Hypertree Decompositions Revisited for PGMs

Aarthy Shivram Arun[1], **Sai Vikneshwar Mani Jayarman**[1], **Christopher Ré**[2] **and Atri Rudra**[1]
[1] University at Buffalo, SUNY, {ashivram, saivikne, atri}@buffalo.edu.
[2] Stanford University, chrismre@cs.stanford.edu.

Abstract

We revisit the classical problem of exact inference on probabilistic graphical models (PGMs). Our algorithm is based on recent *worst-case optimal database join* algorithms, which can be asymptotically faster than traditional data processing methods. We present the first empirical evaluation of these algorithms via JoinInfer – a new exact inference engine. We empirically explore the properties of the data for which our engine can be expected to outperform traditional inference engines, refining current theoretical notions. Further, JoinInfer outperforms existing state-of-the-art inference engines (ACE, IJGP and libDAI) on some standard benchmark datasets by up to a factor of 630x. Finally, we propose a promising data-driven heuristic that extends JoinInfer to automatically tailor its parameters and/or switch to the traditional inference algorithms.

Introduction

Efficient inference on probabilistic graphical models (PGMs) is a core topic in artificial intelligence (AI) and standard inference techniques are based on tree decompositions (Jensen, Lauritzen, and Olesen 1990; Dechter 1996; Kask et al. 2005; Mateescu et al. 2010). The runtime of such inference algorithms is exponential in the treewidth (tw) of the underlying graph, which in the worst case, is unavoidable. Over the years, efforts in the logic, database and AI communities to refine tw into a finer-grained measure of complexity have culminated in *generalized hypertree decompositions* (GHDs) (Fischl, Gottlob, and Pichler 2016; Gottlob et al. 2005). Recently, FAQ/AJAR (Abo Khamis, Ngo, and Rudra 2016; Joglekar, Puttagunta, and Ré 2016) theoretically reconnected such GHD-based algorithms with probabilistic inference exploiting recent developments in *worst-case optimal database join* algorithms (Ngo et al. 2012) and achieved tighter bounds based on a finer-grained notion of width called *fractional-hypertreewidth* (fhtw). Further, given the known connection between database joins and CSPs (**?**; **?**), their bounds apply to many classes of CSPs as well.

However, the practical significance of GHD-based inference algorithms has met with some skepticism so far. In particular, Dechter et al. (**?**) use a predictive ratio (R) (based on hypertreewidth htw and tw) and concluded that classical treewidth-based algorithms outperform their GHD-based counterparts on an overwhelming majority of PGM benchmarks. Further, their study suggests that the advantages of GHD-based algorithms manifests only in instances with substantial factor sparsity (i.e. a large number of factor entries having zero probabilities) and high factor arity.

In addition to the above constraints, translating the superior theoretical guarantees of GHD-based algorithms into practice is a non-trivial challenge. In particular, these algorithms (Abo Khamis, Ngo, and Rudra 2016; Joglekar, Puttagunta, and Ré 2016) typically assume that one can exhaustively search over all potential GHDs (which grows exponentially in the number of variables) and their runtime analysis ignores the dependence on number of variables/factors. Unfortunately, in practice, these assumptions could negate the theoretical gains.

Our Contributions. In this paper, we introduce JoinInfer, a proof-of-concept inference engine, to address all the above problems: JoinInfer is efficient and can be up to 630x faster than its competitors. Further, we introduce a theoretical measure that better predicts when JoinInfer can outperform its competitors.

GHDs Revisited: We empirically demonstrate that JoinInfer (a GHD-based algorithm) has wider scope than previously predicted, revisiting the conclusions of (**?**). We do so with two new measures – ρ, the total number of entries processed across all bags of the GHD and R_J, a better predictor of JoinInfer's performance. We observe that R_D (our analogous version of R from (**?**) where htw is replaced by fhtw), is contingent upon ρ (see ρ high and ρ low classes in Figure 1), an insight that is not captured in (**?**)'s experimental paradigm. In particular, engines employing truth-table indices (libDAI/IJGP) do not scale to higher values of ρ, whereas JoinInfer does (see Bands $1-3$ in Figure 1). For instance, in Band 3 in Figure 1, JoinInfer outperforms competing engines by up to 2.7x, while R_D predicts underperformance by more than 10^{10}x.

We then exploit the better theoretical runtime analysis of JoinInfer to introduce R_J, a finer-grained theoretical measure that better predicts its performance. For instance, consider the 'R_D small' column in Figure 1 – when ρ is high (Band 1 in Figure 1), JoinInfer is up to 630x faster than competing engines and when ρ is low (Band 4 in Figure 1), JoinInfer is only up to 5x faster. We note that R_J can actu-

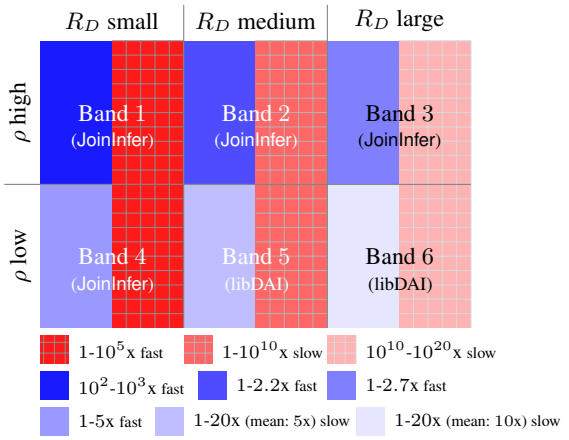

Figure 1: Datasets are divided into six bands depending on the sizes of R_D and ρ. The grids in each box denote the expected speedup of any GHD-based system over a treewidth based system and gray shades show the actual speedup of JoinInfer with respect to libDAI. The "winner" is stated explicitly for each band.

ally differentiate between these two rows while R_D predicts a similar speedup for both bands (see columns 4-7 in Table 1).

Technical Contributions: JoinInfer leverages recently introduced worst-case optimal join algorithms (Ngo, Ré, and Rudra 2014) in conjunction with improved data structures. In particular, we use two data representations for use in different passes of the algorithm – a level-order trie, which collapses a conventional trie into a single array and (two variants of) an index-based compressed list. We find that the resulting gains more than compensate for the overheads involved in maintaining both data representations.

Hybrid Architecture: Given the relative advantages of JoinInfer (e.g, Bands $1-4$ in Figure 1) and libDAI (e.g., Bands $5-6$ in Figure 1) in different spaces, we explore the feasibility of exploiting their strengths in a 'best-of-all-worlds' architecture. Our hybrid system (HJYAR) outperforms libDAI, IJGP and ACE for 75% of the networks (see Table 1), illustrating its promise.

