Exploiting Shared Correlations in Probabilistic Databases

Prithviraj Sen
sen@cs.umd.edu

Amol Deshpande
amol@cs.umd.edu

Lise Getoor
getoor@cs.umd.edu

Computer Science Department, University of Maryland, College Park 20742, MD, USA.

ABSTRACT

There has been a recent surge in work in probabilistic databases, propelled in large part by the huge increase in noisy data sources — from sensor data, experimental data, data from uncurated sources, and many others. There is a growing need for database management systems that can efficiently represent and query such data. In this work, we show how data characteristics can be leveraged to make the query evaluation process more efficient. In particular, we exploit what we refer to as *shared correlations* where the same uncertainties and correlations occur repeatedly in the data. Shared correlations occur mainly due to two reasons: (1) Uncertainty and correlations usually come from general statistics and rarely vary on a tuple-to-tuple basis; (2) The query evaluation procedure itself tends to re-introduce the same correlations. Prior work has shown that the query evaluation problem on probabilistic databases is equivalent to a probabilistic inference problem on an appropriately constructed probabilistic graphical model (PGM). We leverage this by introducing a new data structure, called the *random variable elimination graph* (rv-elim graph) that can be built from the PGM obtained from query evaluation. We develop techniques based on bisimulation that can be used to compress the rv-elim graph exploiting the presence of shared correlations in the PGM, the compressed rv-elim graph can then be used to run inference. We validate our methods by evaluating them empirically and show that even with a few shared correlations significant speed-ups are possible.

1. INTRODUCTION

Many real-world applications produce large amounts of uncertain data and there is a need for systems that can store, retrieve and query such data. Traditional database systems are geared towards storing exact data and are not suited for storing uncertain data. Probabilistic databases, on the other hand, are designed to handle uncertainty expressed in terms of probabilities and provide an attractive option to store data with uncertainty. Applications for probabilistic databases include information retrieval [6, 24], recommendation systems [21, 22], mobile object data management [5], information extraction [15], data integration [1] and data management for sensor networks [9].

Probabilistic databases based on possible worlds semantics present a particularly attractive brand of probabilistic databases because of their intuitive query semantics, and a lot of recent work has been devoted to these formulations [6, 22, 3, 24, 7, 26]. It has been shown that query processing in probabilistic databases based on possible worlds semantics can be reduced to probabilistic inference problems in probabilistic graphical models (PGMs) [24]. At a broad level of abstraction, this observation suggests that a query processor for probabilistic databases may consist of two components:

1. Take the user-submitted query and the uncertain data in the database to construct a PGM.

2. Run probabilistic inference on the constructed PGM to compute the result.

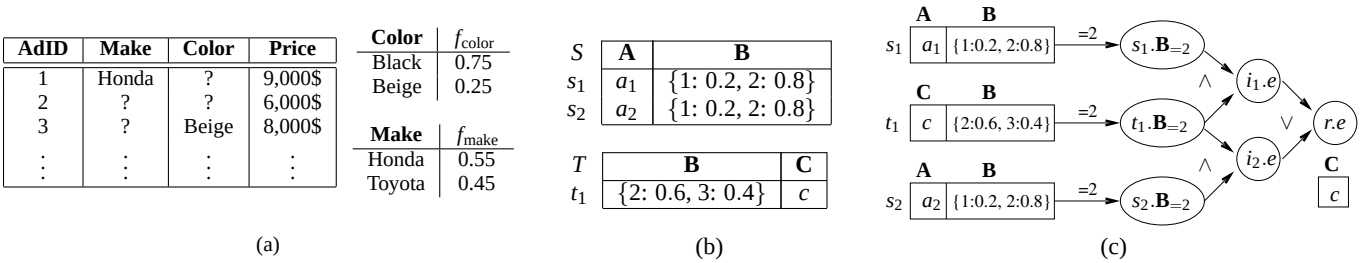
However, even for queries that lead to easy* probabilistic inference problems, there is still a large gap between the time required to construct the PGM and the time spent to run inference, with time spent to run inference dominating the overall runtime by a fair margin in most cases.

We aim to bridge this gap by leveraging on special properties of the uncertain data at hand to reduce the complexity of inference during query evaluation for probabilistic databases. One such property is the presence of *shared correlations* or *shared factors*. Consider the small probabilistic relation shown in Figure 1 (a) containing pre-owned car ads. The first tuple shows an ad with the **Color** of the car missing, the third tuple shows one with the **Make** missing and the second tuple represents an ad with both attributes missing. Figure 1 (a) also shows the probability distributions associated with these missing values, more specifically, f_{make} defines the distribution over missing **Make** values in the database (assuming our universe can contain only two makes Honda and Toyota) and f_{color} defines the distribution over missing **Color** values (assuming our universe contains only black and beige cars). Note that the distributions make no reference to any tuple specific information. In other words, no matter how many tuples with missing **Color** are present in the relation, their uncertainty will still be defined by the same distribution represented by f_{color} and along with f_{make}, these distributions are examples of *shared correlations*.

Such shared correlations are ubiquitious in practice, and arise because, in most cases, the uncertainty in the data is defined using general statistics that *do not* vary on a per-tuple basis. Various earlier works have also described applications with shared correlations. For instance, Andritsos et al. [1] describe a customer relationship management application where the objective is to merge

*Even though inference is #P-complete in general [6], for graphical models with low treewidth it is still feasible [23].

PVLDB '08, August 23-28, 2008, Auckland, New Zealand



(a) (b) (c)

Figure 1: (a) Pre-owned car ads with missing values. (b) A 2-relation database with uncertainty and (c) evaluating $\prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$ on it.

data from two or more source databases and each source database is assigned a probability value based on the quality of the information it contains. Even here, probabilities don't change from tuple to tuple, since tuples from the same source are assigned the same source probability.

Another source of shared correlations in probabilistic databases is the query evaluation approach itself. As we showed in our earlier work [24], constructing the PGM corresponding to a query to be evaluated over a probabilistic database requires the introduction of small factors that depict probability distributions and correlations introduced on the fly. For instance, if tuples t and t' join to produce join tuple r, then we need to introduce a factor that encodes the correlation that r exists iff both t and t' exist (an $\wedge$-factor). More importantly, such a factor is introduced whenever *any* pair of tuples join, thus leading to repeated copies of the same $\wedge$-factor, thus introducing additional shared correlations. Our aim, in this paper, is to exploit such shared correlations to make exact probability computation for query evaluation in probabilistic databases more efficient.

Most probabilistic database formulations require general probabilistic inference at some level of abstraction. Several query evaluation approaches construct boolean formulas (sometimes called lineage) that can be seen as special cases of PGMs that we construct [12, 22, 7]. The techniques and ideas we develop in this paper should be of use to the above mentioned works. Our general goal is to go beyond the currently known set of queries that can be efficiently evaluated (e.g., safe plans [6]). Given that we already know general inference is a #P-complete problem [6], the only way we can achieve this is to utilize the special properties of the data at hand, shared correlations being one such property. We make the following contributions in this paper:

- We introduce the concept of shared correlations and shared factors using simple motivating examples and enrich our probabilistic database model to explicitly represent them.
- We introduce a novel graph-based data structure, referred to as the *rv-elim* graph, that helps identify shared factors in the PGM, and we show how to construct it from the PGM.
- We develop an approach based on bisimulation [17] that takes an rv-elim graph and compresses it using its shared factors.
- We show how to perform inference efficiently on the compressed rv-elim graph.
- We validate our approach empirically and show that significant speedups are possible even in the presence of just a few shared factors.

The rest of the paper is organized as follows: In the next section, we introduce a motivating example that shows how standard inference algorithms fail to exploit shared correlations; in Section 3 we review definitions and introduce the notation which will be used for the rest of the paper; Section 4 and Section 5 describe our approach to inference with shared correlations; in Section 6 we describe our experimental results; in Section 7 we discuss related work and finally, we conclude with Section 8.

2. MOTIVATING EXAMPLE

We start with a simple motivating example and describe how attribute and tuple uncertainty are represented and illustrate how, for a given query, we can construct a PGM to compute the results.

Consider the small two-relation database shown in Figure 1 (b) where every tuple has an uncertain attribute value. We denote an uncertain attribute value by its domain where each entry in the domain is followed by the probability with which the attribute value can take the assignment, and these are, essentially, random variables. For instance, $s_2.\mathbf{B}$ can be assigned the value 1 with probability 0.2 and the value 2 with probability 0.8. We represent such uncertain information using small functions over the corresponding random variables which we refer to as factors (we define all terms introduced here formally in the next section). For instance, for the three random variables in Figure 1 (b) we would define factors $f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$, $f_{s_2.\mathbf{B}}(s_2.\mathbf{B})$ and $f_{t_1.\mathbf{B}}(t_1.\mathbf{B})$ as follows:

$s_1.\mathbf{B}$	$f_{s_1.\mathbf{B}}$
1	0.2
2	0.8

$s_2.\mathbf{B}$	$f_{s_2.\mathbf{B}}$
1	0.2
2	0.8

$t_1.\mathbf{B}$	$f_{t_1.\mathbf{B}}$
2	0.6
3	0.4

In addition to the random variables that denote the potential attribute values, we can introduce tuple *existence* random variables $s_1.e$, $s_2.e$, and $t_1.e$, to capture tuple uncertainty. These are boolean-valued random variables and can have associated factors. In the example, we assume the tuples are certain, so we don't show the existence random variables for the base tuples.

A probabilistic database with uncertainty represents a probability distribution over posssible databases (also refered to as *possible worlds*). The probability associated with a possible world is the probability of the joint assignment of the attribute value variables and tuple existence random variables that is consistent with the possible world. For instance, the probabilistic database shown in Figure 1 (b) represents a distribution over $2^3 = 8$ possible worlds and each possible world's probability can be obtained by extracting the appropriate probabilities from the factors, multiplying them together and normalizing if necessary. For instance, for the possible world obtained by the assignment $s_1.\mathbf{B} = 1$, $s_2.\mathbf{B} = 2$, $t_1.\mathbf{B} = 2$ the probability is $0.2 \times 0.8 \times 0.6 = 0.096$.

