The VLDB Journal (2009) 18:1065–1090
DOI 10.1007/s00778-009-0153-2

SPECIAL ISSUE PAPER

PrDB: managing and exploiting rich correlations in probabilistic databases

Prithviraj Sen · Amol Deshpande · Lise Getoor

Received: 15 September 2008 / Revised: 7 June 2009 / Accepted: 10 June 2009 / Published online: 15 July 2009

Abstract Due to numerous applications producing noisy data, e.g., sensor data, experimental data, data from uncurated sources, information extraction, etc., there has been a surge of interest in the development of probabilistic databases. Most probabilistic database models proposed to date, however, fail to meet the challenges of real-world applications on two counts: (1) they often restrict the kinds of uncertainty that the user can represent; and (2) the query processing algorithms often cannot scale up to the needs of the application. In this work, we define a probabilistic database model, PrDB, that uses graphical models, a state-of-the-art probabilistic modeling technique developed within the statistics and machine learning community, to model uncertain data. We show how this results in a rich, complex yet compact probabilistic database model, which can capture the commonly occurring uncertainty models (tuple uncertainty, attribute uncertainty), more complex models (correlated tuples and attributes) and allows compact representation (shared and schema-level correlations). In addition, we show how query evaluation in PrDB translates into inference in an appropriately augmented graphical model. This allows us to easily use any of a myriad of exact and approximate inference algorithms developed within the graphical modeling community. While probabilistic inference provides a generic approach to solving queries, we show how the use of shared correlations, together with a novel inference algorithm that we developed based on bisimulation, can speed query processing significantly. We present a comprehensive experimental evaluation of the proposed techniques and show that even with a few shared correlations, significant speedups are possible.

P. Sen · A. Deshpande (✉) · L. Getoor
Computer Science Department, University of Maryland,
College Park, MD 20742, USA
e-mail: amol@cs.umd.edu

Keywords Probabilistic databases · Uncertain databases · Graphical models · Query processing · Lifted inference · Bisimulation

1 Introduction

Many real-world applications produce large amounts of uncertain data. Traditional database systems are geared toward storing exact data and harbor fundamental limitations when it comes to storing uncertain data. This has led to renewed interest in designing databases that can store and query uncertain data. To date, numerous approaches have been proposed including fuzzy-logic based approaches [3,6], logic based approaches [27], approaches based on Dempster–Shafer theory [8] and approaches based on probability theory [11,13,21,28,31,40,46]. We refer to the last category, collectively, as *probabilistic databases*. Probabilistic databases have been used in various applications including information retrieval [11,40], recommendation systems [36], mobile object data management [7], information extraction [24], data integration [1] and sensor network data management [17].

Even though a number of probabilistic database models have been proposed, most of them limit the kinds of uncertainty they allow the user to represent. Some require complete tuple independence or permit only simplistic correlations—restrictive assumptions that are rarely true in practice. Most others have no way to represent or benefit from the commonality in the probabilistic information. Such commonalities are observed quite frequently in practice, and arise for two reasons: (1) uncertainty and correlations usually come from general statistics derived from probability distributions defined at the attribute or schema level, and rarely vary on a tuple-to-tuple basis; (2) the query evaluation process tends to

Ad	SellerID		Date	Type		Model	mpg	Price	$prob_e$
	-	*prob*		-	*prob*				
101	201		1/1	Sedan		Civic(EX)	?	$6000	0.5
102	201		1/10	Sedan		Civic(DX)	?	$4000	0.45
103	201	0.6	1/15	Sedan	0.3	Civic	?	$12000	0.8
	202	0.4		Hybrid	0.7				
104	202		1/1	Hybrid		Civic	?	$20000	0.2
105	202		1/1	Hybrid		?	?	$20000	0.2

(a) Advertisements

Type	Model	mpg	*prob*
Sedan	Civic(EX)	26	0.2
		28	0.6
		30	0.2
	Civic(DX)	32	0.1
		35	0.7
		37	0.2
	Civic	28	0.4
		35	0.6
Hybrid	Civic	45	0.4
		50	0.6

(e)

SellerID	Reputation
201	Shady
202	Good

(b) Sellers

Ad 101	Ad 102	*prob*
valid	valid	0.4
valid	invalid	0.1
invalid	valid	0.05
invalid	invalid	0.45

(c)

Model	*prob*
Civic	0.02
Civic(EX)	0.01
Camry	0.01
...	...

(d)

Fig. 1 **a**, **b** A simple car advertisement database with two relations, one containing uncertain data, **c** a joint probability function (*factor*) that represents the correlation between the validity of two of the ads ($prob_e$ for the corresponding tuples in the *Advertisements* table can be computed from this), **d** a *shared* factor over *Model*, **e** a *shared* factor that captures the correlations between several attributes in *Advertisements*—this can be used to obtain a probability distribution over missing attribute values for any tuple

re-introduce the same correlations between base tuples and intermediate tuples generated during execution.

Example 1 We illustrate the need for modeling correlations at various levels of abstraction through a motivating application scenario. Consider a simple car advertisement database (Fig. 1) containing information regarding pre-owned cars for sale, culled from various sources on the Internet. By its very nature, the data in such a database contains various types of uncertainties that interact in complex ways. First off, we may have uncertainty about the validity of a tuple, for example, older ads are likely to correspond to cars that have already been sold, and thus are perhaps less likely to be valid. We may represent such uncertainty by associating an *existence probability* (denoted $prob_e$) with each tuple. Second, many of the attribute values may not be known precisely. In some cases, we may have an explicit probability distribution over an attribute value (e.g., the *SellerID* attribute for Ad 103 in Fig. 1a). The data may also exhibit complex attribute-level or tuple-level correlations. For instance, since the ads 101 and 102 are both by the same seller, their validity is expected to be highly correlated; such a correlation may be represented using a joint probability distribution (Fig. 1c).

Many of the uncertainties in this database, however, are derived from general statistics defined at the schema level. For instance, if the *model* information for a car is missing (Ad 105), we may use a probability distribution as shown in Fig. 1d to derive the uncertainty for that attribute value. Similarly, we may have a joint probability distribution over the attributes of a relation, and the uncertainty in the attribute values for a specific tuple may be computed using the known attribute values for that tuple. Figure 1e shows such a joint probability distribution over the attributes *type, model* and *mpg*; this can then be used to compute a distribution over the *mpg* attribute for a specific tuple (given the tuple's *type* and/or *model* information). We refer to these schema level probabilities as *shared correlations*.

Our goal is to design a probabilistic database model that can capture the uncertainties and complex correlations that appear in such real-world application domains, yet at the same time allows the flexibility to capture probabilistic regularities. To achieve our goal, we base our approach on work done by the machine learning community, a community that has spent a considerable amount of effort to develop expressive yet compact uncertainty modeling techniques.

One of the most popular uncertainty modeling techniques developed by the statistics and machine learning communities, known for its compactness and generality, is the family of techniques referred to as *probabilistic graphical models* (PGM). In the first part of this article, we show how PGMs can be used as a base model of uncertainty in PrDB. We define the semantics of PrDB and show how it, like other probabilistic database models, defines a distribution over possible databases. We also show how PrDB has none of the representational limitations that usually accompany other probabilistic database models. More precisely, we can represent the presence/absence of tuples (tuple-level uncertainty), uncertainty associated with the values of unknown attributes (attribute-level uncertainty) and rich correlations including intra-tuple attribute value correlations, inter-tuple attribute value correlations and even inter-relation attribute value correlations. In addition to the added flexibility, PrDB allows explicit representation of *shared* and *schema-level* probabilistic information (*shared correlations*); this allows for compactness in spite of such a rich modeling framework.

Having defined our probabilistic database in terms of a flexible uncertainty modeling technique, the next question we address is query evaluation. To this end, in the second part of this article, we show that query evaluation in our probabilistic database is no harder than probabilistic inference in an appropriately constructed (augmented) PGM. Further, we show that augmenting the base PGM to answer a query is almost as easy as query evaluation in traditional database management systems where one takes each operator from the user-submitted query and creates intermediate tuples until all operators from the query are exhausted. Perhaps the biggest advantage of making this connection to probabilistic inference is that it lets us utilize any of the probabilistic inference algorithms (exact or approximate) developed in prior literature by the artificial intelligence, machine learning and theory communities.