Related Work

Several streams of inquiry have emerged in the exact inference setting. One such stream involves *conditioning algorithms* (Pearl 1989; Darwiche 2001) that adopt a case-based reasoning approach. Another class of algorithms seeks to exploit local structure (Larkin and Dechter 2003; Poole and Zhang 2003), where (Chavira and Darwiche 2007; Huang, Chavira, and Darwiche 2006; Chavira and Darwiche 2005; Darwiche 2001) exploit factor sparsity to improve tractability of inference. Recently, an emerging area *lifted probabilistic inference* (de Salvo Braz, Amir, and Roth 2005; Milch et al. 2008; Kersting 2012), exploits symmetric structures within graphs to speed up inference. Yet another line of work runs along variable elimination (Dechter 1996; Zhang and Poole 1996) and tree decomposition-based routines (Jensen, Lauritzen, and Olesen 1990; Kask et al. 2005; Mateescu et al. 2010). Finally, past work in PGMs has also focused on approximate inference (Koller and Friedman 2009; Bekker et al. 2015; Bach and Jordan 2001); we believe that advancements in JoinInfer could enhance their performance.

JoinInfer: An Overview

We start by giving a brief overview of the background concepts and outline the *worst-case optimal join* algorithm MultFacProd that JoinInfer uses for computing factor products. We then show how MultFacProd fits in JoinInfer – our GHD-based Message Passing Algorithm. Finally, we talk about implementation challenges and our solutions, including a Hybrid Architecture.

Preliminaries and Notation

Definition 1 *A (discrete)* **probabilistic graphical model** *can be defined by the triplet $\langle \mathcal{H}, D, \mathcal{K} \rangle$, where hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ represents the underlying graphical structure (note $\mathcal{E} \subseteq 2^{\mathcal{V}}$). There are $n = |\mathcal{V}|$ discrete random variables on finite domains $D = \{D(U) : U \in \mathcal{V}\}$ and $m = |\mathcal{E}|$ factors $\mathcal{K} = \{\phi_e\}_{e \in \mathcal{E}}$, where each factor ϕ_e is a mapping: $\phi_e : \prod_{U \in e} D(U) \to \mathbb{R}_+$.*

For instance, Figure **??** in the full version is a hypergraph representing a PGM with variables $\mathcal{V} = \{A, B, C, D\}$, edges $\mathcal{E} = \{e{=}(A, B),\ f{=}(A, C),\ g{=}(B, C, D)\}$ and factors $\mathcal{K} = \{\phi_e(A, B), \phi_f(A, C), \phi_g(B, C, D)\}$.

Definition 2 *For any factor ϕ, the size of ϕ is its support size, i.e., the number of entries with non-zero probabilities. Storing only the non-zero entries (as well as their ϕ values) is called the* listing representation *of ϕ.* Factor sparsity *is defined as $\frac{N}{\prod_{U \in \phi} |D(U)|}$, where N is the size of factor ϕ.*

A typical inference task in PGMs is to compute the marginal estimates given by: $\forall F \subseteq \mathcal{V}, \mathbf{y} \in \prod_{U \in F} D(U)$,

$$\phi_F(\mathbf{y}) = \frac{1}{Z} \sum_{\mathbf{z} \in \prod_{U \in \mathcal{V} \setminus F} D(U)} \prod_{S \in \mathcal{E}} \phi_S(\mathrm{x}_S), \qquad (1)$$

where $\mathbf{x} = (\mathbf{y}, \mathbf{z})$, x_S denotes the projection of x onto the variables in S and Z is a normalization constant. Variable/Factor marginals are a special case of (1); $F = \{U\}$ for $U \in \mathcal{V}$ for variable marginals and $F \in \mathcal{E}$ for factor marginals.

Exact inference in PGMs is usually performed by propagating on a ***generalized hypertree decomposition (GHD)*** of the underlying hypergraph $\mathcal{H}$.

Definition 3 *A GHD of $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is defined by a triple $\langle T, \chi, \lambda \rangle$, where $T = (V(T), E(T))$ is a tree, $\chi : V(T) \to 2^{\mathcal{V}}$ is a function associating a set of vertices $\chi(v) \subseteq \mathcal{V}$ to each node v of T, and $\lambda : V(T) \to 2^{\mathcal{E}}$ is a function associating a set of hyperedges to each node v of T such that the following properties hold (i) for each $e \in \mathcal{E}$, there is a node $v \in V(T)$ such that $e \subseteq \chi(v)$ and $e \in \lambda(v)$; and (ii) for every $V' \subseteq \mathcal{V}$, the set $\{v \in V(T) | V' \subseteq \chi(v)\}$ is connected in T. We define* treewidth $\mathrm{tw} = \max_{v \in V(T)}(|\chi(v)|)$.

Existing GHD-based Message Passing Algorithms. A GHD can be thought of as a labeled (hyper)tree T, where sets assigned to each node in T are called *bags* of the hypertree. Inference propagation on T involves a two-pass 'message-passing' algorithm (Jensen, Lauritzen, and Olesen 1990). In the first pass (message-up), *messages* are propagated 'up' from leaf (child) to root (parent). Subsequently in the second pass (message-down), they are propagated 'down' from root (parent) to leaf (child).

More formally, a *message* ${\phi'}_{m_{v,u}}$ from node v to node u is a marginal estimate given by $\sum_{U \notin \chi(v) \cap \chi(u)} \phi'_v$, where

$$ {\phi'}_v = \prod_{e \in \lambda(v)} \phi_e \cdot \prod_{w \in \mathsf{Children}(v)} {\phi'}_{m_{w,v}}, \tag{2} $$

for the upward pass. For the downward pass, $\mathsf{Children}(v)$ will be replaced by $\mathsf{Parent}(v)$. Upon completion of both passes (up/down), variable marginals for all $U \in \chi(v)$ can be retrieved using label $\chi(v)$ and factor marginals for all $e \in \lambda(v)$ can be retrieved using $\lambda(v)$ from each node $v \in V(T)$. Computing ϕ'_v is the major bottleneck in message-passing algorithms and we focus on this step next.

MultFacProd: A New Algorithm for Computing Factor Products

In this section, we describe MultFacProd, the *worst-case optimal join* algorithm used by JoinInfer to compute ϕ'_v for every $v \in V(T)$. We first present a motivating example, followed by the runtime analysis of MultFacProd.