The result of evaluating a user-submitted query (expressed in some standard query language such as relational algebra) over such a probabilistic database is defined as the collection of result tuples produced by running the query against all possible worlds, and the probability of each result tuple is simply the sum of probabilities of the possible worlds that return the tuple as a result. This is better explained through an example. Suppose we want to execute the

$$\mu(r.e) = \sum_{\mathcal{O}} f_{r.e}(r.e, i_1.e, i_2.e)\, f_{i_1.e}(i_1.e, s_1.\mathbf{B}_{=2}, t_1.\mathbf{B}_{=2})\, f_{i_2.e}(i_2.e, s_2.\mathbf{B}_{=2}, t_1.\mathbf{B}_{=2})\, f_{t_1.\mathbf{B}_{=2}}(t_1.\mathbf{B}_{=2}, t_1.\mathbf{B})$$
$$f_{s_1.\mathbf{B}_{=2}}(s_1.\mathbf{B}_{=2}, s_1.\mathbf{B})\, f_{s_2.\mathbf{B}_{=2}}(s_2.\mathbf{B}_{=2}, s_2.\mathbf{B})\, f_{t_1.\mathbf{B}}(t_1.\mathbf{B})\, f_{s_1.\mathbf{B}}(s_1.\mathbf{B})\, f_{s_2.\mathbf{B}}(s_2.\mathbf{B})$$

$$= \sum_{\mathcal{O}\setminus\{s_1.\mathbf{B}, s_2.\mathbf{B}\}} f_{r.e}(r.e, i_1.e, i_2.e)\, f_{i_1.e}(i_1.e, s_1.\mathbf{B}_{=2}, t_1.\mathbf{B}_{=2})\, f_{i_2.e}(i_2.e, s_2.\mathbf{B}_{=2}, t_1.\mathbf{B}_{=2})\, f_{t_1.\mathbf{B}_{=2}}(t_1.\mathbf{B}_{=2}, t_1.\mathbf{B})\, f_{t_1.\mathbf{B}}(t_1.\mathbf{B})$$
$$\underbrace{\sum_{s_1.\mathbf{B}} f_{s_1.\mathbf{B}}(s_1.\mathbf{B})\, f_{s_1.\mathbf{B}_{=2}}(s_1.\mathbf{B}_{=2}, s_1.\mathbf{B})}_{m_{s_1.\mathbf{B}}(s_1.\mathbf{B}_{=2})} \underbrace{\sum_{s_2.\mathbf{B}} f_{s_2.\mathbf{B}}(s_2.\mathbf{B})\, f_{s_2.\mathbf{B}_{=2}}(s_2.\mathbf{B}_{=2}, s_2.\mathbf{B})}_{m_{s_2.\mathbf{B}}(s_2.\mathbf{B}_{=2})}$$

$$= \sum_{\mathcal{O}\setminus\{s_1.\mathbf{B}_{=2}, s_2.\mathbf{B}_{=2}, s_1.\mathbf{B}, s_2.\mathbf{B}\}} f_{r.e}(r.e, i_1.e, i_2.e)\, f_{t_1.\mathbf{B}_{=2}}(t_1.\mathbf{B}_{=2}, t_1.\mathbf{B})\, f_{t_1.\mathbf{B}}(t_1.\mathbf{B})$$
$$\underbrace{\sum_{s_1.\mathbf{B}_{=2}} f_{i_1.e}(i_1.e, s_1.\mathbf{B}_{=2}, t_1.\mathbf{B}_{=2}) m_{s_1.\mathbf{B}}(s_1.\mathbf{B}_{=2})}_{m_{s_1.\mathbf{B}_{=2}}(i_1.e, t_1.\mathbf{B}_{=2})} \underbrace{\sum_{s_2.\mathbf{B}_{=2}} f_{i_2.e}(i_2.e, s_2.\mathbf{B}_{=2}, t_1.\mathbf{B}_{=2}) m_{s_2.\mathbf{B}}(s_2.\mathbf{B}_{=2})}_{m_{s_2.\mathbf{B}_{=2}}(i_2.e, t_1.\mathbf{B}_{=2})}$$

Figure 2: How variable elimination proceeds to solve the query evaluated on the database shown in Figure 1 (b). $\mathcal{O}$ denotes the elimination order used $\{s_1.\mathbf{B}, s_2.\mathbf{B}, s_1.\mathbf{B}_{=2}, s_2.\mathbf{B}_{=2}, t_1.\mathbf{B}, i_1.e, i_2.e, t_1.\mathbf{B}_{=2}\}$.

query $\prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$ on the database in Figure 1 (b). The result is the single tuple $r = \boxed{\mathbf{C}}{\boxed{c}}$, since each possible world either returns this result tuple or not. Among all the possible worlds, the ones that return the result tuple have at least one of $s_1.\mathbf{B}$ or $s_2.\mathbf{B}$ assigned to 2 in addition to having $t_1.\mathbf{B}$ assigned to 2 (otherwise the join will be empty). Of the 8 possible worlds, the ones that satisfy this property correspond to the following joint assignments:

- $s_1.\mathbf{B} = 2$, $s_2.\mathbf{B} = 1$, $t_1.\mathbf{B} = 2$ (prob. = 0.096)
- $s_1.\mathbf{B} = 1$, $s_2.\mathbf{B} = 2$, $t_1.\mathbf{B} = 2$ (prob. = 0.096)
- $s_1.\mathbf{B} = 2$, $s_2.\mathbf{B} = 2$, $t_1.\mathbf{B} = 2$ (prob. = 0.384)

Thus the probability associated with the result tuple is $0.096 + 0.096 + 0.384 = 0.576$.

Unfortunately, since the number of possible worlds is proportional to the product of the domain sizes of all the random variables contained in the database, evaluating queries directly using the above definition will be feasible only for the smallest of probabilistic databases. One way to get around this issue is to cast the query evaluation problem as a probabilistic inference problem. In order to achieve this we need to introduce new random variables and factors which capture the dependencies among the intermediate tuples and random variables produced while evaluating the query [24]. For instance, to solve the above query using this approach we would have to perform the following operations:

- Notice that the common entry in the domains of the join attribute values ($S.\mathbf{B}$ and $T.\mathbf{B}$) is 2; hence only those possible worlds where $S.\mathbf{B}$=2 and $T.\mathbf{B}$=2 can produce an intermediate join tuple. We capture this by introducing three intermediate random variables $s_1.\mathbf{B}_{=2}$, $t_1.\mathbf{B}_{=2}$ and $s_2.\mathbf{B}_{=2}$ each of which is boolean-valued and assigned the value true when the corresponding base tuple attribute value is assigned the value 2. We enforce these equalities by introducing factors such as:

$$f_{s_1.\mathbf{B}_{=2}}(s_1.\mathbf{B}_{=2}, s_1.\mathbf{B}) = \begin{cases} 1 & \text{if } s_1.\mathbf{B}_{=2} \Leftrightarrow (s_1.\mathbf{B} == 2) \\ 0 & \text{otherwise} \end{cases}$$

  In essence, this factor says that $s_1.\mathbf{B}_{=2}$ is true iff $s_1.\mathbf{B}$ is assigned the value 2. Similarly, $f_{s_2.\mathbf{B}_{=2}}(s_2.\mathbf{B}_{=2}, s_2.\mathbf{B})$ and $f_{t_1.\mathbf{B}_{=2}}(t_1.\mathbf{B}_{=2}, t_1.\mathbf{B})$ are introduced for the other two intermediate random variables.

- Next we introduce two intermediate tuples, i_1 (resulting from the join between s_1 and t_1), and i_2 (from the join between s_2 and t_1). These join tuples are not produced by every possible world, and we will capture their existence uncertainty using two boolean-valued existence random variables $i_1.e$ and $i_2.e$ respectively. Having introduced the three intermediate random variables in the previous step, it is now easy to describe the conditions when i_1 and i_2 exist. i_1 is produced, or $i_1.e$ is true, in only those possible worlds where both $s_1.\mathbf{B}_{=2}$ and $t_1.\mathbf{B}_{=2}$ are true. Similarly, $i_2.e$ is true only when boh $s_2.\mathbf{B}_{=2}$ and $t_1.\mathbf{B}_{=2}$ are true. We capture these correlations by introducing two factors, for $k = 1, 2$:

$$f_{i_k.e}(i_k.e, s_k.\mathbf{B}_{=2}, t_1.\mathbf{B}_{=2}) = \begin{cases} 1 & \text{if } i_k.e \Leftrightarrow s_k.\mathbf{B}_{=2} \wedge t_1.\mathbf{B}_{=2} \\ 0 & \text{otherwise} \end{cases}$$

- Finally, we introduce a random variable to capture the existence uncertainty of the result tuple r. r exists, or $r.e$ is true, in only those possible worlds where either $i_1.e$ or $i_2.e$ or both are true. This correlation is captured by introducing the factor:

$$f_{r.e}(r.e, i_1.e, i_2.e) = \begin{cases} 1 & \text{if } r.e \Leftrightarrow i_1.e \vee i_2.e \\ 0 & \text{otherwise} \end{cases}$$

The query evaluation procedure, along with the intermediate random variables and factors introduced, is shown in Figure 1 (c).

Now, to compute the result of the query all we need to do is to compute the probability of $r.e$. We can do this by multiplying all the factors ($f_{s_1.\mathbf{B}}$, $f_{s_2.\mathbf{B}}$, $f_{t_1.\mathbf{B}}$ from the database and the ones introduced above) and summing over all random variables for $r.e$ using any standard probabilistic inference algorithm. This can help avoid the prohibitive cost of summing over possible worlds.