While probabilistic inference provides a viable option for solving queries in many cases, the hardness of the problem necessitates that we look for further avenues to reduce the complexity of query evaluation. To this end, in the third part of this article, we show how to leverage the existence of shared correlations for efficient query evaluation, by developing a novel inference algorithm based on *bisimulation*. Most probabilistic database formulations require 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 [13,21,36]. The techniques we develop in this paper should be of use to the above mentioned works, by enabling us to go beyond the currently known set of queries that can be efficiently evaluated (e.g., safe plans [11]).

We make the following contributions in this paper:

- We show how state-of-the-art uncertainty models can be used in conjunction with probabilistic databases. We define the semantics of such a probabilistic database.
- We introduce the concept of shared correlations using simple motivating examples and enrich our probabilistic database model to explicitly represent them.
- We show how query evaluation in a probabilistic database reduces to probabilistic inference in an augmented PGM for which any inference algorithm previously developed in the literature can be utilized.
- We introduce a novel graph-based data structure, called the *rv-elim* graph, that helps identify regularities during query evaluation, and we show how it can be used to exploit shared correlations to speed up query evaluation.
- We present a novel algorithm for choosing an elimination order that attempts to maximize the regularities exploited by our inference algorithm.
- We show how to efficiently store the uncertainty model with shared correlations in a relational database system.
- We validate our approach empirically and show that significant speedups are possible, even in the presence of just a few shared correlations.

The rest of the article is organized as follows. In the next section (Sect. 2), we review models for uncertainty from the machine learning literature. In Sect. 3, we present our probabilistic database based on PGMs and define its semantics. In Sect. 4, we show how probabilistic inference relates to query evaluation in probabilistic databases. In Sect. 5, we show how shared correlations can be used to speed up query evaluation. In Sect. 6, we develop novel approaches to generate elimination orders that maintain symmetry and help speed up query evaluation. In Sect. 7, we present a comprehensive experimental study comparing the different inference algorithms. In Sect. 8, we survey related work and we conclude with Sect. 9.

Preliminary portions of this article have been presented in various conferences [40–42]. In contrast to the previous papers, in this article, we present our approach in full generality and in greater detail. Also, we present a new approach to determining the elimination order that preserves symmetry in the rv-elim graph (Sect. 6.2) and present a more comprehensive set of experiments evaluating our approach.

2 Background: graphical models overview

Our proposed PrDB model builds upon research on graphical models from the statistics and machine learning communities [9,33]. Graphical models provide a flexible and yet compact representation for complex probability distributions. There is also a wide literature on effective computation of various marginal and a posteriori probabilities (referred to as *probabilistic inference* in the graphical models literature), and learning (learning the structure and estimating the probabilities from observational data). In addition, there has been work on first-order models, which allow even more compact modeling of the probability distributions [20,37].

In this section, we provide a quick primer on graphical models. We begin by introducing the concepts of random variables and factors, the basic building blocks for most uncertainty modeling techniques. Let X denote a random variable that can be assigned any value from a predefined *domain* of assignments denoted by $dom(X)$.

Definition 2.1 (*Factor*) A factor $f(\mathbf{X})$ is a function over a (small) set of random variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ such that $f(\mathbf{x}) \geq 0,\ \forall \mathbf{x} \in dom(X_1) \times \cdots \times dom(X_n)$.

One simple and intuitive uncertainty modeling technique is to directly model a *joint probability distribution* (*pdf*) in the obvious manner. Let $\mathscr{X} = (X_1, \ldots X_n)$ denote a list of random variables whose values are unknown and let $dom(\mathscr{X})$ denote $dom(X_1) \times \cdots \times dom(X_n)$. Then the joint probability, $\Pr : dom(\mathscr{X}) \to \Re_{\geq 0}$ such that $\sum_{\mathbf{x} \in dom(\mathscr{X})} \Pr(\mathbf{x}) = 1$, can be explicitly represented by using a factor that takes $\mathscr{X}$ as argument providing for each joint assignment $\mathbf{x}$ the corresponding probability $\Pr(\mathbf{x})$. One can then use this factor to perform various computations such as:

- compute the most probable joint assignment or most probable explanation (MPE), $\operatorname{argmax}_{\mathbf{x}} \Pr(\mathbf{x})$,
- given assignments to a subset of random variables $\mathbf{X} \subset \mathscr{X}$, compute the most probable assignment to the rest (also called maximum a posteriori or MAP estimation),
- compute the distribution for a single random variable X summed over the rest of the random variables (a marginal probability computation).

Unfortunately, since the space required to store a factor with n arguments is proportional to the product of the corresponding domain sizes, the approach of representing the joint distribution with a single explicit factor is rarely feasible. Even if the random variables are all binary, the space required for storing the joint distribution would be 2^n (the probabilities of the 2^n different joint assignments to the variables).

This has led to various approaches for *compactly* modeling uncertainty for large collections of random variables. We review two of the most common and most widely applicable approaches, *Bayesian networks* and *Markov networks*. The intuition behind both approaches is to break the joint distribution into a product of many smaller distributions over smaller sets of random variables. The basis for such a breakup lies in exploiting *conditional independences*:

Definition 2.2 (*Conditional independence*) Let $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ be disjoint sets of random variables such that $\mathscr{X} = \mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z}$. $\mathbf{X}$ is *conditionally independent* of $\mathbf{Y}$ given $\mathbf{Z}$ (denoted $\mathbf{X} \perp \mathbf{Y} | \mathbf{Z}$) in distribution Pr if:

$$\Pr(\mathbf{x}, \mathbf{y}|\mathbf{z}) = \Pr(\mathbf{x}|\mathbf{z}) \Pr(\mathbf{y}|\mathbf{z})$$

for all values $\mathbf{x} \in dom(\mathbf{X})$, $\mathbf{y} \in dom(\mathbf{Y})$ and $\mathbf{z} \in dom(\mathbf{Z})$, where $Pr(\mathbf{x}|\mathbf{z}) = Pr(\mathbf{x}, \mathbf{z})/Pr(\mathbf{z})$.

The utility of exploiting conditional independence lies in the fact that, usually, storing $Pr(\mathbf{x}|\mathbf{z})$ and $Pr(\mathbf{y}|\mathbf{z})$ is much cheaper than storing $Pr(\mathbf{x}, \mathbf{y}|\mathbf{z})$. Assuming all random variables have domain size d, storing $Pr(\mathbf{x}|\mathbf{z})$ and $Pr(\mathbf{y}|\mathbf{z})$ requires $d^{|\mathbf{X}|+|\mathbf{Z}|} + d^{|\mathbf{Y}|+|\mathbf{Z}|}$ units of space whereas storing $Pr(\mathbf{x}, \mathbf{y}|\mathbf{z})$ requires $d^{|\mathbf{X}|+|\mathbf{Y}|+|\mathbf{Z}|}$ units of space. It is easy to see that, barring a few degenerate cases ($|\mathbf{X}| = 0$ or $|\mathbf{Y}| = 0$ or $|\mathbf{X}| = |\mathbf{Y}| = 1$ and $d = 2$), the former is strictly smaller than the latter, often by a large amount.

2.1 Directed graphical models

Directed graphical models, popularly called *Bayesian networks* [33], are typically used to represent causal or asymmetric interactions amongst a set of random variables. In Bayesian networks, a special type of factor is used to break up the joint pdf into numerous small distributions:

Definition 2.3 (*Conditional probability factor*) Let $f(X|\Pi)$ denote a factor that takes as arguments $X \cup \Pi$ where Π denotes a set of random variables such that:

$$\sum_{x \in dom(X)} f(x|\pi) = 1, \quad \forall \pi \in dom(\Pi)$$

X is referred to as the child and Π, as its parents.

Note that Π can be the empty set. Intuitively, a conditional probability factor defines a conditional probability distribution over the child given an assignment to its parents. Often the conditional probability factor is represented as a table, and the factor is often referred to as a *conditional probability table*, but this is not a requirement; any function which performs the mapping is possible. For example, decision trees or special functional forms such as noisy-or and noisy-max are often used in practice.