Triangle Query As an example, consider the *triangle* PGM with variables $\mathcal{V} = \{A, B, C\}$, edges $\mathcal{E} = \{e = (A, B), f = (B, C), g = (A, C)\}$ and factors $\mathcal{K} = \{\phi_e(A, B), \phi_f(B, C), \phi_g(A, C)\}$. Let $|D(U)| = D$ for all $U \in \mathcal{V}$ and $|\phi_e(A, B)| = |\phi_f(B, C)| = |\phi_g(A, C)| = N \leq D^2$. We would like to compute the factor product $\phi'(A, B, C) = \phi_e(A, B) \cdot \phi_f(B, C) \cdot \phi_g(C, A)$ since the GHD for this PGM would contain only one node with variables (A, B, C). Prior to (Abo Khamis, Ngo, and Rudra 2016; Joglekar, Puttagunta, and Ré 2016), there are two algorithms used for computing factor products:

- The first would go over all D^3 possible output tuples (a, b, c) and compute the product of corresponding probability values, resulting in an overall runtime of $O(D^3)$. This algorithm is called MultiplyFactors [(**?**), Chapter 6].
- The second would compute an intermediate product (say) $\phi_e(A, B) \cdot \phi_f(B, C)$ to get up to N^2 possible tuples (a, b, c) and then filter this against $\phi_g(A, C)$. This takes $O(N^2)$ time and is faster than $O(D^3)$ if $N \ll D^{\frac{3}{2}}$.

We would like to mention here that the engines libDAI and IJGP compute intermediate pairwise products on a truth-table indexing scheme, staying closer to the runtime of $O(D^3)$ (see full version of this algorithm). The most interesting thing about *worst-case join algorithms* (and hence MultFacProd) is that they compute the above product in time $O(N^{\frac{3}{2}})$, which is asymptotically better than both the above algorithms as long as $N \ll D^2$. The key insight in MultFacProd is that if $\phi_e(A, B) \cdot \phi_f(B, C) \cdot \phi_g(A, C)$ is computed in a multiway fashion, then one can exploit sparsity in the input factors (which MultiplyFactors fails to do) and avoid computing larger intermediate products (which PairwiseProd fails to do). As an extreme example, consider the case when $\phi_e(A, B)$ has N non-zero tuples of the form $[N] \times [1]$ and $\phi_f(B, C)$ has N non-zero tuples of the form $[1] \times [N]$. The pairwise product $\phi_e(A, B) \cdot \phi_f(B, C)$ will thus have N^2 non-zero entries $[N] \times [1] \times [N]$. However, we know only N of them would survive since $\phi_g(A, C)$ has exactly N non-zero entries. In this case, MultFacProd will first fix variable A's value to a and B's value of 1 (i.e, $(a, 1) \in \phi_e(A, B)$) and then obtain all values of c such that $(1, c) \in \phi_f(B, C)$ and $(a, c) \in \phi_g(A, C)$. Note that there exists a unique value of c for a fixed a. Thus, MultFacProd will process only N entries overall in this case since $|\phi_e(A, B)| \leq N$. (See Algorithm 1 for our algorithm for the triangle case and full version for general algorithm.)

Algorithm 1 MultFacProd for Triangle

1: **Input:** Variables $\mathcal{V} = \{A, B, C\}$, Edges $\mathcal{E} = \{e = (A, B), f = (B, C), g = (A, C)\}$ and Factors $\mathcal{K} = \{\phi_e(A, B), \phi_f(B, C), \phi_g(A, C)\}$.
2: **Output:** Factor Product $\phi'(A, B, C) = \phi_e(A, B) \cdot \phi_f(B, C) \cdot \phi_g(A, C)$.
3: **for all** a s.t. $\exists b, c$ with $\phi_e(A = a, B = b) \cdot \phi_f(A = a, C = c) \neq 0$ **do**
4: **for all** b s.t. $\exists c$ with $\phi_e(A = a, B = b) \cdot \phi_f(B = b, C = c) \neq 0$ **do** ▷ Value for variable A is fixed as a.
5: **for all** c s.t. $\phi_e(A = a, B = b) \cdot \phi_f(B = b, C = c) \neq 0$ **do** ▷ Value for variables A and B are fixed as a and b respectively.
6: $\phi'(A, B, C) \leftarrow \phi'(A, B, C) \cup \{(a, b, c), \phi_e(a, b) \cdot \phi_f(b, c) \cdot \phi_g(a, c)\}$ ▷ The entry (a, b, c) is added to the factor product along with its corresponding probability.
7: **return** $\phi'(A, B, C)$

Runtime Complexity of MultFacProd A recent result of Atserias, Grohe and Marx showed how to tightly bound the worst-case output size of a factor product (Atserias, Grohe, and Marx 2013) and subsequently, (Ngo et al. 2012) came up with MultFacProd that can run in time of the worst-case output size. For a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, let $\mathbf{x} \in \mathbb{R}^{|\mathcal{E}|}$ be a vector indexed by edges, such that $\mathbf{x}^* = (x_e)_{e \in \mathcal{E}}$ is an optimal solution to the linear program

$$ \min \quad \sum_{e \in \mathcal{E}} x_e \log_2 |\phi_e| \tag{3} $$

$$ \text{s.t.} \quad \sum_{v \ni e} x_e \geq 1 \forall v \in \mathcal{V}; x_e \geq 0 \forall e \in \mathcal{E}. \tag{4} $$

Then, we can bound the size of the factor product $\phi' = \prod_{e \in \mathcal{E}} \phi_e$ as follows:

$$ |\phi'| \leq \prod_{e \in \mathcal{E}} |\phi_e|^{x_e}. \tag{5} $$

In particular, the runtime of MultFacProd is $O(\prod_{e \in \mathcal{E}} |\phi_e|^{x_e})$ and is reflected in the triangle example we considered earlier.

Example 1 *In the previous section, we considered the triangle query with edges* $e = (A, B)$, $f = (B, C)$ *and* $g = (A, C)$ *respectively. Solving the linear program* (3) *for this case, we have* $\mathbf{x}^* = (\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. *Note that this gives us an asymptotically better bound of* $N^{\frac{3}{2}}$ *since* $|\phi_e(A, B)| = |\phi_f(B, C)| = |\phi_g(A, C)| = N$. *In comparison, the hypertreewidth-based bound is* N^2 $(\mathrm{htw}(T) = 2)$ *and the treewidth-based bound is* D^3 $(\mathrm{tw}(T) = 3)$.

Using MultFacProd in GHD-based Message Passing Algorithms

The GHD-based Message Passing Algorithm (also known as Junction-Tree Algorithm) forms the structure of JoinInfer (we present a sketch of this procedure in Algorithm 2). The input PGM network is transformed into a Junction Tree (i.e., a GHD) using the min-fill heuristic. Then, we root the GHD arbitrarily, determining the parent-child relationship for every node. We assume that each input factor is assigned to a unique bag in the GHD. In particular, for every node $v \in V(T)$, $\alpha(v)$ denotes the input factors assigned to it.