Variable Elimination: One possible inference algorithm that we can use for this purpose is variable elimination (VE) [27]. VE runs by first choosing an elimination order which specifies the order in which to sum over (eliminate) the random variables. It then repeatedly picks the next random variable from the order, pushes the

corresponding summation as far into the product of factors as possible, sums it out and proceeds in this fashion. In Figure 2 we show the first few steps of how VE would proceed when used to compute the probability of $r.e$ using the elimination order $\mathcal{O} = \{s_1.\mathbf{B}, s_2.\mathbf{B}, s_1.\mathbf{B}_{=2}, s_2.\mathbf{B}_{=2}, t_1.\mathbf{B}, i_1.e, i_2.e, t_1.\mathbf{B}_{=2}\}$ (variables are eliminated left to right). For instance in Figure 2, to eliminate $s_1.\mathbf{B}$ we first multiply factors $f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$ and $f_{s_1.\mathbf{B}_{=2}}(s_1.\mathbf{B}_{=2}, s_1.\mathbf{B})$, and then sum out random variable $s_1.\mathbf{B}$ to produce the new factor $m_{s_1.\mathbf{B}}(s_1.\mathbf{B}_{=2})$.

$m_{s_1.\mathbf{B}_{=2}}$
args.: $i_1.e, t_1.\mathbf{B}_{=2}$
func.: $\begin{cases} T,T \rightarrow 0.8 \\ T,F \rightarrow 0.0 \\ F,T \rightarrow 0.2 \\ F,F \rightarrow 1.0 \end{cases}$

$m_{s_2.\mathbf{B}_{=2}}$
args.: $i_2.e, t_1.\mathbf{B}_{=2}$
func.: $\begin{cases} T,T \rightarrow 0.8 \\ T,F \rightarrow 0.0 \\ F,T \rightarrow 0.2 \\ F,F \rightarrow 1.0 \end{cases}$

Figure 3: Pair of shared factors (T, F denote *True*, *False*, resp.).

2.1 Limitations of Naive Inference Algorithms

The main issue with VE (or any other standard probabilistic inference algorithm) is that it does not exploit shared correlations. For instance, in Figure 2, in the process of computing the probabilities for $r.e$ we produce intermediate factors $m_{s_1.\mathbf{B}_{=2}}(i_1.e,\ t_1.\mathbf{B}_{=2})$ and $m_{s_2.\mathbf{B}_{=2}}(i_2.e, t_1.\mathbf{B}_{=2})$. If we take a closer look at both of these factors, we notice that they both map exactly the same inputs to the same outputs:

$i_1.e$	$t_1.\mathbf{B}_{=2}$	$m_{s_1.\mathbf{B}_{=2}}$
True	*True*	0.8
True	*False*	0
False	*True*	0.2
False	*False*	1

$i_2.e$	$t_1.\mathbf{B}_{=2}$	$m_{s_2.\mathbf{B}_{=2}}$
True	*True*	0.8
True	*False*	0
False	*True*	0.2
False	*False*	1

This indicates that we went through the exact same multiplication and summation steps to compute both $m_{s_1.\mathbf{B}_{=2}}(i_1.e, t_1.\mathbf{B}_{=2})$ and $m_{s_2.\mathbf{B}_{=2}}(i_2.e, t_1.\mathbf{B}_{=2})$. In fact, these are *shared factors* (which will be defined more precisely in the next section), and this repeated computation is what we will be trying to avoid. $m_{s_1.\mathbf{B}}(s_1.\mathbf{B}_{=2})$ and $m_{s_2.\mathbf{B}}(s_2.\mathbf{B}_{=2})$ are also shared factors, and we are performing redundant computations in this case as well.

In hindsight, it is not really surprising that $m_{s_1.\mathbf{B}}(s_1.\mathbf{B}_{=2})$ and $m_{s_2.\mathbf{B}}(s_2.\mathbf{B}_{=2})$ turned out to be virtual copies of each other. Let us take a closer look at these factors:

- $m_{s_1.\mathbf{B}}(s_1.\mathbf{B}_{=2})$ was computed by multiplying $f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$ with $f_{s_1.\mathbf{B}_{=2}}(s_1.\mathbf{B}_{=2}, s_1.\mathbf{B})$ followed by a summation operation,
- $m_{s_2.\mathbf{B}}(s_2.\mathbf{B}_{=2})$ was computed by multiplying $f_{s_2.\mathbf{B}}(s_2.\mathbf{B})$ with $f_{s_2.\mathbf{B}_{=2}}(s_2.\mathbf{B}_{=2}, s_2.\mathbf{B})$ followed by a summation operation.

Now, $f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$ and $f_{s_2.\mathbf{B}}(s_2.\mathbf{B})$, and $f_{s_1.\mathbf{B}_{=2}}(s_1.\mathbf{B}_{=2}, s_1.\mathbf{B})$ and $f_{s_2.\mathbf{B}_{=2}}(s_2.\mathbf{B}_{=2}, s_2.\mathbf{B})$ were pairs of shared factors. It is a similar situation with $m_{s_1.\mathbf{B}_{=2}}(i_1.e, t_1.\mathbf{B}_{=2})$ and $m_{s_2.\mathbf{B}_{=2}}(i_2.e, t_1.\mathbf{B}_{=2})$, the factors involved in the generation of these intermediate factors were pairs of shared factors thus giving rise to more shared factors. We need to recognize and take advantage of such symmetry *before we actually compute these shared factors* and, in Section 4, we develop an approach to do so. In the next section, we review some definitions and introduce the notation we need for the rest of the paper.

3. PRELIMINARIES

Let X denote a random variable and $dom(X)$ denote its domain.

DEFINITION 3.1. *A* factor $f(\mathbf{X})$ *is a function over a (small) set of random variables* $\mathbf{X} = \{X_1, \ldots X_m\}$ *such that* $f(\mathbf{x}) \geq 0,\ \forall \mathbf{x} \in dom(X_1) \times \ldots \times dom(X_m)$.

DEFINITION 3.2. *A* probabilistic graphical model *(PGM)* $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$ *defines a joint distribution over the set of random variables* $\mathcal{X}$ *via a set of factors* $\mathcal{F}$*, where* $\forall f(\mathbf{X}) \in \mathcal{F},\ \mathbf{X} \subseteq \mathcal{X}$*. Given a complete joint assignment* $\mathbf{x} \in \times_{X \in \mathcal{X}} dom(X)$*, the joint distribution is defined by* $P(\mathbf{x}) = \frac{1}{\mathcal{Z}} \prod_{f \in \mathcal{F}} f(\mathbf{x}_f)$ *where* $\mathbf{x}_f$ *denotes the assignments restricted to arguments of* f *and* $\mathcal{Z} = \sum_{\mathbf{x}'} \prod_{f \in \mathcal{F}} f(\mathbf{x}'_f)$[†].

Given a PGM, it is useful to define the following operation:

DEFINITION 3.3. *Given a PGM* $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$ *and a random variable* $X \in \mathcal{X}$*, the* marginal probability *associated with the assignment* $X = x$*, where* $x \in dom(X)$*, is defined as* $\mu(x) = \sum_{\mathbf{x} \sim x} P(\mathbf{x})$*, where* $P(\mathbf{x})$ *denotes the distribution defined by the PGM and* $\mathbf{x} \sim x$ *denotes a joint assignment to* $\mathcal{X}$ *where* X *is assigned* x*.*

In this paper, we are interested in computing marginal probability distributions of random variables from a PGM that contains shared factors and for this we need to take a closer look at the definition of a factor (Definition 3.1). A factor consists of two distinct parts: (1) the list of random variables it takes as arguments, and (2) the function that maps input assignments to outputs. Thus, it may be possible for two factors f_1 and f_2 to have different arguments lists but use the same function to map inputs to outputs.

DEFINITION 3.4. *Two factors* f_1 *and* f_2 *are* shared factors*, denoted* $f_1 \cong f_2$*, if they both consist of the same function component.*

Figure 3 shows two factors from the previous section where we have clearly separated their arguments and function components.

Let R denote a probabilistic relation or simply, relation, and let $attr(R)$ denote the set of attributes of R. A relation R consists of a set of probabilistic tuples or simply, tuples, each of which is a mapping from $attr(R)$ to random variables. Let $t.a$ denote the random variable of tuple $t \in R$ such that $a \in attr(R)$. Besides mapping each attribute to a random variable, every tuple t is also associated with a boolean-valued random variable which captures the existence uncertainty of t and we denote this by $t.e$.

DEFINITION 3.5. *A* probabilistic database *or simply, a* database*,* $\mathcal{D}$ *is a pair* $\langle \mathcal{R}, \mathcal{P} \rangle$ *where* $\mathcal{R}$ *is a set of relations and* $\mathcal{P}$ *denotes a PGM defined over the set of random variables associated with the tuples in* $\mathcal{R}$*.*

Let $\mathcal{X}$ denote the set of random variables associated with database $\mathcal{D} = \langle \mathcal{R}, \mathcal{P} \rangle$. As we indicated in the previous section, $\mathcal{D}$ is simply a distribution over deterministic databases or possible worlds each of which is obtained by assigning $\mathcal{X}$ a joint assignment $\mathbf{x} \in \times_{X \in \mathcal{X}} dom(X)$[‡]. The probability associated with the possible world

[†]Note that since we allow factors to return 0, technically, there is a possibility of $\mathcal{Z}$ being 0. This only happens when we are dealing with a PGM $\mathcal{P}$ that encodes the trivial joint probability distribution which maps all joint assignments to 0. As long as there exists at least one joint assignment $\mathbf{x}$ such that $\prod_{f \in \mathcal{F}} f(\mathbf{x}_f) > 0$ this case should not arise.

[‡]Note that not all joint assignments are legal. A legal joint assignment should satisfy: $\overline{t.e} \Rightarrow (t.a = \emptyset),\ \forall t \in R, \forall a \in attr(R), \forall R \in \mathcal{R}$ where $\mathcal{R}$ denotes the set of relations in $\mathcal{D}$ and $\emptyset$ is a special "null" assignment. In other words a tuple's attributes cannot be assigned values unless it exists. It is easy to define the factors in such a way that all illegal assignments are assigned 0 probabilities.

obtained from the joint assignment **x** is given by the distribution defined by the PGM $\mathcal{P}$.