Moreover, conditional probability factors can be used to enforce *deterministic constraints*:

Definition 2.4 (*Deterministic conditional probability factor*) A conditional probability factor $f(X|\Pi)$ is a deterministic factor if $\exists x \in dom(X)$ s.t. $f(x|\pi) = 1$, $\forall \pi \in dom(\Pi)$.

We make extensive use of such deterministic factors during query evaluation over probabilistic databases. Figure 5d shows an example of such a factor that enforces $X \Leftrightarrow Y$.

Having defined a set of such conditional probability factors, the complete joint distribution of the Bayesian network is then given by the product of all the factors:

$$\Pr(x_1, \ldots, x_n) = \prod_{i=1}^{n} f(x_i|\pi_i)$$

It is easy to show that the $\Pr(x_1, \ldots, x_n)$, as defined above, sums to 1 and is a normalized (legal) distribution.

Traditionally, a Bayesian network is represented as a directed acyclic graph (DAG) in which vertices denote random variables and a directed edge from random variable X_i to random variable X_j in the graph indicates that X_i directly influences (i.e., is a parent of) X_j. Figure 2 shows a simple example Bayesian network, showing both the graph structure and the conditional probability factors, that models the *model, type, m.p.g.* and *color* of a car. The graph also compactly represents the collection of conditional independences that hold in the distribution. Denote a random variable X_j as non-descendant of X_i if there does not exist a directed path from X_i to X_j. Then, X_i is conditionally independent of non-descendant X_j given parents of X_i. For instance, in Fig. 2, *m.p.g.* $\perp$ *color* | *model*. The complete details of computing all of the conditional independences are beyond the scope of our quick introduction; we refer the interested reader to Pearl [33].

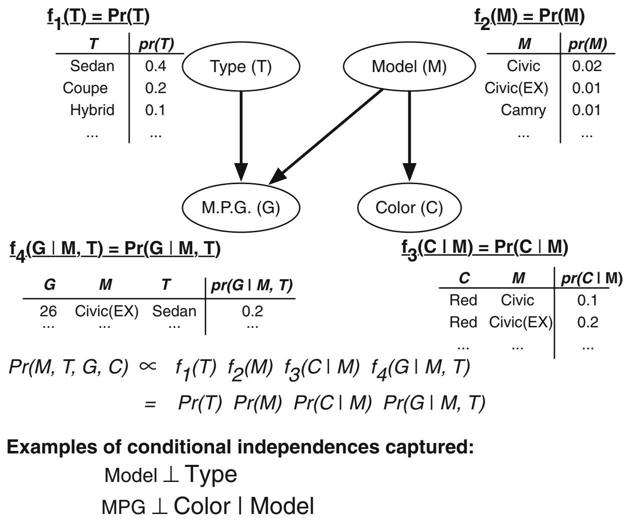


Fig. 2 Example of a directed model with four random variables

2.2 Undirected graphical models

Undirected graphical models, or *Markov networks* [9], are useful for representing distributions over variables where there is no natural directionality to the influence of one variable over another and where the interactions are more symmetric. In their most general form, Markov networks are defined as a product of a set of factors or *clique potentials* (as they are more popularly known) with a normalizing constant:

$$\Pr(\mathbf{x}) = \frac{1}{\mathscr{Z}} \prod_{i=1}^{n} f_i(\mathbf{x}_i)$$

where $\mathbf{x}_i$ denotes the assignments to arguments of f_i.

As opposed to a Bayesian network where each conditional probability factor denotes the distribution over a child conditioned on some (joint) assignment to its parents, in a Markov network each factor or clique potential denotes a *compatibility function* over all its arguments. The compatibility function values must be non-negative, but other than that they can be arbitrary, i.e., the entries are not required to sum to 1. Also, whereas Bayesian networks are represented by directed graphs, Markov networks are represented graphically using undirected edges denoting that the dependencies are symmetric in nature. More precisely, given a Markov network, its corresponding graphical representation contains an edge between random variables X_i and X_j if there exists a factor f_k with arguments $\mathbf{X}$ such that $X_i, X_j \in \mathbf{X}$. Again, just as in Bayesian networks, the conditional independences can be read off the graphical representation. Essentially, X_i and X_j are conditionally independent given a set of random variables $\mathbf{X}$ if every undirected path from X_i to X_j contains at least one variable from $\mathbf{X}$. Note that unlike Bayesian networks a Markov network can contain cycles. We refer the interested reader to Cowell et al. [9] for further details.

2.3 Probabilistic graphical models: general formulation

Besides Bayesian networks and Markov networks, there have been attempts to combine the two approaches such as *chain graph models* [9] and *directed factor graphs* [19]. The directed factor graph formalism is sometimes preferred over chain graphs because their representation can express arbitrary factorization of the joint probabilistic model; the graphical representation of directed factor graphs contains vertices explicitly representing the factors in addition to vertices representing random variables and is thus more detailed. Both chain graphs and directed factor graphs allow a mix of directed and undirected dependencies but neither allows probabilistic models whose underlying uncertainty structure contains directed cycles (although generalizations that allow such structures have also been studied [38]).

In the rest of this article, we adopt a fairly general uncertainty model that makes no assumptions about the underlying uncertainty structure. We will refer to this model as a PGM. A PGM is completely described by providing a list of random variables $\mathscr{X}$ and a set of factors $\mathscr{F} = \{f_1, \dots f_m\}$:

Definition 2.5 A PGM $\mathscr{P} = \langle \mathscr{F}, \mathscr{X} \rangle$ defines a joint distribution over the list of random variables $\mathscr{X} = (X_1, \dots X_n)$ via a set of factors $\mathscr{F}$, each defined over a subset of $\mathscr{X}$. Given a complete joint assignment $\mathbf{x} \in dom(\mathscr{X})$ to the variables in $\mathscr{X}$, the joint distribution is defined by: $\Pr(\mathbf{x}) = \frac{1}{\mathscr{Z}} \prod_{f \in \mathscr{F}} f(\mathbf{x}_f)$, where $\mathbf{x}_f$ denotes the assignments restricted to the arguments of f and $\mathscr{Z} = \sum_{\mathbf{x}'} \prod_{f \in \mathscr{F}} f(\mathbf{x}'_f)$ is a normalization constant referred to as the partition function[1].

Note that, as opposed to the previous definitions for joint probability distributions that we described for Bayesian networks and Markov networks, the above definition allows the use of both conditional probability factors (directed dependencies) and clique potentials (undirected dependencies), and so is completely general.

We next move on to our discussion of probabilistic databases, and how the above general formulation of a PGM can be used to model data uncertainty.

3 PrDB model

We begin with some notation. 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 corresponding to $t \in R$ and $a \in attr(R)$. Besides

[1] Note that since we allow $f(\mathbf{x}) = 0$, there is a possibility that $\mathscr{Z} = 0$ but this only happens when $\Pr(\mathbf{x}) = 0,\ \forall \mathbf{x}$. As long as there is at least one $\mathbf{x}$ such that $\Pr(\mathbf{x}) > 0$, this is not an issue.

(a)

S	**A**	**B**	Pr
s_1	a_1	2	0.8
s_2	a_2	2	0.8

T	**B**	**C**
t_1	{2: 0.6, 3: 0.4}	c

(b)

possible world	probability
$d_1 = \{s_1.e/\texttt{t}, s_2.e/\texttt{t}, t_1.\mathbf{B}/2\}$	0.384
$d_2 = \{s_1.e/\texttt{t}, s_2.e/\texttt{t}, t_1.\mathbf{B}/3\}$	0.256
$d_3 = \{s_1.e/\texttt{t}, s_2.e/\texttt{f}, t_1.\mathbf{B}/2\}$	0.096
$d_4 = \{s_1.e/\texttt{t}, s_2.e/\texttt{f}, t_1.\mathbf{B}/3\}$	0.064
$d_5 = \{s_1.e/\texttt{f}, s_2.e/\texttt{t}, t_1.\mathbf{B}/2\}$	0.096
$d_6 = \{s_1.e/\texttt{f}, s_2.e/\texttt{t}, t_1.\mathbf{B}/3\}$	0.064
$d_7 = \{s_1.e/\texttt{f}, s_2.e/\texttt{f}, t_1.\mathbf{B}/2\}$	0.024
$d_8 = \{s_1.e/\texttt{f}, s_2.e/\texttt{f}, t_1.\mathbf{B}/3\}$	0.016

(c)

S	**A**	**B**
s_1	a_1	2

T	**B**	**C**
t_1	2	c

(d)

world	probability
d_1	0.576
d_2	0.064
d_3	0.144
d_4	0.016
d_5	0.032
d_6	0.128
d_7	0.008
d_8	0.032

Fig. 3 **a**, **b** A small probabilistic database and its possible worlds (`t` denotes `true`, `f` denotes `false`). **c** Possible world d_3. **d** Distribution with the *implies* dependency

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 3.1 A *probabilistic database* PrDB $\mathscr{D} = \langle \mathscr{R}, \mathscr{P} \rangle$ is a pair, where $\mathscr{R}$ is a set of relations and $\mathscr{P}$ denotes a PGM over the random variables associated with the tuples in $\mathscr{R}$.

