fulfil all requirements, but we may combine LJT, LVE, and FOKC into one algorithm to answer queries for models where LJT with LVE as a subroutine struggles.

4 Fusing LJT, LVE, and FOKC

We now use LJT as a backbone and LVE and FOKC as subroutines, fusing all three algorithms. Algorithm 3 shows an outline of the fused algorithm named LJTKC. Inputs are a model G , a set of queries $\{Q_j\}_{j=1}^m$, and evidence \mathbf{E} . Each query Q_j has a single query term in contrast to a set of randvars \mathbf{Q}_j in LVE and LJT. The change stems from FOKC to ensure a correct result. As a consequence, LJTKC has the same expressiveness regarding the query language as FOKC.

Algorithm 3 Outline of LJTKC

```

procedure LJTKC(Model  $G$ , Queries  $\{Q_j\}_{j=1}^m$ , Evidence  $\mathbf{E}$ )
  Construct FO jtree  $J$  for  $G$ 
  Enter  $\mathbf{E}$  into  $J$ 
  Pass messages on  $J$  ▷ LVE as subroutine
  for each parcluster  $\mathbf{C}_i$  of  $J$  with local model  $G_i$  do
    Form submodel  $G' \leftarrow G_i \cup \bigcup_{j \in \text{nbrs}(i)} m_{ij}$ 
    Reduce  $G'$  to WFOMC problem with  $\Delta_i, w_T^i, w_F^i$ 
    Compile a circuit  $\mathcal{C}_i$  for  $\Delta_i$ 
    Compute  $c_i = \text{WFOMC}(\mathcal{C}_i, w_T^i, w_F^i)$ 
  for each query  $Q_j$  do
    Find parcluster  $\mathbf{C}_i$  where  $Q_j \in \mathbf{C}_i$ 
    Compile a circuit  $\mathcal{C}_q$  for  $\Delta_i, Q_j$ 
    Compute  $c_q = \text{WFOMC}(\mathcal{C}_q, w_T^i, w_F^i)$ 
    Compute  $P(Q_j|\mathbf{E}) = c_q/c_i$ 

```

The first three steps of LJTKC coincide with LJT as specified in Alg. 2: LJTKC builds an FO jtree J for G , enters \mathbf{E} into J , and passes messages in J using LVE for message calculations. During evidence entering, each local model covering evidence randvars absorbs evidence. LJTKC calculates messages based on local models with absorbed evidence, spreading the evidence information along with other local information. After message passing, each parcluster \mathbf{C}_i contains in its local model and received messages all information from G and \mathbf{E} . This information is sufficient to answer queries for randvars contained in \mathbf{C}_i and remains valid as long as G and \mathbf{E} do not change. At this point, FOKC starts to interleave with the original LJT procedure.

LJTKC continues its preprocessing. For each parcluster \mathbf{C}_i , LJTKC extracts a submodel G' of local model G_i and all messages received and reduces G' to a WFOMC problem with theory Δ_i and weight functions w_F^i, w_T^i . It does not need to incorporate \mathbf{E} as the information from \mathbf{E} is contained in G' through evidence entering and message passing. LJTKC compiles an FO d-DNNF circuit \mathcal{C}_i for Δ_i and computes a WFOMC c_i on \mathcal{C}_i . In precomputing a WFOMC c_i for each parcluster, LJTKC utilises that the denominator of Eq. (1) is identical for varying queries on the same model and evidence. For each query handled at \mathbf{C}_i , the submodel consists of G' , resulting in the same circuit \mathcal{C}_i and WFOMC c_i .

To answer a query Q_j , LJTKC finds a parcluster \mathbf{C}_i that covers Q_j and compiles an FO d-DNNF circuit \mathcal{C}_q for Δ_i and Q_j . It computes a WFOMC c_q in \mathcal{C}_q and determines an answer to $P(Q_j|\mathbf{E})$ by dividing the just computed WFOMC c_q by the precomputed WFOMC c_i of this parcluster. LJTKC reuses Δ_i, w_T^i , and w_F^i from preprocessing.