Given a user-submitted query q the query result is defined to be the set of all results obtained by running q against each possible world augmented with the corresponding possible world's probability. As we discussed earlier, executing this definition directly will be feasible only for the smallest of databases since the number of possible worlds is the product of all domain sizes of the random variables associated with the database. One way to circumvent this problem and (possibly) alleviate the computational complexity is to return only marginal probabilities associated with the random variables of the result tuples appearing in the query result [6]. One way to achieve this is to augment the PGM associated with the database by introducing new factors as we showed in the previous section when we introduced new factors to evaluate the query $q = \prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$. In general, to answer a query q on a database $\mathcal{D}$ it is possible to build a (augmented) PGM on the fly and compute the marginal probabilities of random variables associated with the relevant result tuples from it[§]. Speeding up this computation by exploiting shared factors is the subject of this paper.

4. INFERENCE WITH SHARED FACTORS

We assume that we are given a random variable X whose marginal probabilities need to be computed from a PGM $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$ constructed by running a query on a database. We also assume that every $f \in \mathcal{F}$ is associated with an id denoted by $\mathrm{id}(f)$ such that for any pair of factors $\mathrm{id}(f_1) = \mathrm{id}(f_2) \Leftrightarrow f_1 \cong f_2$.

The basic idea behind our approach is to represent a run of the inference algorithm explicitly as a labeled graph. Once we do that, we then show that it is possible to examine the graph and identify the shared intermediate factors that are generated during the inference process. To explain our approach, we first define the semantics associated with the edges of the labeled graph by introducing an operator that forms the basis of most exact probabilistic inference algorithms (e.g., variable elimination [27] and junction tree algorithm [16]).

4.1 The ELIMRV Operator

The *elimrv* operator (which stands for ELIMinate a Random Variable) is the basic operator that is used repeatedly while running inference to compute marginal probabilities. It essentially takes a random variable Y and a collection of factors **F** each of which involves Y as an argument and sums Y out from the product of all factors in **F** to return a new factor. We denote the resulting (intermediate) factor produced by m_Y followed by its list of arguments, if they are not clear from the context. For instance, when we were computing $\mu(r.e)$ for the example in Section 2, to sum over $s_2.\mathbf{B}_{=2}$ we had to first multiply the collection of factors formed by $f_{i_2.e}(i_2.e, s_2.\mathbf{B}_{=2}, t_1.\mathbf{B}_{=2})$ and $m_{s_2.\mathbf{B}}(s_2.\mathbf{B}_{=2})$ and then sum over $s_2.\mathbf{B}_{=2}$ from the product to produce the new intermediate factor $m_{s_2.\mathbf{B}_{=2}}(i_2.e, t_1.\mathbf{B}_{=2})$. Note that **F** may contain intermediate factors produced by earlier applications of the elimrv operator.

We first note a few properties about elimrv operator. The order in which the factors appear in **F** is important. For instance, suppose we want to sum over X_2 from the collection formed by $f_a(X_1, X_2)$ and $f_b(X_2, X_3)$. Then we would produce the product $f_c(X_1, X_2, X_3)$ and perform the summation to produce $f_d(X_1, X_3)$. In other words, there is an implicit assumption of ordering the arguments in the product by scanning the arguments of the input factors from left to right and this affects the resulting factor produced after the summation operation. If instead, we had multiplied $f_b(X_2, X_3)$ and $f_a(X_1, X_2)$, then we would first produce a factor $f'_c(X_2, X_3, X_1)$ and then produce $f'_d(X_3, X_1)$ after the summation. In addition, the way the arguments overlap across the input factors (in the above case, the second argument of f_a overlaps with the first argument of f_b) and the position of the argument that is being summed over also matter. We would like to make these points about the elimrv operator clear, and for this purpose, we feed the operator an explicit label that specifies the above described information.

EXAMPLE 4.1. *For the examples that follow we use the following simple format for constructing labels that specify the argument order, how the arguments overlap and which argument is being summed over. We begin by choosing three distinct special symbols, we will use #, * and %. For each* elimrv *operation, we go through the list of factors in* **F** *assigning each argument a unique id if it hasn't been seen before. Then we construct the label by traversing the list of factors again, writing the id of the argument that appears separated by # (the first symbol), separating the lists of arguments of each factor by * (the second symbol) and finally, appending a % (the third symbol) followed by the id of the argument being summed over. For the above example involving X_2, $f_a(X_1, X_2)$ and $f_b(X_2, X_3)$, the label turns out to be 1#2*2#3%2 using this format.*

We can now define the elimrv operator as follows:

DEFINITION 4.2. *The* elimrv$(Y, \mathbf{F}, l)$ *operator takes as input a random variable Y, an ordered list of factors* **F** *and a label l, and computes a new factor $\sum_Y \prod_{f \in \mathbf{F}} f$ according to the label l.*

Variable Elimination: The variable elimination inference algorithm (VE) can now be seen as applying a sequence of elimrv operations. Essentially, VE begins by collecting all factors from $\mathcal{F}$ in a pool and repeatedly applying the elimrv operator to sum over a random variable picked from an elimination order. Each time we pick a random variable Y to eliminate, we collect all factors that include Y as an argument from the pool, perform the corresponding elimrv operation, add the resulting intermediate factor m_Y back to the pool, and continue in the same fashion until we have exhausted all random variables.

4.2 The RV-ELIM Graph

For the purposes of introducing our graph-based data structure, we will assume that we are given, besides X and $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$, an elimination order $\mathcal{O}$ that contains all random variables involved in $\mathcal{X}$ except for X. In the next section (Section 5), we discuss in detail how to construct such an elimination order that suits our purposes. The *rv-elim* graph (which stands for Random Variable ELIMination graph) essentially encodes the sequence of elimrv operations encountered during the run of inference using a labeled graph.

DEFINITION 4.3. *The* rv-elim *graph $G = (V, E)$ is a directed graph with vertex labels $l(v), \forall v \in V$, and edge labels $l(e), \forall e \in E$, that represents a run of inference on a PGM $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$ according to elimination order $\mathcal{O}$ such that:*

- *Every $v \in V$ represents a factor. If v is a root, then it represents a factor from $\mathcal{F}$ and $l(v) = id(f)$; if v is not a root then it represents an intermediate factor $m_Y =$*elimrv$(Y, \mathbf{F}, l)$ *produced during the run of inference and $l(v) = l$.*
- *For each $m_Y =$* elimrv$(Y, \mathbf{F}, l)$ *produced during inference, for the i^{th} factor in* **F***, we add an edge $v_f \xrightarrow{i} v_{m_Y}$ where v_f denotes the vertex corresponding to f and v_{m_Y} denotes the vertex corresponding to m_Y, and i is the label on the edge.*

[§]Exactly what factors need to be introduced to evaluate q depends on the operators in q. In earlier work [24], we describe the necessary steps in detail, including aggregation operators, for tuple-uncertainty databases.

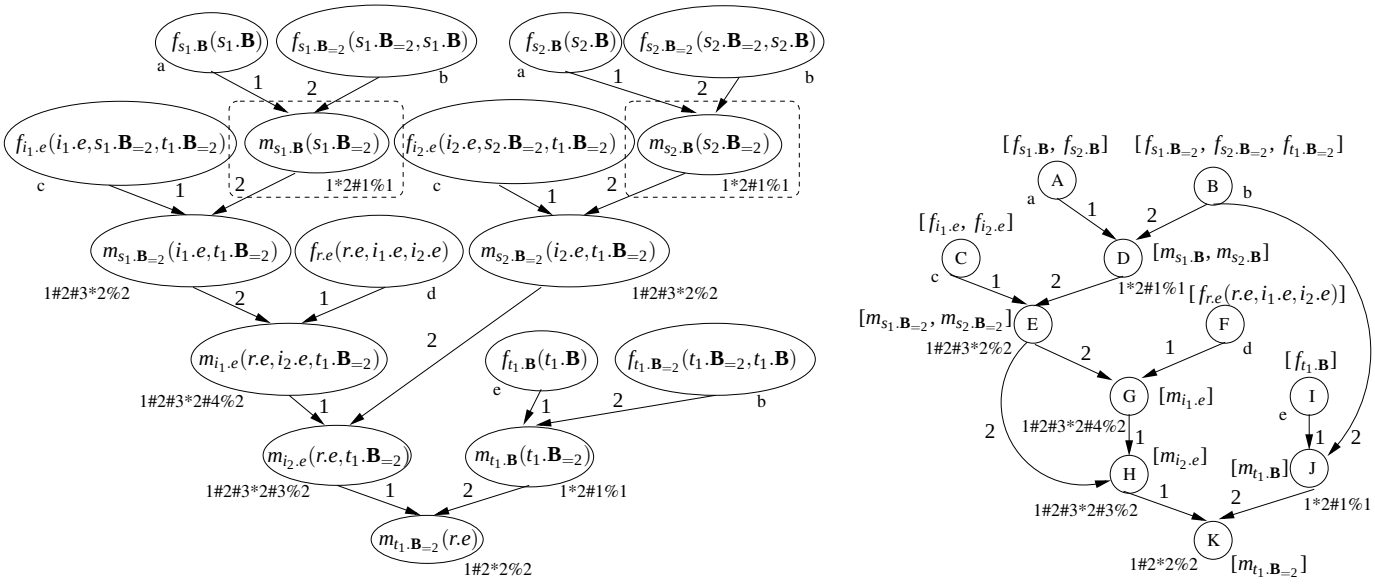


Figure 4: (a) rv-elim graph for the example from Section 2 and (b) its compressed version obtained using bisimulation. The rv-elim graph on the left is a vertex-labeled, edge-labeled graph. The edges are labeled with integers (in this case, 1 or 2) and denote the order in which the parent factors are present in the elimrv operation. The vertices are labeled with strings and these are shown alongside the vertex, if the vertex is a root then the label is a letter (e.g., *a* for the first root in the top left corner), or a string if it is a vertex with parents denoting how the arguments overlap for the elimrv operation that created the intermediate factor corresponding to this vertex (for instance, 1#2*2%2 for the leaf in the rv-elim graph). The compressed rv-elim graph on the right is also an edge-labeled, vertex-labeled graph with the extent of every vertex depicted next to it in square braces. Note that the compressed rv-elim graph in this case consists of 11 vertices whereas the rv-elim graph itself contains 17 vertices, a significant reduction considering we have such a small running example.