Semantics Let $\mathscr{X}$ denote the random variables associated with the relations in PrDB $\mathscr{D}$. Note that a complete joint assignment to $\mathscr{X}$ where we assign each $X \in \mathscr{X}$ with a value from the corresponding domain $dom(X)$ produces a "traditional" deterministic database devoid of any uncertainty. Thus the PGM $\mathscr{P}$ defines a distribution over numerous deterministic databases, and gives us the familiar possible worlds semantics [11,25] for our PrDB model. Each possible world is a complete joint assignment to $\mathscr{X}$, we denote a specific assignment by $\mathbf{x}$, where $\mathbf{x} \in \times_{X \in \mathscr{X}} dom(X)$. The probability associated with the possible world corresponding to $\mathbf{x}$ is given by the distribution defined by the PGM $\mathscr{P}$ (Definition 2.5), i.e., by multiplying the numbers returned by the factors in $\mathscr{P}$ for $\mathbf{x}$ and dividing by the partition function.

Let us now consider a simple running example that will help ground these definitions.

Example 2 (Running example) Consider the small two-relation database shown in Fig. 3a where the first relation S contains uncertain tuples and the second relation, T, contains a tuple with an uncertain attribute. Each tuple in S can exist with a certain (existence) probability, shown next to the tuples in the figure. 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. For instance, $t_1.\mathbf{B}$ can be assigned the value 2 with probability 0.6 and the value 3 with probability 0.4. We represent uncertain tuples and uncertain attributes using random variables. We represent the probability distributions associated with the random variables (which may be involved in correlations) using factors (Definition 2.1). For our example, we will also assume complete independence among all random variables. For the three random variables in Fig. 3a, we would define factors $f_{s_1}(s_1.e)$, $f_{s_2}(s_2.e)$ and $f_{t_1}(t_1.\mathbf{B})$:

$s_1.e$	f_{s_1}
`false`	0.2
`true`	0.8

$s_2.e$	f_{s_2}
`false`	0.2
`true`	0.8

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

where in $f_{s_1}(s_1.e)$ and $f_{s_2}(s_2.e)$ we use the assignment `true` to denote that the tuple exists and `false` to denote that the tuple is absent. The PGM defining the full joint distribution is then given by the product of these three factors:

$$\Pr(x_{s_1.e}, x_{s_2.e}, x_{t_1.\mathbf{B}}) = f_{s_1}(x_{s_1.e}) f_{s_2}(x_{s_2.e}) f_{t_1}(x_{t_1.\mathbf{B}})$$

Each possible world is then obtained by assigning all three random variables $s_1.e$, $s_2.e$ and $t_1.\mathbf{B}$ assignments from their respective domains. Since each of them can take 2 values, there are $2^3 = 8$ possible worlds. Figure 3b shows all eight possible worlds with their corresponding probabilities. In Fig. 3b, x/c denotes that random variable x is substituted with $c \in dom(X)$. Thus, for d_1, $s_1.e/\texttt{t}$ denotes that s_1 is present in possible world d_1 and so on. The probability associated with each possible world is obtained by multiplying the appropriate numbers returned by the factors and normalizing if necessary. For instance, for the possible world obtained by the assignment $d_1 = \{s_1.e/\texttt{t}, s_2.e/\texttt{t}, t_1.\mathbf{B}/2\}$, the probability is $0.8 \times 0.8 \times 0.6 = 0.384$. Figure 3c shows one of the possible worlds, d_3.

3.1 Representing correlations

As mentioned earlier, in the running example, we assumed complete independence which is why we could express our uncertainty using single argument factors. Let us now see how to represent correlations. This is where the flexibility of defining a probabilistic database in terms of PGMs becomes apparent. For instance, consider the previous example, but on this occasion we will try to represent an *implies* dependency between random variables $s_1.e$ and $t_1.\mathbf{B}$. More precisely, we would like to enforce that when s_1 exists $t_1.\mathbf{B}$ is assigned 2 with probability 0.9, otherwise $t_1.\mathbf{B}$ is assigned 3 with probability 0.8. We can achieve this by defining a factor, f_{implies}, that *takes both* $s_1.e$ *and* $t_1.\mathbf{B}$ *as arguments*:

$s_1.e$	$t_1.\mathbf{B}$	f_{implies}
`false`	2	0.2
`false`	3	0.8
`true`	2	0.9
`true`	3	0.1

Note that f_{implies}, by its construction, is actually a conditional probability factor. The PGM distribution is now given by:

$$\Pr(x_{s_1.e}, x_{s_2.e}, x_{t_1.\mathbf{B}}) = f_{s_1}(x_{s_1.e}) f_{s_2}(x_{s_2.e}) \\ f_{\text{implies}}(x_{s_1.e}, x_{t_1.\mathbf{b}})$$

Figure 3d shows the distribution over the possible worlds using the new PGM. Notice how possible worlds with favorable assignments ($\{s_1.e/\mathtt{t}, t_1.\mathbf{B}/2\}$ or $\{s_1.e/\mathtt{f}, t_1.\mathbf{B}/3\}$) map to relatively higher probabilities. More precisely, the probability mass assigned to these favorable assignments is $\Pr(d_1) + \Pr(d_3) + \Pr(d_6) + \Pr(d_8) = 0.88$. Also, note that the two distributions depicted in Fig. 3b, d are very different.

The above example illustrates that including different correlations into the uncertainty model can produce starkly differing distributions over the possible worlds. Presumably, the results of evaluating a query on these databases will also provide differing results (in fact, we will demonstrate this in the next section where we show the results of running a query with and without the *implies* dependency on our example database). Thus, it is imperative to model correlations accurately, if and when the application demands it. Also, since we do not impose any restrictions over which random variables can appear in factors, our representation for modeling uncertainty is completely general. Thus, if the user so wishes, she may define a factor containing random variables from the same tuple, different tuples, tuples from different relations or tuple existence and attribute value random variables. Further, it is not necessary to introduce random variables where they are not needed. For instance in the above example, since we knew t_1 exists with certainty, we did not include $t_1.e$ in any of the factors.

3.2 Storing uncertainty models

Most earlier work on probabilistic databases represents probabilistic relations by storing uncertainty with each tuple in isolation [11,13,35]. This is inadequate for our purposes since the same tuple's random variables can be involved in multiple factors, and the same factor can be associated with different random variables. Our approach to storing PGMs in a database decouples the uncertainty model from the data. Figure 4 shows how we store the factors and associate them with the tuples in our current prototype implementation. Essentially, the idea is to have relations that store factors in a normalized fashion so that we minimize replication. Further, recall that in Sect. 1 we discussed how shared correlations lead to the same correlation showing up repeatedly in the database. Since we represent correlations using factors, this implies that the same factor may need to be represented multiple times. In what follows, we discuss our approach that stores such shared factors in an intelligent fashion so that such repetition is minimized.