Algorithm 2 JoinInfer

1: **Input:** A PGM $\mathcal{P} = (\mathcal{H}, D, \mathcal{K})$.
2: **Output:** Variable and Factor Marginals.
3: Create a rooted GHD $\mathcal{G} = ((V, E), \chi, \lambda)$ for $\mathcal{P}$. ▷ Using min-fill variable ordering.
4: $R \leftarrow$ JoinInferSampling$(\mathcal{G})$ ▷ $R : V \to \{0, 1, 2\}$ and JoinInferSampling (i.e., Algorithm in the full version.
5: $(\{\phi'_v\}_{v \in V}, \{\phi'_{m_{v,\mathsf{Parent}(v)}}\}_{v \in V}) \leftarrow$ JoinInferUp$(\mathcal{G}, R)$
6: $\{\phi'_v\}_{v \in V} \leftarrow$ JoinInferDown$(\mathcal{G}, \{\phi'_v\}_{v \in V}, \{\phi'_{m_{v,\mathsf{Parent}(v)}}\}_{v \in V})$
7: Compute Variable and Factor Marginals from $(\{\phi'_v\}_{v \in V})$

We now describe our contributions in the message-up and message-down phases.

Message-Up Phase The upward pass propagates messages (i.e., marginalized factor products) from leaves to root along the rooted GHD. Recall that this involves computing factor products as in (2) at every node (bag) of the GHD. JoinInfer lends its core contribution in this phase – it uses MultFacProd to compute the factor products. As shown in the Triangle example earlier, MultFacProd is different from previously proposed exact inference algorithms and performs a multiway product to achieve asymptotically better bounds (see Algorithm 1 for an outline). We present the general algorithm in the full version (adapted from (Ngo et al. 2012)).

Further, for a given bag, in addition to input factors mapped/messages received by it, JoinInfer includes factors that are not originally mapped to the bag, but have non-trivial intersections with variables in the bag (using ideas from Section 3.2 in (**?**) and we call these factors "01-projections"). Note that these are crucial for theoretical results for FAQ/AJAR and can prune factor product entries early on in the presence of factor sparsity.

We present our Message-Up procedure in Algorithm 3.[1]

Algorithm 3 JoinInferUp

1: **Input:** GHD $\mathcal{G} = ((V, E), \chi, \lambda)$ and the map R.
2: **Output:** Factor Products $\{\phi'_v\}_{v \in V}$ and Up Messages $\{\phi'_{m_{v,\mathsf{Parent}(v)}}\}_{v \in V}$ both as tries.
3: **for all** nodes $v \in V$ **do**: ▷ This is done in a level-order traversal from leaves-to-root.
4: $\mathcal{E}_v \leftarrow \lambda(v), \mathcal{K}_v \leftarrow \{\phi_e : e \in \lambda(v)\}$ ▷ Initialize the PGM Query corresponding to v's Factor Product.
5: **for all** $w \in$ Children(v) **do** ▷ We add all the messages sent to v from its children.
6: $\mathcal{E}_v \leftarrow \mathcal{E}_v \cup \{\chi(v) \cap \chi(w)\}, \mathcal{K}_v \leftarrow \mathcal{K}_v \cup \{\phi'_{m_{w,v}}\}$
7: **if** $R(v) = 1$ **then** ▷ We include the $0/1$ projections while computing the Factor Product for v.
8: **for all** $e \in \mathcal{E} \setminus \lambda(v)$ **do**
9: **if** $e \cap \chi(v) \neq \emptyset$ **then**
10: $\mathcal{E}_v \leftarrow \mathcal{E}_v \cup \{e \cap \chi(v)\}, \mathcal{K}_v \leftarrow \mathcal{K}_v \cup \{\phi_{e/\chi(v)}\}$
11: **if** v is not a root **then**
12: Let $u =$ Parent(v), $\phi'_{m_{v,u}}$ be $\sum_{q \in \chi(v) \setminus \chi(u)} \phi'_v$ and $\mathcal{F}_v = \chi(v) \cap \chi(u)$.
13: $(\phi'_v, \phi'_{m_{v,u}}) \leftarrow$ MultFacProd$(R_v, \chi(v), \mathcal{E}_v, \mathcal{K}_v, \mathcal{F}_v)$
14: **else** $\phi'_v \leftarrow$ MultFacProd$(R_v, \chi(v), \mathcal{E}_v, \mathcal{K}_v, \chi(v))$

Message-Down Phase The downward pass (from root-to-leaves) involves updating factor products for each bag (except the root) using the received messages. Note that there are two products computed for each bag – one between the sent and received message and the second, between the result of the previous step with the bag's factor product. We use *in-place hash products* for this computation (more details in the full version).

Runtime Complexity: An Analysis of JoinInfer For a GHD $G = (T = (V, E), \chi, \lambda)$, we can bound the size of ϕ'_u for every $u \in V(T)$ as follows. For a hypergraph $\mathcal{H}_u = (\mathcal{V}_{\chi(u)}, \mathcal{E})$, let $\mathbf{x} \in \mathbb{R}^{|\mathcal{E}|}$ be a vector indexed by edges, such that $\mathbf{x}^*_u = (x^u_e)_{e \in \mathcal{E}}$ is an optimal solution to (5), bounding $|\phi'_u|$:

$$|\phi'_u| \leq \prod_{e \in \mathcal{E}} |\phi_e|^{x^u_e}. \tag{6}$$

Since we run MultFacProd for every $u \in V(T)$, the total runtime of JoinInfer is $O(\sum_{u \in V(T)} \prod_{e \in \mathcal{E}} |\phi_e|^{x^u_e})$.

Upper-bounding each $|\phi_e|$ by $N = \max_{e \in \mathcal{E}} |\phi_e|$ in the above equation and replacing the sum over all $u \in V(T)$ by max gives us an asymptotic bound of $N^{\mathrm{fhtw}(T)}$. $\mathrm{fhtw}(T)$ is guaranteed to be at most $\mathrm{htw}(T)$ (hypetreewidth), which in turn is at most $\mathrm{tw}(T)$ (treewidth) for the same GHD, giving us the best known theoretical bounds for exact inference in PGMs. (More details in the full version.)

Recall that the asymptotic bounds for tw based algorithms is given by D^{tw}, where $D = \max_{U \in \mathcal{V}} |D(U)|$. However, a more realistic measure here would be $\rho =$

[1] When $R_v = 2$, PairwiseProd (Algorithm in the full version) is run instead of MultFacProd.