Figure 4 (a) shows the rv-elim graph for our running example using the same elimination order we defined in Section 2. One point to note about the rv-elim graph is that, in general, it can never contain a directed cycle (in other words, it has to be a directed acyclic graph (DAG)).

4.3 Identifying Shared Factors

The advantage of representing a run of inference as a graph is that we can now identify exactly when two vertices in the graph represent shared factors. Denote by f_v the factor represented by vertex v in an rv-elim graph.

CLAIM 4.4. *For rv-elim graph $G = (V, E)$, two vertices $v_1, v_2 \in V$ are shared factors $f_{v_1} \cong f_{v_2}$ if:*

- $l(v_1) = l(v_2)$.
- $\forall u_1 \xrightarrow{i} v_1, \exists u_2 \xrightarrow{i} v_2$ *and* $f_{u_1} \cong f_{u_2}$.
- $\forall u_2 \xrightarrow{i} v_2, \exists u_1 \xrightarrow{i} v_1$ *and* $f_{u_1} \cong f_{u_2}$.

Essentially, what the claim says is that two intermediate factors f_{v_1} and f_{v_2} generated during inference (using elimrv operations) are shared if:

- they were produced by multiplying sets of factors containing the same function components (the parents are shared)
- the argument orders, argument alignments and the argument being summed over, all match (the labels on v_1 and v_2 are the same)

Note that for a given internal vertex in the rv-elim graph all incoming edges from parents are assigned distinct edge labels since we label the edges with the index indicating the position of the factor represented by the parent in **F** of the corresponding elimrv operation and two factors cannot be at the same position (Definition 4.3).

We can now use Claim 4.4 to determine the intermediate shared factors that get generated during the inference process. The important thing to realize is that we can do this *without actually computing these intermediate factors*. For instance, recall that in Section 2 we showed that during the run of inference for our running example, $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ were intermediate factors that turned out to be shared (shown in dashed boxes in Figure 4 (a)). By looking at the rv-elim graph (Figure 4 (a)) this is now easy to see since:

- They have the same vertex label 1*2#1%1.
- Both $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ have parents $f_{s_1.\mathbf{B}}$ and $f_{s_2.\mathbf{B}}$, resp., via edges labeled 1, and $f_{s_1.\mathbf{B}} \cong f_{s_2.\mathbf{B}}$ since they have the same vertex label (viz., *a*) and are roots.
- Both $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ have parents $f_{s_1.\mathbf{B}_{=2}}$ and $f_{s_2.\mathbf{B}_{=2}}$, resp., via edges labeled 2, and $f_{s_1.\mathbf{B}_{=2}} \cong f_{s_2.\mathbf{B}_{=2}}$ since they have the same vertex label (viz., *b*) and are also roots.

Thus by Claim 4.4, $m_{s_1.\mathbf{B}} \cong m_{s_2.\mathbf{B}}$. Once we have found out that $m_{s_1.\mathbf{B}} \cong m_{s_2.\mathbf{B}}$, we can also determine that $m_{s_1.\mathbf{B}_{=2}} \cong m_{s_2.\mathbf{B}_{=2}}$ by following similar logic. This is another pair of intermediate factors that we noticed were shared in Section 2.

Given a graph (like the rv-elim graph shown in Figure 4 (a)) and a property (such as the one specified in Claim 4.4), there exist

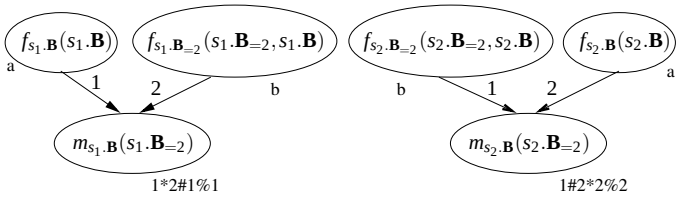


Figure 5: A poor ordering of parent vertices.

reasonably fast algorithms that can partition the set of vertices into disjoint sets such that every pair of vertices in each set satisfies the property. These algorithms are generally referred to as *bisimulation* [17] or computing the *relational coarsest partition* [19]). Given the special case of the graph being a DAG, there exist algorithms that run in time linear in the size of the graph [10].

Dovier et al. [10] describe one such algorithm that runs on an edge-labeled, vertex-labeled graph and not only partitions the set of vertices but also returns another (smaller) graph where each disjoint set in the partition is represented by a vertex and the edges between vertices p_1, representing one disjoint set in the partition, and p_2, representing another disjoint set in the partition, are the result of taking the union of all edges between all vertices from the input graph in p_1 and all vertices from p_2. We will refer to each resulting disjoint set of the vertices of the rv-elim graph as an *extent* and the resulting graph returned as a result of running bisimulation on the rv-elim graph as the *compressed rv-elim graph*. Figure 4 (b) shows the compressed rv-elim graph returned as a result of running bisimulation on the rv-elim graph shown Figure 4 (a). Notice how vertex A represents both factors $f_{s_1.\mathbf{B}}$ and $f_{s_2.\mathbf{B}}$. We show this in Figure 4 (b) by indicating A's extent in square braces next to it. More interestingly, both pairs of intermediate shared factors that we identified earlier have also been collapsed into one single vertex each in the compressed rv-elim graph: D represents $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$, and E represents $m_{s_1.\mathbf{B}_{=2}}$ and $m_{s_2.\mathbf{B}_{=2}}$.

Unfortunately, we cannot apply the bisimulation algorithm described in Dovier et al. directly to our problem because we have not discussed how to choose the order in which the factors appear in each elimrv operation. Recall that the order in which the factors appear affects the results (Section 4.1). Traditional exact inference algorithms simply choose an order for multiplying the factors arbitrarily. However, in our case, Claim 4.4 actually uses the order of the parents of the vertices in the rv-elim graph to determine which ones represent shared factors. Figure 5 illustrates how we may lose compression if we do not choose orders judiciously. In Figure 5, instead of ordering the parents of $m_{s_2.\mathbf{B}}$ with $f_{s_2.\mathbf{B}}$ as the first parent and $f_{s_2.\mathbf{B}_{=2}}$ as the second, we have placed $f_{s_2.\mathbf{B}_{=2}}$ as the first parent and $f_{s_2.\mathbf{B}}$ as the second. A direct consequence of this is that the labels on the vertices representing $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ in the rv-elim graph are now different, which means that using Claim 4.4 we cannot decree them to form a pair of shared factors.

The problem is that even though factor multiplication is a commutative operation, different orders lead to rv-elim graphs with varying degrees of symmetry. We need to choose those orders that lead to rv-elim graphs with more symmetry (consisting of more shared factors). One approach is to try all possible parent orderings but this will likely be too expensive. Instead, we introduce a novel heuristic for choosing better orderings. Our bisimulation algorithm, based on Dovier et al., requires a different interleaving of the steps, so for completeness we first present our bisimulation algoirthm, and then the heuristic we developed for ordering parents.

Algorithm 1: Bisimulation for rv-elim graphs

input: rv-elim graph $G = (V,E)$ with roots labeled

$$rank(v) = \begin{cases} 0 & \text{if } v \text{ is a root} \\ 1 + \max\{rank(v') | v' \to v \in E\} & \text{o.w.} \end{cases}$$

$\rho \leftarrow \max\{rank(v) | v \in V\}$

$B_{0,l} = \{v \in V | v \text{ is a root} \wedge l(v) = l\}\ \forall l$ labels on roots in G

$C = \{B_{0,l}\}$

$B_i = \{v \in V | rank(v) = i\}, \forall i = 1 \ldots \rho$

for $i = 1 \ldots \rho$ **do**

  foreach $v \in B_i$ **do**

    $o \leftarrow$ choose order on v's parents

    construct $l(v)$ based on o

    construct key k_v using $l(v)$ and all (j, b_j) where b_j is the block-id of the j^{th} parent

  construct blocks $B_{i,k} = \{v \in B_i | k_v = k\}$

  add $\{B_{i,k}\}$ thus constructed to C

return final partition C

4.4 Bisimulation for RV-ELIM Graphs

We will assume that we are given an rv-elim graph $G = (V,E)$ for computing marginal probabilities of random variable X from PGM $\mathcal{P}$ using the elimination order $\mathcal{O}$. Each root $v \in V$ is labeled by the id(f_v) where f_v denotes the factor from $\mathcal{F}$ represented by v, we will assign the remaining vertex labels (for the internal vertices) and the the edge labels in G dynamically through the bisimulation algorithm we present.

A *partition* denotes a division of the set of vertices of the rv-elim graph into disjoint sets; each disjoint set is denoted a *block*. The full algorithm is described in Algorithm 1. The bisimulation algorithm starts by computing ranks for each vertex in the rv-elim graph (using a simple depth-first search). After computing ranks, the algorithm starts by assigning the roots in the rv-elim graph to the blocks formed by their labels. After this it goes through the vertices at rank i, partitioning them into blocks. Note that when we are dealing with vertices at rank i, we only need the partitioning on the vertices at ranks $i' < i$ since according to Claim 4.4, the partitioning of a vertex only depends on its label and its parents' partitioning and the parents of vertices at rank i can only have ranks $i' < i$ (the rank computation scheme guarantees this). The nested for loops basically achieve this. They take all vertices at rank i, choose orders for each vertices' parents (we will discuss how this is done shortly), forms the label and the key based on this ordering and partitions these vertices based on the constructed key. See Dovier et al. [10] for proof of correctness when the vertex and edge labels can be statically allocated.

Parent ordering heuristic: We now discuss the parent ordering heuristic we developed. Recall that Claim 4.4 requires both the labels to match and the parent sets of both vertices to be aligned before we decree vertices v and v' to represent shared factors. Our heuristic simply orders the list of parents by their block-ids before constructing the label for the vertex. This helps align the parent vertices.