(a)

tid	A	B	e
s_1	a_1	2	$\perp$
s_2	a_2	2	$\perp$

tid	B	C	e
t_1	$\perp$	c	`t`

(b)

fid	rv	pos
f_{s_1}	$s_1.e$	1
f_{s_2}	$s_2.e$	1
f_{implies}	$s_1.e$	1
f_{implies}	$t_1.\mathbf{B}$	2

fid	funcid
f_{s_1}	ϕ_1
f_{s_2}	ϕ_1
f_{implies}	ϕ_2

(c)

funcid	func
ϕ_1	$\{[\mathtt{f}]: 0.2, [\mathtt{t}]: 0.8\}$
ϕ_2	$\{[\mathtt{f}, 2]: 0.2, [\mathtt{f}, 3]: 0.8, [\mathtt{t}, 2]: 0.9, [\mathtt{t}, 3]: 0.1\}$

(d)

rv	domid
$s_1.e$	δ_1
$s_2.e$	δ_1
$t_1.\mathbf{B}$	δ_2

domid	dom
δ_1	$\{\mathtt{f}, \mathtt{t}\}$
δ_2	$\{2, 3\}$

Fig. 4 **a** The base tables for the running example. **b** `f2args` and `f2funcid`. **c** `funcid2func`. **d** `rv2domid` and `domid2dom`. See text for descriptions of the various relations

We first formally define a shared factor. We begin by taking a closer look at a factor (Definition 2.1). A factor is composed of two separate components: its argument, which is an ordered list of random variables, and the function component, which maps joint assignments to the arguments to corresponding outputs. For instance, $f_{s_1}(s_1.e)$ defined earlier, consists of the argument $s_1.e$ and its function component is simply the table that maps `false` to 0.2 and `true` to 0.8.

Definition 3.2 Two factors f_1 and f_2 are *shared factors*, denoted $f_1 \cong f_2$, if they both have the same function component.

Thus, $f_{s_1}(s_1.e)$ and $f_{s_2}(s_2.e)$ defined for the running example form a pair of shared factors since both map `true` to 0.8.

Since shared factors share their function components, we store them such that the function component of each set of shared factors is stored only once. We now describe how we would store the database with the *implies* dependency introduced earlier:

– With each base tuple we store a unique id (**tid**), so that we can refer to the tuple's random variables using the id. To indicate that an attribute value is uncertain or a tuple's existence is uncertain, we use a special symbol $\perp$. Figure 4a shows the base relations for our example.

- For each factor, we make an entry into a special relation `f2args` that contains three attributes: the id of the factor (**fid**), its argument (a random variable) and the position of the argument. Figure 4b shows `f2args` for our example. Thus, we have two rows for f_{implies} since it contains two arguments: $s_1.e$ and $t_1.\mathbf{B}$.
- We store the factor functions by first mapping each factor to a function's id in the relation `f2funcid` (Fig. 4b) and then mapping the function id to the function itself stored in a serialized form in the relation `funcid2func` (Fig. 4c). This way we avoid repeated storing of shared factors' functions. For instance, in Fig. 4b, both f_{s_1} and f_{s_2} map to the same function id ϕ_1 and ϕ_1 is stored once in Fig. 4c.
- We also store the domains of random variables in a similar manner, avoiding repeated storing of random variables' domains. This is achieved by utilizing two special relations: `rv2domid` maps each random variable to a domain id, and `domid2dom` maps each domain id to the corresponding domain stored in a serialized form. Thus, for our example in Fig. 4d, $s_1.e$ and $s_2.e$ both map to domain id δ_1. δ_1 is then mapped to its corresponding domain and stored exactly once.

4 Query evaluation in PrDB

We now move our discussion to query evaluation. A key advantage of associating possible world semantics with a probabilistic database is that it lends precise semantics to the query evaluation problem. Given a user-submitted query q (expressed in some standard query language such as relational algebra or SQL) and a probabilistic database $\mathscr{D}$, the result of evaluating q against $\mathscr{D}$ is defined to be the set of results obtained by evaluating q against each possible world of $\mathscr{D}$, augmented with the probabilities of the possible worlds. Since every possible world looks exactly like a traditional database, running q on a possible world is a well-defined operation. Thus, the above definition of evaluating q on a probabilistic database is both clear and precise. However, since the number of possible worlds is typically exponential in the number of tuples, it is not feasible to return the entire set of results to the user. Instead it is traditional to combine the results from the different possible worlds into a compressed form before presenting it to the user. One way to do that is to compute the marginal probability for each result tuple by summing the probabilities of the possible worlds that return that tuple:

$$\mu(t) = \sum_{\substack{\mathbf{x}\in dom(\mathscr{X}) \\ t\in q(\mathscr{D}[\mathscr{X}/\mathbf{x}])}} \Pr(\mathbf{x}),$$

where t denotes a result tuple and $q(\mathscr{D}[\mathscr{X}/\mathbf{x}])$ denotes the result obtained by evaluating q on the possible world obtained by substituting $\mathscr{X}$ with $\mathbf{x}$.

Relating back to our earlier examples, suppose we want to run the query $q = \prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$. The result consists of a single tuple $r = \boxed{\begin{array}{c}\mathbf{C}\\ \hline c\end{array}}$. Figure 5a shows the result of the query when it is run against each possible world of the database from Fig. 3b. The possible worlds which return r are d_1, d_3 and d_5. Thus, the probability associated with the result tuple will be the sum $\Pr(d_1) + \Pr(d_3) + \Pr(d_5)$. More precisely, the result tuple's probability is 0.576 for the database with independent random variables (Fig. 3b) and 0.752 for the database with the *implies* dependency (Fig. 3d).

Next we make a connection between query evaluation and probabilistic inference, more specifically, the marginal probability computation problem. Given a PGM $\mathscr{P} = \langle\mathscr{F}, \mathscr{X}\rangle$ and a random variable $X \in \mathscr{X}$, the marginal probability distribution for X is defined as:

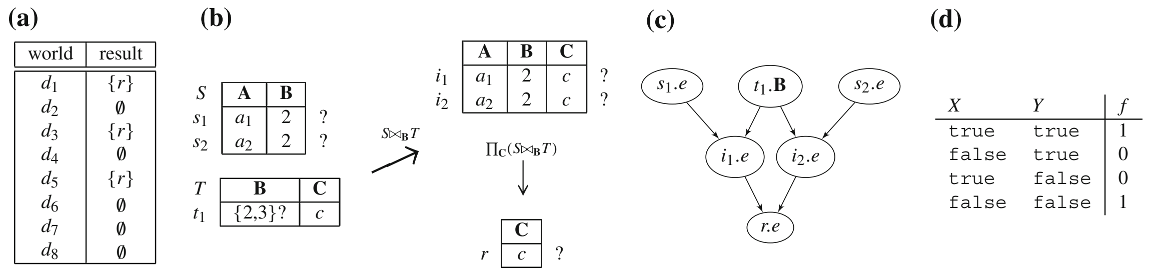
$$\mu(X = x) = \frac{1}{\mathscr{Z}} \sum_{\mathscr{X}\setminus X} \prod_{f\in\mathscr{F}} f(\mathbf{X}_f), \quad \forall x \in dom(X) \qquad (1)$$

where $\mathbf{X}_f$ denotes the arguments to factor f. Note that, from Definition 2.5, $\mathscr{Z} = \sum_{\mathscr{X}} \prod_{f\in\mathscr{F}} f(\mathbf{X}_f) = \sum_X \sum_{\mathscr{X}\setminus X} \prod_{f\in\mathscr{F}} f(\mathbf{X}_f)$ which implies that any probabilistic inference algorithm that computes $\sum_{\mathscr{X}\setminus X} \prod_{f\in\mathscr{F}} f(\mathbf{X}_f)$ can be used to compute $\mathscr{Z}$ by performing an extra summation.

We next illustrate this connection using our running example, and then we discuss the steps required to evaluate general relational algebra queries. As we shall see subsequently, while evaluating queries we make extensive use of deterministic conditional probability factors (Definition 2.4).