$\sum_{u \in V(T)} \prod_{U \in \chi(u)} |D(U)|$. This gives us a fine-grained ratio (as compared to (**?**)) to evaluate JoinInfer against classical engines:

$$R_J = \frac{\sum_{u \in V(T)} \prod_{e \in \mathcal{E}} |\phi_e|^{x_e^u}}{\rho}. \quad (7)$$

Replacing the numerator with N^{fhtw} and the denominator with D^{tw} in (7) gives us

$$R_D = \frac{N^{\text{fhtw}}}{D^{\text{tw}}}. \quad (8)$$

This ratio is analogous to the one in (**?**), which was based on hypertree width:

$$R = \log_{10}\left(\frac{N^{\text{htw}^*}}{D^{tw}}\right).$$

Since computing htw is NP-Hard, (**?**) used an approximation for it (denoted by htw^*). In our measure R_D, we overcome this issue by using fhtw over htw. Using fhtw offers two significant advantages – one, it is a more fine-grained measure (since $\text{fhtw} \leq \text{htw} \leq \text{tw}$) and two, fhtw is polynomially computable (basically, solve the LP from (3)). We empirically demonstrate that R_J is a better predictor than R_D in Section 4.2.1 since it is a more fine-grained measure.

Technical Contributions

Consistent with (Abo Khamis, Ngo, and Rudra 2016; Joglekar, Puttagunta, and Ré 2016), we represent every factor table as a trie (storing only entries with non-zero probabilities),[2] in our implementation of MultFacProd and use a *flattened* version of the tries to exploit caching advantages. In addition to this, we store factor tables as lists of ⟨index value, probability⟩ pairs, where each factor tuple is converted into a corresponding index value. We use two variants – the first stores only 'reverse' indices (computed in reverse variable order) and the second stores forward and reverse indices (for representing intermediary messages). Note that these representations enable us to optimize the up and down passes. In particular, the reverse index enables efficient construction of tries in the up-phase and in decoding message entries over all children in a single pass in the down-phase. Moreover, the reverse indices of the up-messages act as placeholders for down-messages, enabling the reuse of data structures. Finally, the forward indices are used while merging down-messages with cluster products in-place, thus optimizing decoding/encoding steps.

Hybrid Architecture

In Bands $5-6$ (see Figure 1), libDAI's pairwise product implementation demonstrates distinct advantages over JoinInfer's multi-way product. We explore the feasibility of leveraging the respective advantages of both these strategies in a new {HY}brid {J}oin {AR}chitecture (HYJAR). To build such a system, we use the native structure of JoinInfer and import only the pairwise-product functionality from libDAI (we do not integrate the entire engine). Given the high costs of switching between the data structures required for JoinInfer and libDAI, the main challenge here was to devise a system that not only optimally chooses between the strategies per bag, but at the same time minimizes the switches between bags.[3] We overcome this challenge by introducing a deterministic heuristic that decides the optimal strategy (JoinInfer or PairwiseProd) for each bag v in the GHD that has at least one input factor assigned to it (i.e. $\alpha(v) \geq 1$).[4] We then propagate this decision along the subtree of v, until it reaches a bag that was already assigned a decision. To decide the order of preference, we consider bags v in decreasing order of $\prod_{U \in \chi(v)} |D(U)|$, with the intuition that larger bags dominate the runtime of libDAI. More details in the full version.

Experimental Evaluation

In this section, we empirically validate JoinInfer and outline features that influence its performance. We start by describing our empirical setup introducing the standard benchmarks and demonstrate the scope of JoinInfer vis-a-vis state-of-the-art-systems on them. Then, we document performance gains of our hybrid system and finally, evaluate our technical contributions.

Experimental Setup

We first create a testbed of 52 networks that spans the full range of cases illustrated in Figure 1, sampling from three publicly available benchmarks – UAI'06 (Bilmes and Dechter 2006), PIC 2011 (**?**) and the BN Learns dataset (**?**) (which subsumes IJCAI'05 networks (Chavira and Darwiche 2005)). Further, in order to improve the tractability of some of the larger networks (Bands 1, 2 and 3) for exact inference (high ρ cases), we randomly induce factor sparsity.[5] Note that these sparsity levels are consistent with the ranges found in other networks in the benchmark and inducing sparsity to improve model tractability is a well-accepted procedure in many practical settings (**?**; Larkin and Dechter 2003). For the Band 2 networks BN_$30-39$, we modify the original probabilities (these networks are marked with a '*' in Table 1) and provide details in the full version.

We compare JoinInfer against three state-of-the-art systems on the exact inference query of computing all variable marginals: ACE (Chavira and Darwiche 2005), an engine that explicitly exploits determinism, and, libDAI (Mooij 2010) and IJGP (Mateescu et al. 2010), two award winning systems in the UAI 2010 inference challenge. Additionally, while JoinInfer, IJGP and ACE process evidence, libDAI

[2] A trie is a multi-level data structure where each factor tuple corresponds to a unique path from root to leaf and the probability value associated with each tuple are stored in the leaf.

[3] We would like to mention here that ACE uses a similar rule at the GHD-level but ours works at the bag-level.

[4] Recall our earlier assumption that each factor table is assigned to a unique bag. As a result, not many bags are chosen in this process. Further we ignore the incoming messages for a bag v when deciding on R_v, making this decision faster.

[5] Details in the full version.

does not. Hence, to ensure a fair comparison, we incorporate evidence information directly into the input given to the engines,[6] We compare our marginal outputs with these engines, with an error limit of 0.00001. We evaluate all the systems on the average time taken across 5 runs to compute all the variable marginals (setting a timeout of 60 minutes for each run). Further, ACE requires separate compilation of the arithmetic circuit representing the input network (a non-standard design). For a fair comparison with other engines with end-to-end computations, we report two times for ACE – the sum of compilation and inference times, followed by inference time.

We ran all our experiments on a Linux server (Ubuntu 16.04 LTS) with Intel Xeon E5-2640 v3 CPU @ 2.60GHz and 64 GB RAM.

Experimental Results

Benchmark Experiments The results in Table 1 are laid out along the lines of Figure 1. These networks span over a wide range of sparsity ($20\% - 100\%$), domain sizes ($2 - 100$) and factor arity levels ($1 - 10$).