Algorithm 1, by itself, is reasonably efficient. Its time complexity, assuming we use the heuristic that orders based on block-ids, is $O(|V| + |E|)$ (to compute ranks in step 1) $+ \sum_{v \in V} d_v \log d_v + d_v$ (to order the parents and form the key) where d_v is the in-degree of v (ignoring the time spent to construct $l(v)$) $+ O(|V|)$ to partition vertices at rank i into blocks based on their key. Adding up, this gives us $O(\sum_v d_v \log d_v + |V|) = O(|E| \log D + |V|)$ where D is the maximum in-degree of any vertex in the rv-elim graph.

4.5 Performing Inference on the Compressed RV-ELIM Graph

Having computed the partitioning of the vertices using Algorithm 1, we can now construct the compressed rv-elim graph as described earlier in Section 4.3 by representing each block in the partition using a vertex, copying the label on the vertices to the label on the block, and introducing an edge with label *i* between two blocks if there exists a pair of vertices that have an edge with label *i*. These definitions are consistent because the blocks of the partition correspond to particular keys constructed by Algorithm 1 which contain the vertex labels and edge labels, and all vertices in block have the same key.

We can now perform inference on the compressed rv-elim graph. To seed the inference, we simply copy the function components of the factors corresponding to roots of the rv-elim graph to the roots in the compressed rv-elim graph. Then we call a depth-first search procedure (dfs) from the leaf in the compressed rv-elim graph that begins by looking at the parents, the labels on the edges and the vertices and applies the elimrv operator to compute the functions on the child. If a parent's functions haven't been computed yet then we make the dfs call on the parent before applying elimrv on the child. Finally, we will have the (unnormalized) marginal distribution computed at the leaf of the compressed rv-elim graph.

The inference procedure presented in this section is fairly flexible and a number of extensions are possible. We can use our inference procedure to compute, besides single-node marginal probabilities, multiple marginal probability distributions at once but in this case the compressed rv-elim graph may have multiple leaves. Another extension is to use it to compute maximum-a-posteriori (MAP) assignments instead of marginal probabilities for which we simply need to switch from the sum-product elimrv operator to the max-product operator.

5. COMPUTING ELIMINATION ORDERS

One of the important steps in performing probabilistic inference is to choose a good elimination order that helps run inference without producing large intermediate factors (in terms of number of arguments) during the run of inference. This can make the difference between inference being tractable or intractable since the size of a factor is proportional to the product of the domain sizes of its argument random variables. In our case, since we are interested in exploiting shared factors, and since the elimination order affects the rv-elim graph constructed, we would like to construct elimination orders that produce smaller factors *and*, at the same time, produce rv-elim graphs that can be compressed using bisimulation. Unfortunately, even without the consideration of shared factors this problem is known to be NP-Hard [2]. Thus, as is done in traditional inference algorithms, we resort to heuristics. In this section, we first review the popular minimum size heuristic [18] that is often used to construct elimination orders for traditional exact inference algorithms, then we describe our approach that uses a modified version of the minimum size heuristic to construct effective elimination orders that lead to symmetric rv-elim graphs.

5.1 Minimum Size Heuristic

Traditional minimum size heuristic (MSH) is a greedy heuristic that returns a list of the random variables that need to be eliminated for inference. The concept of *neighborhood* is central to MSH. Given a random variable X and a collection of factors $\mathbf{F}$, we define the neighborhood of X to be the set of random variables with which it appears as arguments in $\mathbf{F}$. MSH begins by collecting all the factors in the PGM over which we want to run inference. In each iteration, MSH greedily picks the random variable with the smallest-sized neighborhood, updates the collection of factors by deleting the factors where the random variable appears as an argument and adding the new factor produced by the elimination operation, and repeats these steps until all the random variables that need to be eliminated have been picked. Algorithm 2 shows the complete algorithm.

Algorithm 2: Minimum Size Heuristic

```
input: PGM ⟨F, X⟩, random variable X
initialize empty list O
while ∃Y ∈ X s.t. Y ≠ X do
    pick Y ≠ X with minimum sized neighborhood
    add Y to O
    //update F
    delete from F all factors that contain Y as an argument
    add the factor introduced by eliminating Y
    /*note that we only introduce a placeholder, we do not
      actually compute the factor produced by eliminating Y */
add X to O
return O
```

5.2 Elimination Orders for RV-ELIM Graphs

Unfortunately, simply applying MSH to compute elimination orders may not work well in our case because the elimination orders constructed by MSH may not lead to symmetric rv-elim graphs. To this end, we generate elimination orders in two phases:

1. We first identify sets of "similar" random variables. This should help construct elimination orders that lead to rv-elim graphs which can be compressed better.

2. We then use a modified version of MSH with a novel neighborhood definition to complete the construction of elimination orders.

Finding Similar Random Variables: We first explain how determining similar random variables leads to symmetric rv-elim graphs. Recall from our running example that we eliminated $s_1.\mathbf{B}_{=2}$ first followed by $s_2.\mathbf{B}_{=2}$. If instead we had eliminated $t_1.\mathbf{B}_{=2}$ after eliminating $s_1.\mathbf{B}_{=2}$, then we would have created an intermediate factor $m'_{t_1.\mathbf{B}_{=2}}(i_1.e, i_2.e, s_2.\mathbf{B}_{=2}, t_1.\mathbf{B})$ which is unlike any other factor we encountered during the inference run in that example, and thus the rv-elim graph constructed would have less symmetry. Finding sets of similar random variables that occur in shared factors and eliminating them one after another should help generate rv-elim graphs with better compression properties. Fortunately, we can easily represent a PGM as a graph where the random variables are represented using vertices and correlations are represented using edges (Figure 6 (a) shows the PGM graph for our running example) and we can use this PGM graph to find similar random variables simply by labeling the vertices using the ids of the factors from the PGM (if the random variable is present in multiple factors then aggregate their ids using some operation such as max or sum, assuming the ids are numbers). Then we run a bisimulation on the PGM graph to compute a partition on the random variables of the PGM and the corresponding compressed PGM graph (Figure 6 (b) shows the compressed PGM graph for our running example). Each extent thus obtained after bisimulation contains similar random variables. Bisimulation algorithms for general graphs (with cycles) are available [10].

Modified Minimum Size Heuristic: Having constructed the sets of similar random variables we would now like to ensure that we

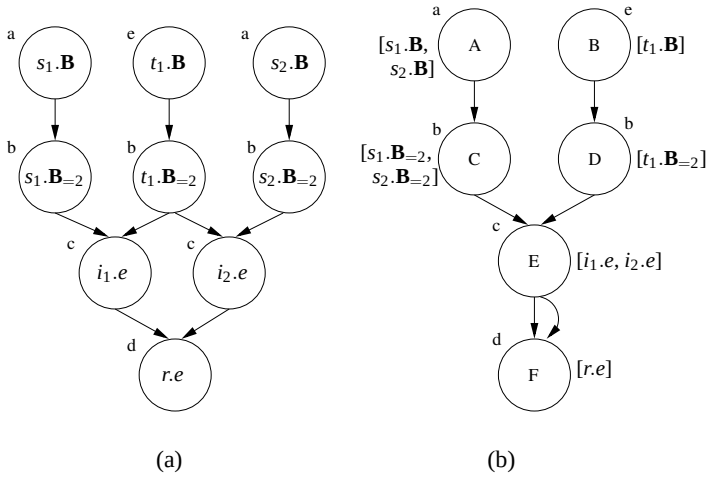


Figure 6: (a) Example PGM graph (b) its compressed version.

eliminate those random variables one after another and that we avoid generating large factors in the process. One simple way to do this is to produce an ordering on the vertices of the compressed PGM graph and then expand the entries in that ordering using the extents. In addition, producing an ordering on the vertices of the compressed PGM graph is likely to be faster since the compressed graph is likely to contain less vertices compared to the number of random variables in the PGM.

To produce an ordering over the vertices in the compressed PGM graph, we could use MSH by replacing the definition of neighborhood presented earlier with the neighborhood of a vertex in the compressed PGM graph. Unfortunately, this is unlikely to work well. This is easily seen from the compressed PGM graph we constructed for our running example. The vertex containing $t_1.\mathbf{B}_{=2}$ in its extent (D) is connected to only one other vertex (E). This may cause MSH to place D early in the ordering produced from the compressed PGM graph, which would cause $t_1.\mathbf{B}_{=2}$ to appear early in the final elimination order. We just saw that this might lead to a 4 argument intermediate factor. The situation becomes worse if we imagine t_1 joining with not 2 (as in our example) but $k > 2$ tuples. The problem here is that the neighborhood of a vertex in the compressed PGM graph is not a good indicator of the size of the intermediate factor produced by an elimination. In the above case, the neighborhood of D is 1 but when we eliminate $t_1.\mathbf{B}_{=2}$ we actually involve at least as many random variables in the intermediate factor as there are in the extent of E. This leads us towards defining a new neighborhood criterion that involves not only the neighborhood in the compressed PGM graph but also the extents of the vertices in the neighborhood. Define *avg. neighborhood size* to be $= \frac{\sum_{v' \in \mathcal{N}(v)} |\text{extent}(v')|}{|\text{extent}(v)|}$ where $\mathcal{N}(v)$ denotes the neighborhood of v in the compressed PGM graph. Essentially, avg. neighborhood assumes that there are as many neighbors to vertex v as there are random variables in all neighbors' extents summed up. It essentially tries to estimate the neighborhood of the vertex with respect to the uncompressed PGM graph, and it compensates for the case when v itself has a large extent by dividing by the extent size. Thus it tries to make MSH behave as if we are running it on the uncompressed PGM graph but actually runs on the compressed PGM graph thus making it more efficient. Algorithm 3 shows the final modified minimum size heuristic that we used in our implementation.