Example 3 (Query evaluation) Our query evaluation approach is very similar to query evaluation in traditional database systems and is depicted in Fig. 5b. Just as in traditional database query processing, in Fig. 5b, we introduce intermediate tuples produced by the join (i_1 and i_2) and produce the result tuple (r) from the projection operation. What makes query processing for probabilistic databases different from traditional database query processing is the fact that we need to preserve the correlations among the random variables representing the intermediate and result tuples and the random variables representing the tuples they were produced from. In our example, there are three such correlations:

- i_1 (produced by the join between s_1 and t_1) exists or $i_1.e$ is `true` only in those possible worlds where both $s_1.e$ is `true` and $t_1.\mathbf{B}$ is assigned the value 2.
- Similarly, $i_2.e$ is `true` in possible worlds where both $s_2.e$ is `true` and $t_1.\mathbf{B}$ is assigned the value 2.



world	result
d_1	$\{r\}$
d_2	$\emptyset$
d_3	$\{r\}$
d_4	$\emptyset$
d_5	$\{r\}$
d_6	$\emptyset$
d_7	$\emptyset$
d_8	$\emptyset$

X	Y	f
`true`	`true`	1
`false`	`true`	0
`true`	`false`	0
`false`	`false`	1

Fig. 5 Evaluating $\prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$ on the example (Fig. 3a). **a** Results from possible worlds semantics. **b** Running the query on the database and **c** the corresponding augmented PGM. **d** A deterministic conditional probability factor enforcing $X \Leftrightarrow Y$

– Finally, r (the result tuple produced by the projection) exists in worlds that produce at least one of i_1 or i_2 or both.

To enforce these correlations, we introduce *deterministic* intermediate factors defined over appropriate random variables (Fig. 5c):

- For the correlation among $i_1.e$, $s_1.\mathbf{B}$ and $t_1.\mathbf{B}$, we introduce the factor f_{i_1} which is defined as:

$$f_{i_1}(i_1.e|s_1.e, t_1.\mathbf{B}) = \begin{cases} 1 & \text{if } i_1.e \Leftrightarrow (s_1.e \wedge (t_1.\mathbf{B} == 2)) \\ 0 & \text{otherwise} \end{cases}$$

- Similarly, for the correlation among $i_2.e$, $s_2.e$ and $t_1.\mathbf{B}$, we introduce the factor f_{i_2} which is defined as:

$$f_{i_2}(i_2.e|s_2.e, t_1.\mathbf{B}) = \begin{cases} 1 & \text{if } i_2.e \Leftrightarrow (s_2.e \wedge (t_1.\mathbf{B} == 2)) \\ 0 & \text{otherwise} \end{cases}$$

- For the correlation among $r.e$, $i_1.e$ and $i_2.e$, we introduce a factor f_r capturing the `or` semantics:

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

Now, to compute the probability of existence of r, we simply need to compute the marginal probability (Eq. 1) associated with the assignment $r.e =$ `true` from PGM formed by the set of factors in the base data and the factors introduced during query evaluation (the augmented PGM). For the example where we assumed complete independence (Fig. 3a), our augmented PGM is given by the collection f_{s_1}, f_{s_2}, f_{t_1}, f_{i_1}, f_{i_2} and f_r (Fig. 5c), and to compute the marginal probability we can simply use any of the exact inference algorithms available in the literature [16,26,47].

For instance, the variable elimination algorithm [16,47] is a particularly simple yet efficient inference algorithm that computes the marginal probability by multiplying all factors in the (augmented) PGM and summing over (eliminating) all random variables except for the random variable of interest. In the case of our example it would compute:

$$\mu(r.e) = \sum_{i_1.e, i_2.e, t_1.\mathbf{B}} f_{t_1}(t_1.\mathbf{B}) \sum_{s_1.e} f_{s_1}(s_1.e) f_{i_1}(i_1.e|s_1.e, t_1.\mathbf{B}) \times f_r(r.e|i_1.e, i_2.e) \sum_{s_2.e} f_{s_2}(s_2.e) f_{i_2}(i_2.e|s_2.e, t_1.\mathbf{B}) \tag{2}$$

It is easy to check that the above computation produces the same marginal probability we computed earlier via enumerating over possible worlds.

4.1 Generating factors for general queries

Query evaluation for general relational algebra queries also follows the same basic ideas. In what follows, we modify the traditional relational algebra operators so that they not only generate intermediate tuples but also introduce intermediate factors which, combined with the factors on the base data, provide a PGM that can then be used to compute marginal probabilities of the random variables associated with the result tuples of interest. We next describe the modified σ, $\times$, $\prod$, δ (duplicate elimination), $\cup$, $-$ and γ (aggregation) operators where we use $\emptyset$ to denote a special "null" symbol.

Select Let $\sigma_c(R)$ denote the query we are interested in, where c denotes the predicate of the select operation. Every tuple $t \in R$ can be jointly instantiated with values from $\times_{a \in attr(R)} dom(t.a)$. If none of these instantiations satisfy c then t does not give rise to any result tuple. If even a single instantiation satisfies c, then we generate an intermediate tuple r that maps attributes from R to random variables and is associated with a tuple existence random variable $r.e$. We then introduce factors encoding the correlations among the random variables for r and the random variables for t. The first factor we introduce encodes the correlations for $r.e$:

$$f^{\sigma}_{r.e}(r.e|t.e, \{t.a\}_{a \in attr(R)}) = \begin{cases} 1 & \text{if } t.e \wedge c(\{t.a\}_{a \in attr(R)}) \Leftrightarrow r.e \\ 0 & \text{otherwise} \end{cases}$$

where $c(\{t.a\}_{a \in attr R})$ is $\texttt{true}$ iff a joint assignment to the attribute value random variables of t satisfies the predicate c.

We also introduce a factor for $r.a$, $\forall a \in attr(R)$ (where $dom(r.A) = dom(t.A)$), denoted by $f^{\sigma}_{r.a}$. $f^{\sigma}_{r.a}$ takes $t.a$, $r.e$ and $r.a$ as arguments and can be defined as:

$$f^{\sigma}_{r.a}(r.a|r.e, t.a) = \begin{cases} 1 & \text{if } r.e \wedge (t.a == r.a) \\ 1 & \text{if } \overline{r.e} \wedge (r.a == \emptyset) \\ 0 & \text{otherwise} \end{cases}$$

Cartesian product Suppose R_1 and R_2 are the two relations involved in the Cartesian product operation. Let r denote the join result of two tuples $t_1 \in R_1$ and $t_2 \in R_2$. Thus r maps every attribute from $attr(R_1) \cup attr(R_2)$ to a random variable, and is associated with a tuple existence random variable $r.e$. The factor for $r.e$, denoted by $f^{\times}_{r.e}$, takes $t_1.e$, $t_2.e$ and $r.e$ as arguments, and is defined as:

$$f^{\times}_{r.e}(r.e|t_1.e, t_2.e) = \begin{cases} 1 & \text{if } t_1.e \wedge t_2.e \Leftrightarrow r.e \\ 0 & \text{otherwise} \end{cases}$$

We also introduce a factor $f^{\times}_{r.a}$ for each $a \in attr(R_1) \cup attr(R_2)$, and this is defined exactly in the same fashion as $f^{\sigma}_{r.a}$. Basically, for $a \in attr(R_1)$ ($a \in attr(R_2)$), it returns 1 if $r.e \wedge (t_1.a == r.a)$ ($r.e \wedge (t_2.a == r.a)$) holds or if $\overline{r.e} \wedge (r.a == \emptyset)$ holds, and 0 otherwise.

Project (*without duplicate elimination*) Let $\prod_{\mathbf{a}}(R)$ denote the operation we are interested in where $\mathbf{a} \subseteq attr(R)$ denotes the set of attributes we want to project onto. Let r denote the result of projecting $t \in R$. Thus r maps each attribute $a \in \mathbf{a}$ to a random variable and is associated with $r.e$. $f^{\prod}_{r.e}$, the factor for $r.e$, takes $t.e$ and $r.e$ as arguments and is defined as follows:

$$f^{\prod}_{r.e}(r.e|t.e) = \begin{cases} 1 & \text{if } t.e \Leftrightarrow r.e \\ 0 & \text{otherwise} \end{cases}$$

Each factor $f^{\prod}_{r.a}$, introduced for $r.a$, $\forall a \in \mathbf{a}$, is defined exactly as $f^{\sigma}_{r.a}$, in other words, $f^{\prod}_{r.a}(r.a|r.e, t.a) = f^{\sigma}_{r.a}(r.a|r.e, t.a)$.