ρ ***High.*** The measure R_D from (**?**) predicts superior performance for JoinInfer only in Band 1 (CELAR). However, in this region of high ρ, JoinInfer performs consistently better than the predictions in (**?**). In Band 1, it is be up to 630x faster on subsets where ACE completes. In Band 2, it is up to 2.2x faster and in Band 3 where the corresponding predictions of (**?**) is under-performance by $10^{10}\times$ - $10^{20}\times$, it can be upto 2.7x faster than ACE (libDAI and IJGP fail in this space). libDAI and IJGP fail in these bands due to huge pre-memory allocation. On the other hand, ACE that takes advantage of *factor sparsity* using arithmetic circuits is the only other engine that completes; that said, compiling these structures is costly. In Band 1, we surmise that JoinInfer's performance advantages are rooted in the use of MultFacProd as opposed to standard algorithms, which is predicted by R_J. The networks in these bands ($1 - 3$) cover a sparsity range of $20\% - 50\%$ and have factor arity level $1 - 4$. Further, in Band 1, given that ACE requires the 20% sparsity levels to complete, we present results at two levels of sparsity for CELAR (20% and 40%).

ρ ***Low.*** R_D predicts superior performance for JoinInfer in Band 4, which it achieves. It is upto 5.29x faster than libDAI (it's closest competitor) and up to 5.4x faster than ACE (on the subsets that ACE completes on). IJGP times out on almost all of the networks. Finally, in Bands $5/6$, the two unfavorable settings, JoinInfer is on an average faster than ACE by 5.8x/9.5x and IJGP by 2.36x/2.28x respectively. It is on an average slower than libDAI by $4.8\times/10\times$ respectively: libDAI's truth-table indexing advantages clearly manifest in these two bands. We would like to note however, that the corresponding predictions of (**?**) for JoinInfer is under performance by 10x - 10^8x for Band 5, and 10^{20}x - 10^{28}x for Band 6, i.e., several orders of magnitude worse.

Predictions by R_J**.** Our finer grained ratio R_J (Column 4) better tracks JoinInfer's speed-ups as compared to R_D (Column 5) on most networks. Though it is similar to R_D in Band 1 and within couple of orders magnitude in Band 2, it diverges considerably (more than 10^5x) in Bands 3 and 4 (where JoinInfer's speed-ups are within 1x - 5x). Moreover, R_J has much tighter predictions than R_D across almost all of the networks in the testbed.

In addition to the above, we find that JoinInfer is competitive in terms of memory usage and the use of "01-projections" do not help significantly in our benchmarks (more details in the full version).

Hybrid Architecture (HYJAR). Since JoinInfer is the only engine that completes on all networks when ρ is high, we now focus on low ρ conditions. HYJAR helps exploit the relative strengths of each strategy–multiway or pairwise products–into a single architecture, yielding consistent performance across a majority of networks ($26/28$). Of these, in 9 cases (e.g., munin1, munin, barley, mildew) HYJAR's completion times are faster than its nearest standalone competitors (JoinInfer or libDAI), in 10 cases it is less than 2.5x slower and in 7 cases it is between 2.5x - 4.5x slower. BN_42 and BN_44 are the only two networks where HYJAR's strategy does not lead to a notable improvement. Further, follow up analysis indicates that HYJAR consistently switches between strategies at the bag level (more details in the full version).

Factor Representations. As described in the Overview Section, we store two variants of the list of $\langle$index, probability$\rangle$ pairs – one with a forward index and the other with a reverse index. Our experiments on the UAI speech recognition datasets show that this results in up to 3x, 1.6x and 1.3x gains during building tries, message-up and message-down phases respectively (details in the full version).

Takeaways. We have identified a threshold for ρ at 10^9 reflecting the current memory limits for truth tables (on our machine).[7] Further, we have demonstrated that R_J is a better predictor of JoinInfer's performance than R_D. Finally, HYJAR outperforms libDAI, IJGP and ACE on 39 out of 52 networks (i.e., 75% of them), illustrating its promise as a practically relevant architecture for building a robust, broadly applicable inference engine. Note that the second largest winner is libDAI with wins on only 11 datasets.

Conclusion Our system can be extended to classes of CSPs studied in (Abo Khamis, Ngo, and Rudra 2016), which we think are potentially relevant for inference on Lifted Graphical Models.

Acknowledgements

SVMJs research is supported in part by NSF grant CCF-1717134 and thanks Microsoft Research for hospitality where a part of this work was done. CR gratefully acknowledges the support of DARPA No. N6600115C4043 (SIMPLEX), No. FA87501720095 (D3M), No. FA87501220335 (XDATA), No. FA87501320039 (DEFT), DOE under

[6] Details in the full version.

[7] While the absolute value of the threshold may change depending on machine configurations, such a threshold will always exist.

Table 1: Benchmark Comparisons: The first column denotes the range of ρ, followed by the bands of the datasets (see Figure 1) and the dataset name. The fourth column denotes the number of variables/factors, followed by R_J and R_D. We report three runtimes for JoinInfer: without 0/1 projections, with all 0/1 projections and HYJAR, followed by our comparison engines – LibDAI, IJGP and ACE (Total Time and Inference Time). (All runtimes are in seconds.) Further, we report the median and mean sparsity for every dataset, followed by fractional hypertree width (fhtw) and tree-width (tw) (computed for the same GHD). The fractional hypertreewidth (fhtw) numbers were generated by solving the linear program (3) using Google OR-Tools (**?**). Finally, we report the maximum domain value (D) and maximum factor table (non-zero) entry size (N). 'T' denotes engine-time out (60 mins). The networks BN_30 − 39 are denoted by '*' since we modify the probabilities in them (details in the full version). Note that LibDAI and IJGP crash on all benchmarks where $\rho > 10^9$, due to huge pre-memory allocation and this is denoted by 'F'. For IJGP, we observed that it does approximate inference in benchmarks Munin1 and BN_43 − 46 respectively. In particular, we recorded its final treewidth using the MinFill ordering on all benchmarks and compared it with JoinInfer's and libDAI's treewidth (both using the MinFill ordering) respectively. We noticed that the final treewidth reported by IJGP was much smaller than the treewidth reported by JoinInfer and libDAI. Note that we preprocess evidence and SAT-based singleton consistency on all the benchmarks and thus, we concluded that IJGP does Approximate Inference on these datasets (which we denote by 'A').