Example Run For G_{ex} , LJTKC builds an FO jtree as depicted in Fig. 2. Without evidence, message passing commences. LJTKC sends messages from parclusters \mathbf{C}_1 and \mathbf{C}_3 to parcluster \mathbf{C}_2 and back. For message m_{12} from \mathbf{C}_1 to \mathbf{C}_2 , LJTKC eliminates $Asthma(X)$ from G_1 using LVE. For message m_{32} from

\mathbf{C}_3 to \mathbf{C}_2 , LJTKC eliminates $Cancer(X)$ from G_3 using LVE. For the messages back, LJTKC eliminates $Friends(X, Y)$ each time, for message m_{21} to \mathbf{C}_1 from $G_2 \cup m_{32}$ and for message m_{23} to \mathbf{C}_3 from $G_2 \cup m_{12}$. Each parcluster holds all model information encoded in its local model and received messages, which form the submodels for the compilation steps. At \mathbf{C}_1 , the submodel contains $G_1 = \{g_4\}$ and m_{21} . At \mathbf{C}_2 , the submodel contains $G_2 = \{g_0, g_1, g_2\}$, m_{12} , and m_{32} . At \mathbf{C}_3 , the submodel contains $G_3 = \{g_3, g_5\}$ and m_{23} .

For each parcluster, LJTKC reduces the submodel to a WFOMC problem, compiles a circuit for the problem specification, and computes a parcluster WFOMC. Given, e.g., query randvar $Cancer(eve)$, LJTKC takes a parcluster that contains the query randvar, here \mathbf{C}_3 . It compiles a circuit for the query and Δ_3 , computes a query WFOMC c_q , and divides c_q by c_3 to determine $P(cancer(eve))$. Next, we argue why QA with LJTKC is sound.

Theorem 1. *LJTKC is sound, i.e., computes a correct result for a query Q given a model G and evidence \mathbf{E} .*

Proof sketch. We assume that LJT is correct, yielding an FO jtree J for model G , which means, J fulfils the three junction tree properties, which allows for local computations based on [23]. Further, we assume that LVE is correct, ensuring correct computations for evidence entering and message passing, and that FOKC is correct, computing correct answers for single term queries.

LJTKC starts with the first three steps of LJT. It constructs an FO jtree for G , allowing for local computations. Then, LJTKC enters \mathbf{E} and calculates messages using LVE, which produces correct results given LVE is correct. After message passing, each parcluster holds all information from G and \mathbf{E} in its local model and received messages, which allows for answering queries for randvars that the parcluster contains. At this point, the FOKC part takes over, taking all information present at a parcluster and compiling a circuit and computing a WFOMC, which produces correct results given FOKC is correct. The same holds for the compilation and computations done for query Q . Thus, LJTKC computes a correct result for Q given G and \mathbf{E} . \square

Theoretical Discussion We discuss space and runtime performance of LJT, LVE, FOKC, and LJTKC in comparison with each other.

LJT requires *space* for its FO jtree as well as storing the messages at each parcluster, while FOKC takes up space for storing its circuits. As a combination of LJT and FOKC, LJTKC stores the preprocessing information produced by both LJT and FOKC. Next to the FO jtree structure and messages, LJTKC stores a WFOMC problem specification and a circuit for each parcluster. Since the implementation of LVE for the $X \neq Y$ cases causes LVE (and LJT) to ground, the space requirements during QA are increasing with rising domain sizes. Since LJTKC avoids the groundings using FOKC, the space requirements during QA are smaller than for LJT alone. W.r.t. circuits, LJTKC stores more circuits than FOKC but the individual circuits are smaller and do not require conditioning, which leads to a significant blow-up for the circuits.

LJTKC accomplishes speeding up QA for certain challenging inputs by fusing LJT, LVE, and FOKC. The new algorithm has a faster runtime than LJT, LVE, and FOKC as it is able to precompute reusable parts and provide smaller models for answering a specific query through the underlying FO jtree with its messages and parcluster compilation. In comparison with FOKC, LJTKC speeds up runtimes as answering queries works with smaller models. In comparison with LJT and LVE, LJTKC is faster when avoiding groundings in LVE. Instead of precompiling each parcluster, which adds to its overhead before starting with answering queries, LJTKC could compile on demand. On-demand compilation means less runtime and space required in advance but more time per initial query at a parcluster. One could further optimise LJTKC by speeding up internal computations in LVE or FOKC (e.g., caching for message calculations or pruning circuits using context-specific information)

In terms of *complexity*, LVE and FOKC have a time complexity linear in terms of the domain sizes of the model logvars for models that allow for a lifted solution. LJT with LVE as a subroutine also has a time complexity linear in terms of the domain sizes for query answering. For message passing, a factor of n , which is the number of parclusters, multiplies into the complexity, which basically is the same time complexity as answering a single query with LVE. LJTKC has the same time complexity as LJT for message passing since the algorithms coincide. For query answering, the complexity is determined by the FOKC complexity, which is linear in terms of domain sizes. Therefore, LJTKC has a time complexity linear in terms of the domain sizes. Even though, the original LVE and LJT implementations show a practical problem in translating the theory into an efficient program, the worst case complexity for liftable models is linear in terms of domain sizes.