Algorithm 3: Modified minimum size heuristic

input: compressed PGM graph $G = (V, E)$, and vertex v_X that contains X (whose marginals we need) in its extent
intialize empty list **O**
while $\exists v \in V$ s.t. $v \neq v_X, v \notin$ **O do**
  pick vertex $v \neq v_X$ with the smallest avg. neighborhood
  add v to **O**
  introduce an edge between every pair of neighbors of v
construct $\mathcal{O}$ by expanding entries in **O** with their extents
add extent(v_X)$\backslash\{X\}$ to $\mathcal{O}$
return $\mathcal{O}$

6. EXPERIMENTAL EVALUATION

Our experiments were designed to answer the question: When is it worthwhile to apply our bisimulation-based approach to a query evaluation problem? Note that standard inference algorithms take a PGM and a random variable, and simply begin multiplying factors and summing over random variables (after computing the elimination order). Instead, our approach first constructs the rv-elim graph, applies bisimulation to compress it, and then begins multiplying function components of factors and summing over arguments from them. So it is plausible that there may be cases where our approach may perform poorly because it spends too much time before actually getting to the point where it can perform (a hopefully smaller set of) multiplications and summations.

We compare against a baseline exact inference algorithm, denoted BatchVE, which is a modified version of variable elimination (VE) except that if the PGM contains multiple random variables whose marginal probabilities we are interested in, then it avoids multiple passes through the PGM like standard VE does [27]. We refer to our approach, which constructs a compressed PGM which exploits shared factors, as SharedInf.
Our experimental results suggest the following:

- In most cases, SharedInf is significantly faster than BatchVE.
- In a small number of cases, SharedInf loses out to BatchVE; but in these cases the difference between the time it took to run SharedInf and BatchVE was not large.[¶]

For each experiment we report 5 numbers:

- **Relational algebra operations** (Rel. alg. ops): time taken to perform the relational algebra operations in the query to construct the PGM.
- **BatchVE arithmatic operations** (BatchVE arith. ops.): time taken to multiply factors and sum over random variables during inference for BatchVE.
- **BatchVE remaining operations** (BatchVE rem.): time required to perform the remaining BatchVE operations such as determining the elimination order.
- **SharedInf arithmatic operations** (SharedInf arith. ops.): time spent multiplying functions and summing over arguments (on the compressed rv-elim graph) for our approach.
- **SharedInf remaining operations** (SharedInf rem.): time taken to perform the remaining operations for our approach. This includes the bisimulation and the time spent to determine the elimination order from the compressed PGM graph.

[¶]Note that early stopping techniques are possible, such as once we run bisimulation on the PGM graph and find out that the extents of the compressed PGM graph are small then we can switch our inference engine and resort to BatchVE, but for our experiments we did not include this approach.

For each experiment we report three bars (except for Figure 7 (e)): the first bar reporting the rel. alg. ops. time; the second, time spent by BatchVE; and the third, time spent by SharedInf. See the legend (top left in Figure 7) for more details. Note that no single bar reports the actual time to run the query. To find out the total time taken to run the query we need to add the rel. alg. ops. time to the second bar or the third bar, depending on the algorithm.

All our experiments were run on a dual processor Xeon 3 GHz machine with 3GBytes of RAM. Our implementation is in JAVA and the numbers we report were averaged over 10 runs.

6.1 Car DB Experiments

For our first set of experiments, we developed the pre-owned car ads example further and randomly generated data and factors to illustrate how the performance of the two algorithms vary with different characteristics of the data. In addition to the relation containing the various advertisements (*Ad*) described in Figure 1 (a), we added another relation which denotes the source websites from which the ads were pulled (*S*). Each tuple in *S* is an uncertain tuple with an associated probability of existence which depends on how reliable the website's information is. For these experiments, we ran the following query: $\prod_{\mathbf{AdID}}((\sigma_{\mathbf{Color}=c}Ad) \bowtie_{\mathbf{SID}} S)$ where *c* denotes a specific color and **SID** is a primary key in *S* and acts as a foreign key in *Ad*. Besides the uncertain tuples in *S*, we set the **Color** attribute values to be uncertain and these were correlated with the corresponding **Make** attributes. A car of a certain **Make** can have one of 4 distinct **Color**s. The parameters that we varied for these experiments are *d* (domain size of **Make**, default was 50), *n* (the number of attribute uncertainty tuples in *Ad*, default value is 1000) and fanout (the number of tuples in *Ad* that each tuple from *S* joins with, default value is 1000).

In Figure 7 (a), we show how SharedInf and BatchVE perform when we vary *n* from 100 to 1000. Notice that SharedInf significantly reduces the time spent performing arithmatic operations. Note that on the x-axis in Figure 7 (a), we report the size of *Ad* in terms of number of uncertain tuples to help the reader compare with previous work on probabilistic databases since our formulation can deal with both attribute uncertainty and tuple uncertainty but most recent work can handle tuple uncertainty only. To convert data with attribute-level uncertainty to tuple-level uncertainty, one simple approach is to compute all possible joint instantiations of every tuple present in the attribute-level uncertainty database. This transformation "flattens" out a relation *R* with uncertain attributes into $n \times d_1 \times d_2 \times \ldots d_{|attr(R)|}$ tuples, where *n* is the number of attribute uncertainty tuples and d_i is the domain size of the i^{th} uncertain attribute in *R* (assuming all uncertain attribute values have the same domain size). For our experiment, this gives us a size of $n \times d \times 4d$ for the *Ad* relation in tuple-uncertainty format.

Figure 7 (b) shows the performance of the two inference algorithms with varying domain sizes. Notice how at $d = 10$, SharedInf performs worse (because small domain sizes means small factors and therefore, less time spent on arithmatic operations) but the difference between its time and BatchVE's time is not large.

The third experiment we ran (Figure 7 (c)) is the most interesting experiment in this subsection. Here we varied the fanout from 1 to 10 to vary the symmetry in the PGMs produced by the query (but kept the number of tuples in *Ad* fixed). At fanout 1, we have no symmetry and no shared factors in the base data since every tuple from *S* has a unique existence probability but the shared factors increase as we increase fanout. Thus, at fanout 1 SharedInf should perform worse, and it does, but not by a huge margin. At fanout 2, where we have a slight amount of symmetry in the query (every tuple from *S* joins with exactly 2 tuples from *Ad*) SharedInf is already doing better than BatchVE. At fanout 10 it does much better than BatchVE.

In Figure 7 (d), instead of keeping the fanout constant for all tuples in *S*, we sampled it from a Poisson distribution with parameter λ. In this case however, we kept the number of tuples in *S* fixed. Note that at $\lambda = 1$, most fanouts sampled turn out to be 1, but some samplings produce 2, 3 ... numbers greater than 1 and SharedInf utilizes this to do better than BatchVE even at $\lambda = 1$. At $\lambda = 10$, SharedInf performs much better.

Until now we had kept the existence probabilities of tuples in relation *S* distinct: in the next experiment we introduced some shared factors here by dividing the tuples in *S* into buckets. Two tuples in the same bucket had the same existence probability. The number of tuples in *S* were fixed to 600, so at 600 buckets (right end of the plot), we have exactly 1 tuple belonging to each bucket. Figure 7 (e) shows how SharedInf's performance deteriorates when the number of buckets increase. Note that we do not show the time taken by BatchVE in this case since it would obscure the trend of SharedInf (BatchVE took around 25 seconds for this experiment).

6.2 Experiments with Uncertain Join Attributes

The next two plots, Figure 7 (f) and Figure 7 (g), relate to a two relation join between *S* and *Ad* where the join attribute **SID** itself was uncertain. This relates to the case of link uncertainty or structure uncertainty [13], where we are unsure about the primary/foreign key values in the data. For instance, we may have another relation in our database which stores the id of the person who posted the pre-owned car ad. We may want to join with that relation so we can take into account the reliability of the seller while trying to return to the user cars of her/his interest. But we may not know the seller's identity as this information may not have been properly extracted or is simply unavailable (s/he used the guest login). Joins on uncertain attributes give rise to very complicated PGMs and to keep some control over the complexity of the PGM, we setup this experiment in the following fashion. First we constructed *k* key values, then for each tuple in either relation we polled from this pool of keys *m* distinct keys randomly to include in the domain of the uncertain join attribute value. Finally we padded each attribute value's domain with unique key values so that the total domain size is 50. Thus increasing *k* makes it less likely that two tuples from the two relations join, on the other hand, increasing *m* increases the chance that two tuples join. Note that if two tuples join then this may be due to multiple entries being common in their domain. Figure 7 (f) shows that increasing the value of *m* (*k* was held constant at 100) both algorithms' times increase, although BatchVE has a more pronounced dependency on the value of *m*. Figure 7 (g) shows how increasing the value of *k* (*m* was held constant at 2) helps reduce SharedInf's times more drastically than BatchVE's.

6.3 Experiments with TPC-H Data

Following previous work, we also ran experiments based on the TPC-H schema. We picked Q5 from the TPC-H specification since this involves a join among six relations, of which we made 4 relations (customer, lineitem, supplier and order) probabilistic. The query tries to determine how much volume of sales is being generated in various regions. Each customer places k_1 orders, each order is broken down into k_2 sub-orders each of which is a lineitem entry, each sub-order is then diverted to a supplier. Each tuple from customer is uncertain and these were divided into p_1 buckets such that tuples from the same bucket had the same existence probabilities: similarly, the supplier tuples were also divided into p_2 buckets. Moreover, each customer sub-order is usually (with 95% probability) routed to one of *c* suppliers, else the supplier is chosen ran-