ρ	Band	Dataset	Var/Factors	R_J	R_D	JoinInfer		HYJAR	libDAI	IJGP	ACE		Sparsity (in %)	fhtw	tw	D/N
						w/o 0/1	0/1				TTime	ITime				
$\rho > 10^9$	Band 1	CELAR6-SUB0_20	16/57	2.00E-03	1.00E-03	0.17	**0.16**	0.19	F	F	97.24	0.36	20/20	4	8	44/387
		CELAR6-SUB1_20	14/75	3.00E-04	3.00E-04	**2.57**	2.58	2.59	F	F	444.38	0.99	20/20	5	10	44/387
		CELAR6-SUB2_20	16/89	1.00E-04	1.00E-04	1.05	**1.04**	1.07	F	F	653.33	0.74	20/20	5.5	11	44/387
		CELAR6-SUB3_20	18/106	1.00E-04	1.00E-04	3.72	3.69	**3.67**	F	F	1219.08	0.78	20/20	5.5	11	44/387
		CELAR6-SUB0_40	16/57	3.00E-02	2.50E-02	4.55	4.59	**4.17**	F	F	855.12	1.2	40/40	4	8	44/774
		CELAR6-SUB1_40	14/75	1.00E-02	1.00E-02	392.7	388.76	**388.42**	F	F	T	T	40/40	5	10	44/774
		CELAR6-SUB2_40	16/89	6.50E-03	6.40E-03	449.02	448.56	**441.41**	F	F	T	T	40/40	5.5	11	44/774
		CELAR6-SUB3_40	18/106	6.00E-03	7.00E-03	796.76	794.98	**780.93**	F	F	T	T	40/40	5.5	11	44/774
	Band 2	BN_30*	1036/1153	15.96	5.10E+02	1.05	1.11	**1.01**	F	F	2.20	0.28	50/44.5	25	41	2/4
		BN_31*	1036/1153	47.47	2.00E+03	1.04	1.1	**1.03**	F	F	2.23	0.31	50/44.5	25	39	2/4
		BN_32*	1294/1441	1.88	1.20E+02	1.61	1.71	**1.60**	F	F	2.94	0.32	50/44.3	28	49	2/4
		BN_33*	1294/1441	6.20E+02	1.00E+03	1.62	1.68	**1.60**	F	F	2.86	0.31	50/44.3	26	42	2/4
		BN_34*	1294/1443	2.60E+04	3.20E+04	1.60	1.66	**1.59**	F	F	2.76	0.31	50/44.3	28	41	2/4
		BN_35*	1294/1443	3.10E+01	5.10E+02	1.59	1.65	**1.58**	F	F	2.82	0.31	50/44.3	26	43	2/4
		BN_36*	1294/1444	1.70E+03	2.00E+03	1.62	1.70	**1.60**	F	F	3.05	0.31	50/43.7	30	49	2/4
		BN_37*	1294/1444	7.10E+02	1.00E+03	1.64	1.70	**1.62**	F	F	2.76	0.30	50/43.7	30	50	2/4
		BN_38*	1294/1442	4.10E+02	2.50E+02	1.58	1.65	**1.55**	F	F	2.94	0.32	50/43.9	27	46	2/4
		BN_39*	1294/1442	10.7	2.50E+02	1.61	1.65	**1.60**	F	F	2.93	0.32	50/43.9	26	44	2/4
		BN_62	657/667	7.70E+09	8.38E+09	0.74	0.7	**0.68**	F	F	1.45	0.24	25/34.1	21	47	2/14
	Band 3	BN_60	530/539	1.20E+08	7.20E+16	**0.7**	0.75	0.73	F	F	1.65	0.24	50/44.03	29	60	2/16
		BN_61	657/667	3.00E+10	1.70E+10	0.74	0.74	**0.69**	F	F	1.69	0.24	25/34.1	21	46	2/14
		BN_63	530/540	1.40E+10	9.20E+18	2.43	0.77	**0.67**	F	F	1.84	0.27	50/43.5	30	57	2/16
		BN_64	530/540	1.60E+09	5.76E+17	0.79	0.68	**0.63**	F	F	1.84	0.25	50/43.5	28.5	55	2/16
		BN_67	430/437	4.60E+10	2.95E+20	2.64	**1.65**	2.05	F	F	1.71	0.26	50/50.57	32.5	62	2/16
$\rho \leq 10^9$	Band 4	BN_20	2433/2840	119	1.00E-04	4.53	**4.47**	14.94	22.73	T	T	T	50/49.3	4	7	91/208
		BN_21	2433/2840	109	1.00E-04	4.49	**4.42**	14.86	23.37	T	T	T	50/49.3	4	7	91/208
		BN_22	2119/2423	0.97	1.00E-05	**2.13**	2.18	2.98	3.77	T	7.83	1.71	50/47	4	7	91/208
		BN_23	2119/2423	0.97	1.00E-05	**2.14**	2.21	3	3.74	T	7.74	1.79	50/47	2	5	91/208
		BN_24	1514/1818	2.08	1.00E-05	**1.33**	1.4	1.74	2.11	T	6.24	1.58	53.8/53	2	5	91/208
		BN_25	1514/1818	2.01	1.00E-05	**1.31**	1.39	1.76	2.12	T	6.34	1.62	53.8/53	2	5	91/208
		Pathfinder	109/109	54.94	1.00E-05	0.15	0.29	0.29	**0.11**	0.34	0.81	0.31	52.4/61.4	2	7	63/6437
	Band 5	Alarm	37/37	3.58	11.39	0.03	0.03	0.05	**0.02**	0.06	0.46	0.21	100/99.4	2	5	4/108
		Hepar2	70/70	5.95	9	0.04	0.05	0.06	**0.03**	0.19	0.42	0.19	100/100	2	7	4/384
		Mildew	35/35	2.00E+03	327	0.78	0.71	**0.24**	0.27	2.81	2.91	1.89	75/61.7	3	5	100/14849
		Munin	1041/1041	35.77	557	13.27	13.82	**1.98**	3.14	11.35	T	T	42.1/46.6	6	9	21/276
		Munin1	186/186	43.23	16.6	598.86	629.75	**20.6**	39.01	A	T	T	46.2/48.6	7	12	21/276
		Munin4	1038/1038	57.9	557	16.86	17.21	2.15	**2.06**	10.67	3.77	2.01	44/46.6	6	9	21/276
		Diabetes	413/413	524.51	4.24E+06	3.28	3.31	**0.72**	0.89	32.98	6.99	4.69	33.3/45.6	4	5	21/2040
		Munin2	1003/1003	2.90E+05	8.90E+08	2.71	2.28	**0.68**	0.79	4.27	2.81	1.63	46.4/48	8	8	21/276
		Munin3	1041/1041	5.00E+05	1.20E+04	2.71	2.52	**0.70**	0.95	5.81	2.38	1.28	45.8/37	6	8	21/276
		Pigs	441/441	144	1.50E+04	0.98	0.92	0.36	**0.24**	1.05	1.37	0.7	55.6/70.2	8	11	3/15
		Link	724/724	2.00E+07	2.70E+08	18.02	19.91	14.27	**3.43**	29.73	201.02	7.49	50/65.1	12	16	4/31
		Barley	48/48	4.00E+07	6.50E+03	26.69	27.17	**1.13**	1.45	15.32	17.53	10.94	100/100	4	8	67/40320
		Hailfinder	56/56	13.02	1.00E+04	0.03	0.05	0.06	**0.01**	0.05	0.53	0.23	94.2/83.9	3	5	11/1181
		Water	32/32	1.00E+05	1.00E+06	0.17	**0.14**	0.27	0.31	0.15	0.81	0.34	50/58.23	4	11	4/1454
		Win95pts	76/76	8.66	3.10E+04	0.05	0.05	0.05	**0.03**	**0.03**	0.53	0.2	100/90	3	9	2/252
	Band 6	Andes	223/223	3.10E+04	1.50E+20	0.57	0.59	0.19	**0.14**	0.59	1.12	0.67	100/95.7	12	17	2/128
		BN_42	870/879	1.23	4.72E+21	32.63	32.11	35.15	**2.66**	19.18	1216.39	19.6	50/54.4	24	24	2/16
		BN_43	870/880	1.14	3.78E+22	65.47	64.03	**4.37**	4.43	A	1132.1	22.24	50/54.4	25	25	2/16
		BN_44	870/880	1.03	2.40E+24	227.87	216.98	133.5	**12.82**	A	1341.72	17.02	50/54.3	27	27	2/16
		BN_45	870/880	1.1	3.78E+22	67.05	68.21	8.07	**6.95**	A	778.91	19.05	50/54.2	25	25	2/16
		BN_46	489/497	1.04	7.92E+28	45.98	46.09	20.16	**5.85**	A	150.25	5.79	50/55.9	24	24	2/16