The next section presents an empirical evaluation, showing how LJTKC speeds up QA compared to FOKC and LJT for challenging inputs.

5 Empirical Evaluation

This evaluation demonstrates the speed up we can achieve for certain inputs when using LJT and FOKC in conjunction. We have implemented a prototype of LJT, named `ljt` here. Taghipour provides an implementation of LVE (available at <https://dtai.cs.kuleuven.be/software/gcfove>), named `lve`. Van den Broeck provides an implementation of FOKC (available at <https://dtai.cs.kuleuven.be/software/wfomc>), named `fokc`. For this paper, we integrated `fokc` into `ljt` to compute marginals at parclusters, named `ljtkc`. Unfortunately, the FOKC implementation does not handle evidence in a lifted manner as described in [7]. Therefore, we do not consider evidence as `fokc` runtimes explode. We have also implemented the propositional junction tree algorithm (`jt`).

This evaluation has two parts: First, we test an input model with inequalities to highlight how runtimes of LVE and LJT explode, and how LJTKC provides a speedup. Second, we test a version of the model without inequalities to highlight how runtimes of LVE and LJT compare to FOKC without inequalities.

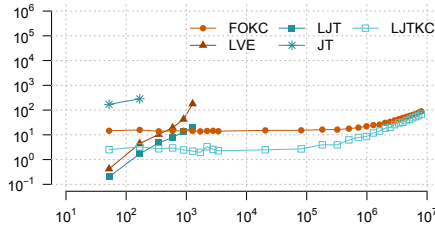


Fig. 3. Runtimes [ms] for G_l ; on x-axis: $|gr(G_l)|$ from 52 to 8,010,000

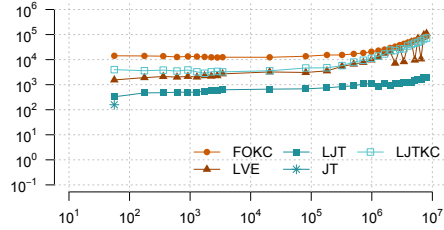


Fig. 4. Runtimes [ms] for G'_l ; on x-axis: $|gr(G'_l)|$ from 56 to 8,012,000

We compare overall runtimes without input parsing averaged over five runs with a working memory of 16GB. `lve` eliminates all non-query randvars from its input model for each query, grounding in the process. `ljt` builds an FO jtree for its input model, passes messages, and then answers queries on submodels. `fokc` forms a WFOMC problem for its input model, compiles a model circuit, compiles for each query a query circuit, and computes the marginals of all PRVs in the input model with random groundings. `ljtkc` starts like `ljt` for its input model until answering queries. It then calls `fokc` at each parcluster to compute marginals of parcluster PRVs with random groundings. `jt` receives the grounded input models and otherwise proceeds like `ljt`.

Inputs with Inequalities For the first part of this evaluation, we test a slightly larger model G_l that is an extension of G_{ex} . G_l has two more logvars, each with its own domain, and eight additional PRVs with one or two parameters. The PRVs are arguments to twenty parfactors, each parfactor with one to three inputs. The FO jtree for G_l has six parclusters, the largest one containing five PRVs. We vary the domain sizes from 2 to 1000, resulting in $|gr(G_l)|$ from 52 to 8,010,000. We query each PRV with random groundings, leading to 12 queries, respectively, among them $Smokes(p_1)$, where p_1 stands for a domain value of X . Figure 3 shows for G_l runtimes in milliseconds [ms] with increasing $|gr(G_l)|$ on log-scaled axes, marked as follows (points are connected for readability): `fokc`: circle, orange, `jt`: star, turquoise, `ljt`: filled square, turquoise, `ljtkc`: hollow square, light turquoise, and `lve`: triangle, dark orange.