Duplicate elimination Duplicate elimination is a slightly more complex operation because it can give rise to multiple intermediate tuples even if there was only one input tuple to begin with. Let R denote the relation from which we want to eliminate duplicates, then the resulting relation after duplicate elimination will contain tuples whose existence is uncertain, more precisely the resulting tuples' attribute values are known. Any element from $\bigcup_{t \in R} \times_{a \in attr(R)} dom(t.a)$ may correspond to the values of a possible result tuple. Let r denote any such result tuple whose attribute values are known, only $r.e$ is not $\texttt{true}$ with certainty. Denote by r_a the value of attribute a in r. We only need to introduce the factor $f^{\delta}_{r.e}$ for $r.e$. To do this we compute the set of tuples from R that may give rise to r. Any tuple t that satisfies $\bigwedge_{a \in attr(R)}(r_a \in dom(t.a))$ may give rise to r. Let y^r_t be an intermediate random variable with $dom(y^r_t) = \{\texttt{true}, \texttt{false}\}$ such that y^r_t is $\texttt{true}$ iff t gives rise to r and $\texttt{false}$ otherwise. This is easily done by introducing a factor $f^{\delta}_{y^r_t}$ that takes $\{t.a\}_{a \in attr(R)}$, $t.e$ and y^r_t as arguments and is defined as:

$$f^{\delta}_{y^r_t}(y^r_t|\{t.a\}_{a \in attr(R)}, t.e) = \begin{cases} 1 & \text{if } t.e \wedge \bigwedge_a (t.a == r_a) \Leftrightarrow y^r_t \\ 0 & \text{otherwise} \end{cases}$$

where $\{t.a\}_{a \in attr(R)}$ denotes all attribute value random variables of t. We can then define $f^{\delta}_{r.e}$ in terms of y^r_t. $f^{\delta}_{r.e}$ takes as arguments $\{y^r_t\}_{t \in T_r}$, where T_r denotes the set of tuples that may give rise to r (contains the assignment $\{r_a\}_{a \in attr(R)}$ in its joint domain), and $r.e$, and is defined as:

$$f^{\delta}_{r.e}(r.e|\{y^r_t\}_{t \in T_r}) = \begin{cases} 1 & \text{if } \bigvee_{t \in T_r} y^r_t \Leftrightarrow r.e \\ 0 & \text{otherwise} \end{cases}$$

Union *and* **set difference** These operators require set semantics. Let R_1 and R_2 denote the relations on which we want to apply one of these two operators, either $R_1 \cup R_2$ or $R_1 - R_2$. We will assume that both R_1 and R_2 are sets of tuples such that every tuple contained in them have their attribute values fixed and the only uncertainty associated with these tuples are with their existence (if not then we can apply a δ operation to convert them to this form). Now, consider result tuple r and sets of tuples T^1_r, containing all tuples from R_1 that match r's attribute values, and T^2_r, containing all tuples from R_2 that match r's attribute values. The required factors for $r.e$ can now be defined as follows:

$$f^{\cup}_{r.e}(r.e|\{t_1.e\}_{t_1 \in T^1_r}, \{t_2.e\}_{t_2 \in T^2_r}) = \begin{cases} 1 & \text{if } (\bigvee_{t \in T^1_r \cup T^2_r} t.e) \Leftrightarrow r.e \\ 0 & \text{otherwise} \end{cases}$$

$$f^{-}_{r.e}(r.e|\{t_1.e\}_{t_1 \in T^1_r}, \{t_2.e\}_{t_2 \in T^2_r}) = \begin{cases} 1 & \text{if } \left((\bigvee_{t \in T^1_r} t.e) \wedge \neg(\bigvee_{t \in T^2_r} t.e)\right) \Leftrightarrow r.e \\ 0 & \text{otherwise} \end{cases}$$

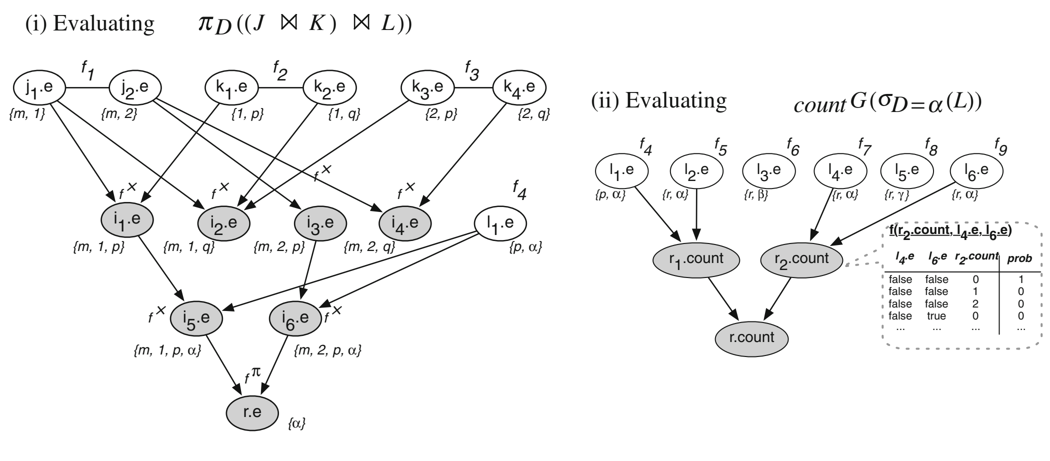
Aggregation operators Aggregation operators are also easily handled using factors. Suppose we want to compute the $\texttt{sum}$ aggregate on attribute a of relation R, then we simply define a random variable $r.a$ for the result and introduce a factor that takes as arguments $\{t.a\}_{a \in attr(R)}$ and $r.a$, and define the factor so that it returns 1 if $r.a == (\sum_{t \in R} t.a)$ and 0 otherwise. Thus for any aggregate operator γ and result tuple random variable $r.a$, we can define the following factor:

$$f^{\gamma}_{r.a}(r.a|\{t.a\}_{t \in R}) = \begin{cases} 1 & \text{if } r.a == \gamma_{t \in R} t.a \\ 1 & \text{if } (r.a == \emptyset) \Leftrightarrow \bigwedge_{t \in R}(t.a == \emptyset) \\ 0 & \text{otherwise} \end{cases}$$

J	A	B	*factors*
j_1	m	1	f_1
j_2	m	2	f_1

K	B	C	*factors*
k_1	1	p	f_2
k_2	1	q	f_2
k_3	2	p	f_3
k_4	2	q	f_3

L	C	D	*factors*
l_1	p	α	f_4
l_2	r	α	f_5
l_3	r	β	f_6
l_4	r	α	f_7
l_5	r	γ	f_8
l_6	r	α	f_9



$l_4.e$	$l_6.e$	$r_2.count$	prob
false	false	0	1
false	false	1	0
false	false	2	0
false	true	0	0
...	...	...	...

Fig. 6 (i) An example query evaluation over a three-relation database with only tuple uncertainty but many correlations (tuples associated with the same factor are correlated with each other, and the corresponding vertices in the *top* layer of the PGM are connected to each other). The intermediate tuples are shown alongside the corresponding random variables. Tuples $l_2, \ldots, l_6$ do not participate in the query. (ii) PGM constructed during the evaluation of $_{count}G(\sigma_{D=\alpha}(L))$ over the same probabilistic database. By exploiting decomposability of *count*, we can limit the maximum size of the newly introduced factors to 3 (the naive implementation would have constructed a 5-variable factor)

Figure 6(i) shows the PGM generated when evaluating a multi-way join query over three relations; computing the result tuple probability is equivalent to computing the marginal probability distribution over the random variable $r.e$.

4.2 Optimizations

For the above operator modifications, we have attempted to be completely general and hence the factors introduced may look more complicated than need be. For example, it is not necessary that $f^{\sigma}_{r.E}$ take as arguments all random variables $\{t.a\}_{a \in attr(R)}$ (as defined above); it only needs to take those $t.a$ random variables as arguments that are involved in the predicate c of the σ operation. Also, given a theta-join we do not need to implement this as a Cartesian product followed by a select operation. It is straightforward to push the select operation into the Cartesian product factors and implement theta-join directly by modifying $f^{\times}_{r.E}$ appropriately using c.