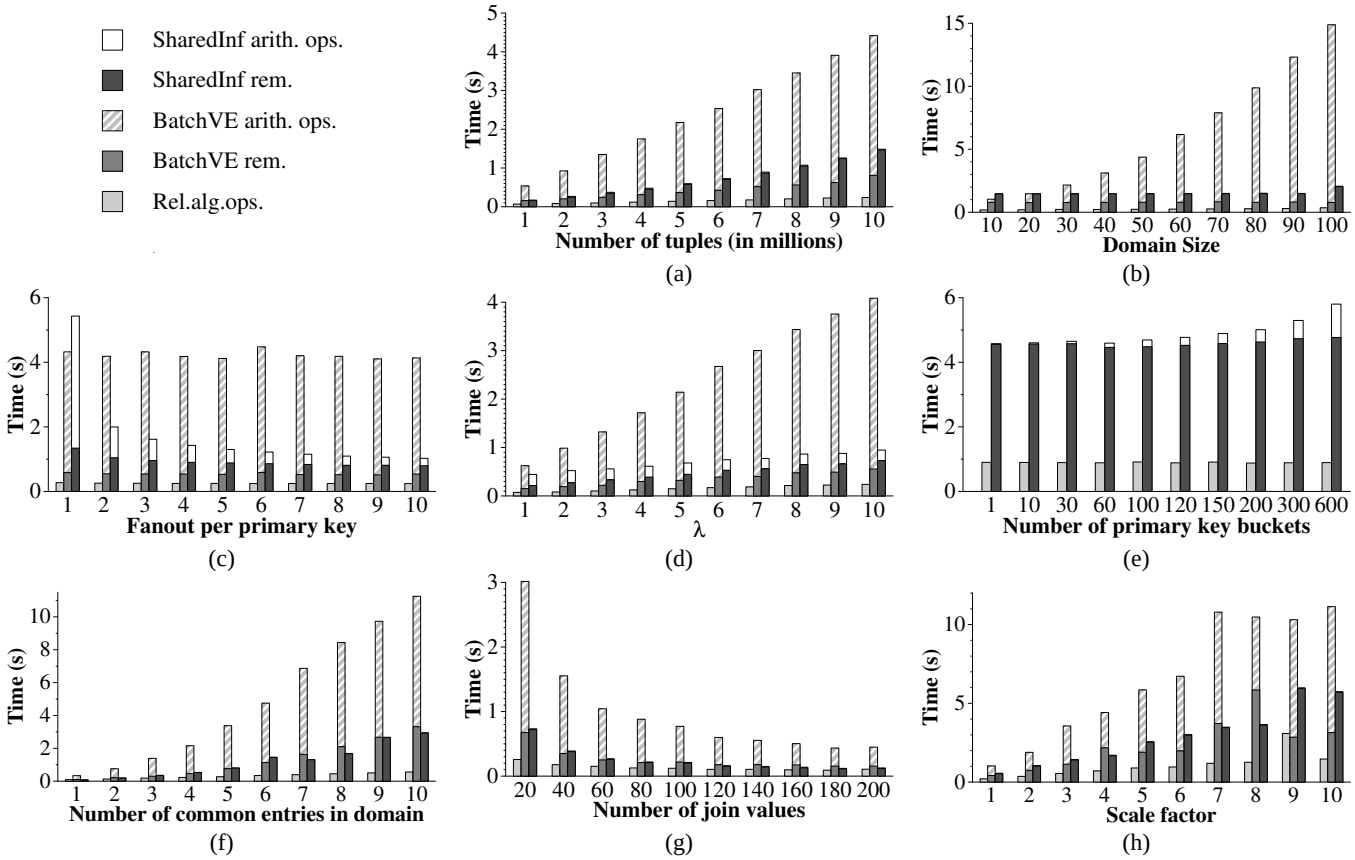


Figure 7: Plots for experiments on synthetic and TPC-H data. The legend is shown in the top left corner.

domly. For the lineitem and order relations, we made the discount attribute uncertain (domain size $4d$) and correlated with part being ordered's type (domain size d), and the orderdate attribute uncertain (domain size d). We set the parameters in the following manner: $k_1 \sim poisson(2)$, $k_2 \sim poisson(3)$, $p_1 = p_2 = 5$, $c = 3$, $d = 50$. We defined the scale factor to be the number of tuples in lineitem in tuple-uncertainty format divided by 6×10^6. The results are shown in Figure 7 (h). The results showed similar trends for other parameter settings, for instance the execution time for SharedInf went down when we decreased c and increased d and so on.

In almost all our experiments, we noticed significant speedups ranging from 200% to 700%. Even in cases where there was no symmetry, SharedInf performed only slightly worse than BatchVE, incurring about 25% extra time to compress rv-elim graphs. Given that the datasets we generated were extremely simple in their correlation structure, we believe we will do even better on real-world data with richer correlation structure containing shared factors.

7. RELATED WORK

Our work on exploiting shared correlations is closely related to recent work on lifted inference [20, 8] done in the artificial intelligence community. Most of the work in lifted inference assumes the presence of a PGM described using a first-order probabilistic model (see [14] for an introduction). The goal of lifted inference is to run inference efficiently by utilizing the shared correlations clearly specified by the first-order description. In our work, we did not assume the presence of a first-order description. This is because the query evaluation approach for probabilistic databases does not provide it and it is not straightforward to redefine the query evaluation approach to do so. Instead, we showed how to automatically discover the symmetry in the PGM by using a bisimulation-based algorithm. To the best of our knowledge, our approach is the first general lifted inference approach that can be applied to any PGM, and our approach is more widely applicable than the case-based special-purpose algorithms on which other work on lifted inference have concentrated [20, 8].

Probabilistic databases have a lot in common with probabilistic relational models (PRMs) [11, 13] since both define a probabilistic model on relational data, although there are some differences. Most of the work on PRMs have concentrated on how to specify and learn a class-level probabilistic model for relational data. Unlike the work on probabilistic databases, answering queries expressed in a standard query language (e.g., relational algebra or SQL) was not their main focus. We believe that the best way to view our work described in this paper is to look upon it as taking the best of these two worlds. PRMs define a concise class-level probabilistic model on relational data, which by definition provides shared factors, and our approach exploits these shared factors to speed up inference for query evaluation in probabilistic databases.

Other work has also tried to make query evaluation in probabilistic databases efficient. Das Sarma et al. [7] (Trio) describe memoization techniques in which every time a tuple t's existence probability (they only deal with tuple-level uncertainty) is computed, Trio caches its result, and this can then be used if some other tuple's probability of existence requires t's probability of existence. In contrast, our approaches reuse computation at a finer level by computing each intermediate factor once and reusing it for every shared intermediate factor that is generated during the run of infer-

ence. Trio does not attempt to exploit shared correlations. Other approaches to faster query processing include using index structures [25], but these only help in data retrieval, and not in speeding up the inference process itself, and approximate query processing [21], where the assumption is that the user is only interested in finding the correct ranking of result tuples and not the exact probabilities. In contrast, for this work, we were more interested in computing exact probabilities. On a side note, Bravo and Ramakrishnan [4] suggest representing factors as relations so that we can take the external memory algorithms already implemented in a traditional RDBMS to efficiently implement the elimrv operation. In our query evaluation procedure, we tend to produce numerous small factors (a 3 argument $\wedge$-factor involving three exists random variables consists of only $2^3 = 8$ rows) and representing each of them as a separate relation will be infeasible. It may be interesting to see how their methods of representing large factors using relations can be combined with our current approach of representing factors as objects.

Finally, on the bisimulation front, Kanellakis and Smolka [17] defined this problem in the context of testing equivalence of two finite state processes and provided an $O(|V||E|)$ time algorithm, Paige and Tarjan [19] improved upon their result to provide an $O(|E|\log|V|)$ time algorithm and Dovier et al. [10] provided an algorithm that runs on DAGs in $O(|V|+|E|)$ time besides describing simple extensions to handle labeled graphs.

8. CONCLUSION

In this paper, we showed how to exploit shared correlations to speed up probabilistic inference during query evaluation for probabilistic databases. Shared correlations are likely to exist in many probabilistic databases since probabilities and correlations often come from general statistics learnt from (large amounts of) data and rarely vary on a tuple-to-tuple basis. In addition, the query evaluation approach itself tends to introduce shared correlations. We introduced a new graph-based data structure and explained how to build it from the probabilistic graphical model. We then showed how the graph can be compressed using an algorithm based on bisimulation. We empirically evaluated our approach and showed that even in the presence of a few shared correlations, we do significantly better than naive inference approaches. Avenues for future work include developing approximate inference algorithms that exploit shared factors and developing algorithms that utilize the first-order description of probabilistic models to build rv-elim graphs if such a description is available in the probabilistic database. We hope that further development of techniques that specifically target PGMs arising out of probabilistic databases will drastically improve query evaluation times beyond the current state of the art.

Acknowledgements: This work was supported in part by the National Science Foundation under Grants No. 0438866 and IIS-0546136. We would also like to thank Brian Milch for stimulating discussions which eventually led to the realization that graph theory can help identify shared factors.

9. REFERENCES

[1] P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. In *ICDE*, 2006.

[2] S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. In *BIT*, 1985.

[3] O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[4] H. C. Bravo and R. Ramakrishnan. Optimizing mpf queries: decision support and prob. inference. In *SIGMOD*, 2007.

[5] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating prob. queries over imprecise data. In *SIGMOD*, 2003.

[6] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

[7] A. Das Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *ICDE*, 2008.

[8] R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In *International Joint Conference on Artificial Inteligence*, 2005.

[9] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[10] A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. In *International Conference on Computer Aided Verification*, 2001.

[11] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *International Joint Conference on Artificial Inteligence*, 1999.

[12] N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 1997.

[13] L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models with link uncertainty. *Journal of Machine Learning Research*, 2002.

[14] L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, 2007.

[15] R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, 2006.

[16] C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 1994.

[17] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *ACM Symposium on Principles of Distributed Computing*, 1983.

[18] U. Kjaerulff. Triangulation of graphs - algorithms giving small total state space. Technical report, University of Aalborg, Denmark, 1990.

[19] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 1987.

[20] D. Poole. First-order probabilistic inference. In *International Joint Conference on Artificial Inteligence*, 2003.

[21] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, 2007.

[22] C. Re and D. Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *VLDB*, 2007.

[23] I. Rish. *Efficient Reasoning in Graphical Models*. PhD thesis, University of California, Irvine, 1999.

[24] P. Sen and A. Deshpande. Representing and querying correlated tuples in probabilistic databases. In *ICDE*, 2007.

[25] S. Singh, C. Mayfield, S. Prabhakar, S. Hambrusch, and R. Shah. Indexing uncertain categorical data. In *ICDE*, 2007.

[26] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville, and R. Cheng. Database support for probabilistic attributes and tuples. In *ICDE*, 2008.

[27] N. L. Zhang and D. Poole. A simple approach to bayesian network computations. In *Canadian Conference on AI*, 1994.