The `jt` runtimes are much longer with the first setting than the other runtimes. Up to the third setting, `lve` and `ljt` perform better than `fokc` with `ljt` being faster than `lve`. From the seventh setting on, memory errors occur for both `lve` and `ljt`. `ljtkc` performs best from the third setting onwards. `ljtkc` and `fokc` show the same steady increase in runtimes. `ljtkc` runtimes have a speedup of a factor from 0.13 to 0.76 for G_l compared to `fokc`. Up to a domain size of 100 ($|gr(G_l)| = 81,000$), `ljtkc` saves around one order of magnitude.

For small domain sizes, `ljtkc` and `fokc` perform worst. With increasing domain sizes, they outperform the other programs. Though not part of the numbers in this evaluation, with an increasing number of parfactors, `ljtkc` promises to outperform `fokc` even more, especially with smaller domain sizes.

Inputs without Inequalities For the second part of this evaluation, we test an input model G'_l , that is the model from the first part but with Y receiving an own domain as large as X , making the inequality superfluous. Domain sizes vary from 2 to 1000, resulting in $|gr(G'_l)|$ from 56 to 8,012,000. Each PRV is a query with random groundings again (without a Y grounding). Figure 4 shows for G'_l runtimes in milliseconds [ms] with increasing $|gr(G)|$, marked as before. Both axes are log-scaled. Points are connected for readability.

`jt` is the fastest for the first setting. With the following settings, `jt` runs into memory problems while runtimes explode. `lve` and `ljt` do not exhibit the runtime explosion without inequalities. `lve` has a steadily increasing runtime for most parts, though a few settings lead to shorter runtimes with higher domain sizes. We could not find an explanation for the decrease in runtime for those handful of settings. Overall, `lve` runtimes rise more than the other runtimes apart from `jt`. `ljtkc` exhibits an unsteady runtime performance on the smaller model, though again, we could not find an explanation for the jumps between various sizes. With the larger model, `ljtkc` shows a more steady performance that is better than the one of `fokc`. `ljtkc` is a factor of 0.2 to 0.8 faster. `fokc` and `ljt` runtimes steadily increase with rising $|gr(G)|$. `ljt` gains over an order of magnitude compared to `fokc`. In the larger model, `ljt` is a factor of 0.02 to 0.06 than `fokc` over all domain sizes. `ljtkc` does not perform best as the overhead introduced by FOKC does not pay off as much for this model without inequalities. In fact, `ljt` performs best in almost all cases.

In summary, without inequalities `ljt` performs best on our input models, being faster by over an order of magnitude compared to `fokc`. Though, `ljtkc` does not perform worst, `ljt` performs better and steadier. With inequalities, `ljtkc` shows promise in speeding up performance.

6 Conclusion

We present a combination of FOKC and LJT to speed up inference. For certain inputs, LJT (with LVE as a subroutine) and FOKC start to struggle either due to model structure or size. LJT provides a means to cluster a model into submodels, on which any exact lifted inference algorithm can answer queries given the algorithm can handle evidence and messages in a lifted way. FOKC fused with LJT and LVE can handle larger models more easily. In turn, FOKC boosts LJT by avoiding groundings in certain cases. The fused algorithm enables us to compute answers faster than LJT with LVE for certain inputs and LVE and FOKC alone.

We currently work on incorporating FOKC into message passing for cases where an problematic elimination occurs during message calculation, which includes adapting an FO jtree accordingly. We also work on learning lifted models to use as inputs for LJT. Moreover, we look into constraint handling, possibly realising it with answer-set programming. Other interesting algorithm features include parallelisation and caching as a means to speed up runtime.