No. 108845 (Integrated Compiler and Runtime Autotuning Infrastructure for Power, Energy, and Resilience), NSF under No.1505728 (Intel/NSF CPS Security grant), NIH under No. U54EB020405 (Mobilize), ONR under No. N000141712266 (Unifying Weak Supervision), No. N00014-14-1-0102 (Data- Driven Systems: Join Algorithms and Random Network Theory), AFOSR under No. 580K753 (Mathemat- ical Foundations of Secure Computing Clouds), Moore Foundation, Okawa Research Grant, American Family Insurance, Accenture, Toshiba, Secure Internet of Things Project, Google, VMware, Qualcomm, Ericsson, Analog Devices, and members of the Stanford DAWN project: Intel, Microsoft, Teradata, and VMware. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of DARPA, DOE, NSF, NIH, ONR, or the U.S. Government. ARs research is supported in part by NSF grants CCF-1763481 and CCF-1717134.

References

Abo Khamis, M.; Ngo, H. Q.; and Rudra, A. 2016. FAQ: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 13–28. ACM.

Atserias, A.; Grohe, M.; and Marx, D. 2013. Size bounds and query plans for relational joins. *SIAM J. Comput.* 42(4):1737–1767.

Bach, F. R., and Jordan, M. I. 2001. Thin junction trees. In *Advances in Neural Information Processing Systems 14 [Neural Information Processing Systems: Natural and Synthetic, NIPS 2001, December 3-8, 2001, Vancouver, British Columbia, Canada]*, 569–576.

Bekker, J.; Davis, J.; Choi, A.; Darwiche, A.; and Van den Broeck, G. 2015. Tractable learning for complex probability queries. In *Advances in Neural Information Processing Systems*, 2242–2250.

Bilmes, J., and Dechter, R. 2006. Evaluation of probabilistic inference systems of UAI06. `http://melodi.ee.washington.edu/~bilmes/uai06InferenceEvaluation/results`.

Chavira, M., and Darwiche, A. 2005. Compiling bayesian networks with local structure. In *IJCAI*, volume 5, 1306–1312.

Chavira, M., and Darwiche, A. 2007. Compiling bayesian networks using variable elimination. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2443–2449.

Darwiche, A. 2001. Recursive conditioning. *Artif. Intell.* 126(1-2):5–41.

de Salvo Braz, R.; Amir, E.; and Roth, D. 2005. Lifted first-order probabilistic inference. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, 1319–1325.

Dechter, R. 1996. Bucket elimination: A unifying framework for probabilistic inference. In *UAI '96: Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence, Reed College, Portland, Oregon, USA, August 1-4, 1996*, 211–219.

Fischl, W.; Gottlob, G.; and Pichler, R. 2016. General and fractional hypertree decompositions: Hard and easy cases. *arXiv preprint arXiv:1611.01090*.

Gottlob, G.; Grohe, M.; Musliu, N.; Samer, M.; and Scarcello, F. 2005. Hypertree decompositions: Structure, algorithms, and applications. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, 1–15. Springer.

Huang, J.; Chavira, M.; and Darwiche, A. 2006. Solving MAP exactly by searching on compiled arithmetic circuits. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, 1143–1148.

Jensen, F.; Lauritzen, S.; and Olesen, K. 1990. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly* 4:269–282.

Joglekar, M. R.; Puttagunta, R.; and Ré, C. 2016. AJAR: aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, 91–106.

Kask, K.; Dechter, R.; Larrosa, J.; and Dechter, A. 2005. Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence* 166(1-2):165–193.

Kersting, K. 2012. Lifted probabilistic inference. In *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, 33–38.

Koller, D., and Friedman, N. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

Larkin, D., and Dechter, R. 2003. Bayesian inference in the presence of determinism. In *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics, AISTATS 2003, Key West, Florida, USA, January 3-6, 2003*.

Mateescu, R.; Kask, K.; Gogate, V.; and Dechter, R. 2010. Join-graph propagation algorithms. *J. Artif. Intell. Res. (JAIR)* 37:279–328.

Milch, B.; Zettlemoyer, L. S.; Kersting, K.; Haimes, M.; and Kaelbling, L. P. 2008. Lifted probabilistic inference with counting formulas. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 1062–1068.

Mooij, J. M. 2010. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research* 11:2169–2173.

Ngo, H. Q.; Porat, E.; Ré, C.; and Rudra, A. 2012. Worst-case optimal join algorithms:[extended abstract]. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, 37–48. ACM.

Ngo, H. Q.; Ré, C.; and Rudra, A. 2014. Skew strikes back: New developments in the theory of join algorithms. *ACM SIGMOD Record* 42(4):5–16.

Pearl, J. 1989. *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann.

Poole, D. L., and Zhang, N. L. 2003. Exploiting contextual independence in probabilistic inference. *J. Artif. Intell. Res. (JAIR)* 18:263–313.

Zhang, N. L., and Poole, D. L. 1996. Exploiting causal independence in bayesian network inference. *J. Artif. Intell. Res. (JAIR)* 5:301–328.