References

1. Ahmadi, B., Kersting, K., Mladenov, M., Natarajan, S.: Exploiting Symmetries for Scaling Loopy Belief Propagation and Relational Training. *Machine Learning* **92**(1), 91–132 (2013)
2. Apsel, U., Brafman, R.I.: Extended Lifted Inference with Joint Formulas. In: *UAI-11 Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence* (2011)
3. Bellodi, E., Lamma, E., Riguzzi, F., Costa, V.S., Zese, R.: Lifted Variable Elimination for Probabilistic Logic Programming. *Theory and Practice of Logic Programming* **14**(4–5), 681–695 (2014)
4. Braun, T., Möller, R.: Lifted Junction Tree Algorithm. In: *Proceedings of KI 2016: Advances in Artificial Intelligence*. pp. 30–42. Springer (2016)
5. Braun, T., Möller, R.: Lifted Most Probable Explanation. In: *Proceedings of the International Conference on Conceptual Structures*. pp. 39–54. Springer (2018)
6. van den Broeck, G.: *Lifted Inference and Learning in Statistical Relational Models*. Ph.D. thesis, KU Leuven (2013)
7. van den Broeck, G., Davis, J.: Conditioning in First-Order Knowledge Compilation and Lifted Probabilistic Inference. In: *Proceedings of the 26th AAAI Conference on Artificial Intelligence*. pp. 1961–1967 (2012)
8. van den Broeck, G., Niepert, M.: Lifted Probabilistic Inference for Asymmetric Graphical Models. In: *AAAI-15 Proceedings of the 29th Conference on Artificial Intelligence*. pp. 3599–3605 (2015)
9. van den Broeck, G., Taghipour, N., Meert, W., Davis, J., Raedt, L.D.: Lifted Probabilistic Inference by First-order Knowledge Compilation. In: *IJCAI-11 Proceedings of the 22nd International Joint Conference on Artificial Intelligence* (2011)
10. Chavira, M., Darwiche, A.: Compiling Bayesian Networks Using Variable Elimination. In: *IJCAI-07 Proceedings of the 20th International Joint Conference on Artificial Intelligence*. pp. 2443–2449 (2007)
11. Chavira, M., Darwiche, A.: On Probabilistic Inference by Weighted Model Counting. *Artificial Intelligence* **172**(6–7), 772–799 (2008)
12. Choi, J., Amir, E., Hill, D.J.: Lifted Inference for Relational Continuous Models. In: *UAI-10 Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence*. pp. 13–18 (2010)
13. Darwiche, A., Marquis, P.: A Knowledge Compilation Map. *Journal of Artificial Intelligence Research* **17**(1), 229–264 (2002)
14. Das, M., Wu, Y., Khot, T., Kersting, K., Natarajan, S.: Scaling Lifted Probabilistic Inference and Learning Via Graph Databases. In: *Proceedings of the SIAM International Conference on Data Mining*. pp. 738–746 (2016)
15. Gogate, V., Domingos, P.: Exploiting Logical Structure in Lifted Probabilistic Inference. In: *Working Note of the Workshop on Statistical Relational Artificial Intelligence at the 24th Conference on Artificial Intelligence*. pp. 19–25 (2010)
16. Gogate, V., Domingos, P.: Probabilistic Theorem Proving. In: *UAI-11 Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*. pp. 256–265 (2011)
17. Kazemi, S.M., Poole, D.: Why is Compiling Lifted Inference into a Low-Level Language so Effective? In: *IJCAI-16 Statistical Relational AI Workshop* (2016)
18. 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)

19. Milch, B., Zettelmoyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted Probabilistic Inference with Counting Formulas. In: AAAI-08 Proceedings of the 23rd Conference on Artificial Intelligence. pp. 1062–1068 (2008)
20. Poole, D.: First-order Probabilistic Inference. In: IJCAI-03 Proceedings of the 18th International Joint Conference on Artificial Intelligence (2003)
21. Poole, D., Zhang, N.L.: Exploiting Contextual Independence in Probabilistic Inference. *Journal of Artificial Intelligence* **18**, 263–313 (2003)
22. de Salvo Braz, R.: Lifted First-order Probabilistic Inference. Ph.D. thesis, University of Illinois at Urbana Champaign (2007)
23. Shenoy, P.P., Shafer, G.R.: Axioms for Probability and Belief-Function Propagation. *Uncertainty in Artificial Intelligence 4* **9**, 169–198 (1990)
24. Singla, P., Domingos, P.: Lifted First-order Belief Propagation. In: AAAI-08 Proceedings of the 23rd Conference on Artificial Intelligence. pp. 1094–1099 (2008)
25. Taghipour, N., Davis, J.: Generalized Counting for Lifted Variable Elimination. In: Proceedings of the 2nd International Workshop on Statistical Relational AI. pp. 1–8 (2012)
26. Taghipour, N., Fierens, D., Davis, J., Blockeel, H.: Lifted Variable Elimination: Decoupling the Operators from the Constraint Language. *Journal of Artificial Intelligence Research* **47(1)**, 393–439 (2013)
27. Vlasselaer, J., Meert, W., van den Broeck, G., Raedt, L.D.: Exploiting Local and Repeated Structure in Dynamic Bayesian Networks. *Artificial Intelligence* **232**, 43–53 (2016)
28. Zhang, N.L., Poole, D.: A Simple Approach to Bayesian Network Computations. In: Proceedings of the 10th Canadian Conference on Artificial Intelligence. pp. 171–178 (1994)