Another type of optimization that is extremely useful for aggregate computation, duplicate elimination and the set-theoretic operations ($\cup$ and $-$) is to exploit decomposable functions. A decomposable function is one whose result does not depend on the order in which the inputs are presented to it. For instance, $\vee$ is a decomposable function, and so are most of the aggregation operators including `sum`, `count`, `max` and `min`. The problem with some of the redefined relational algebra operators is that, if implemented naively, they may lead to large intermediate factors. For instance, while running a δ operation, if T_r contains n tuples for some r, then the factor $f^{\delta}_{r.e}$ will be of size 2^{n+1}. By exploiting decomposability of $\vee$ we can implement the same factor using a linear number of constant sized (three-argument) factors which may lead to significant speedups. We refer the interested reader to [39,48] for more details. The only aggregation operator that is not decomposable is `avg`, but in this case we can exploit the same ideas by implementing `avg` in terms of `sum` and `count` both of which are decomposable. Figure 6(ii) shows the PGM constructed for an aggregate query over a three-relation database.

4.3 Limitations of standard probabilistic inference

The complexity of probabilistic inference is estimated in terms of the *treewidth* [39] which is a natural parameter measuring the connectivity of the graph underlying the PGM. More precisely, the complexity of a probabilistic inference problem is exponential in its (induced) treewidth. The inference problem is easy if the PGM is a tree or closely resembles one, and the problem becomes progressively harder as the PGM deviates more from being a tree. In many cases, the treewidth of the PGM on which we need to run inference turns out to be small. On the other hand, there are cases when the treewidth is too large for us to run a standard inference algorithm. Often in such cases, it may be possible to reduce complexity of inference by exploiting other aspects of the PGM. Let us first take a closer look at a standard inference algorithm and then we will discuss how its performance can be improved.

4.3.1 *Variable elimination* [16,47]

As indicated earlier in this section, VE is a simple and standard probabilistic inference algorithm that can be used to compute marginal probability distributions. Perhaps the most surprising aspect of VE, given its simplicity, is that it still achieves the best known complexity bounds of the inference problem. In a nutshell, VE takes a PGM $\mathscr{P}$, a random variable X and an elimination order $\mathscr{O}$ as input, and returns the marginal probability distribution $\mu(X)$ computed from $\mathscr{P}$ by summing over random variables in order of appearance in $\mathscr{O}$. The step where VE tries to improve complexity is by pushing in summations so we sum over (eliminate) a random variable by multiplying only those factors that involve it as an argument.

Going back to our example (Eq. 2), recall that we needed to eliminate five random variables by multiplying six factors. Assume we chose the elimination order $\mathscr{O} = \{s_1.e, s_2.e, t_1.\mathbf{B}, i_1.e, i_2.e\}$ (variables are eliminated left to right). To eliminate $s_1.e$, VE would first multiply factors $f_{s_1}(s_1.e)$ and $f_{i_1}(i_1.e|s_1.e, t_1.\mathbf{B})$, and then sum out $s_1.e$ from the product to produce the new factor $m_{s_1.e}(i_1.e, t_1.\mathbf{B})$ (the subscript denotes the variable being summed out). Similarly, to eliminate $s_2.e$ (the next random variable in $\mathscr{O}$), VE would multiply $f_{s_2}(s_2.e)$ and $f_{i_2}(i_2.e|s_2.e, t_1.\mathbf{B})$, and then eliminate $s_2.e$ to produce $m_{s_2.e}(i_2.e, t_1.\mathbf{B})$.

4.3.2 *Naive inference algorithms and shared factors*

The main issue with VE (or any other standard probabilistic inference algorithm) is that it does not exploit shared factors. Recall that two factors are shared if they share the same function component (Definition 3.2). For instance, for our running example (Eq. 2), in the process of eliminating $s_1.e$ and $s_2.e$ we produced intermediate factors $m_{s_1.e}(i_1.e, t_1.\mathbf{B})$ and $m_{s_2.e}(i_2.e, t_1.\mathbf{B})$. If we take a closer look at both of these factors, we will notice that they both map exactly the same inputs to the same outputs and thus form a pair of shared factors:

$i_1.e$	$t_1.\mathbf{B}$	$m_{s_1.e}$
true	2	0.8
true	3	0
false	2	0.2
false	3	1

$i_2.e$	$t_1.\mathbf{B}$	$m_{s_2.e}$
true	2	0.8
true	3	0
false	2	0.2
false	3	1

In hindsight, it is not really surprising that $m_{s_1.e}(i_1.e, t_1.\mathbf{B}) \cong m_{s_2.e}(i_2.e, t_1.\mathbf{B})$. Let us take a closer look at these factors:

- $m_{s_1.e}(i_1.e, t_1.\mathbf{B})$ was computed by multiplying $f_{s_1}(s_1.e)$ with $f_{i_1}(i_1.e|s_1.e, t_1.\mathbf{B})$ followed by eliminating $s_1.e$,
- $m_{s_2.e}(i_2.e, t_1.\mathbf{B})$ was computed by multiplying $f_{s_2}(s_2.e)$ with $f_{i_2}(i_2.e|s_2.e, t_1.\mathbf{B})$ followed by eliminating $s_2.e$.

Now, $f_{s_1}(s_1.e)$ and $f_{s_2}(s_2.e)$, and $f_{i_1}(i_1.e|s_1.e, t_1.\mathbf{B})$ and $f_{i_2}(i_2.e|s_2.e, t_1.\mathbf{B})$ were themselves pairs of shared factors thus it is not surprising that a pair of factors derived from these also turn out to be shared.

5 Inference with shared factors

Shared factors provide opportunity to save computation. While running an inference algorithm, if we generate a pair of factors that share the same function component then that indicates that we repeated the same multiplication and summation operations. We need to recognize and take advantage of such symmetry *before we actually compute shared factors* during inference so that we avoid the repeated steps that go into generating them. Ideally, we would like to generate each shared factor once and reuse the result subsequently during inference. In this section, we develop an approach to do so. We assume that we are given a random variable X whose marginal probabilities need to be computed from a PGM $\mathscr{P} = \langle \mathscr{F}, \mathscr{X} \rangle$ constructed by running a query on a database. We also assume that every $f \in \mathscr{F}$ is associated with an id denoted by $\text{id}(f)$ such that $\text{id}(f_1) = \text{id}(f_2) \Leftrightarrow f_1 \cong f_2, \forall f_1, f_2$.

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 [47] and junction tree algorithm [26]).

For brevity, we will simplify the running example; instead of running $\prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$, we will only consider the join query $S \bowtie_{\mathbf{B}} T$. This suffices for demonstrating our proposed inference algorithm. We now only need to compute marginal probabilities for i_1 and i_2 which are the results of the join query. Figure 7 shows how VE would proceed to solve the two marginal probability computations in detail.

5.1 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 as input a random variable Y and a collection of factors $\mathbf{F}$, each of which involves Y as an argument; it then multiplies all factors in $\mathbf{F}$, and sums Y out from the product to generate a new factor. We denote the resulting (intermediate) factor by m_Y followed by its list of arguments, if they are not clear from the context. For instance, when

$$\begin{aligned}\mu_{i_1}(i_1.e) &= \sum_{s_1.e,t_1.\mathbf{B}} f_{t_1}(t_1.\mathbf{B}) f_{s_1}(s_1.e) f_{i_1}(i_1.e|s_1.e,t_1.\mathbf{B}) \\ &= \sum_{t_1.\mathbf{B}} f_{t_1}(t_1.\mathbf{B}) \underbrace{\sum_{s_1.e} f_{s_1}(s_1.e) f_{i_1}(i_1.e|s_1.e,t_1.\mathbf{B})}_{m_{s_1.e}(i_1.e,t_1.\mathbf{B})} \\ \mu_{i_2}(i_2.e) &= \sum_{s_2.e,t_1.\mathbf{B}} f_{t_1}(t_1.\mathbf{B}) f_{s_2}(s_2.e) f_{i_2}(i_2.e|s_2.e,t_1.\mathbf{B}) \\ &= \sum_{t_1.\mathbf{B}} f_{t_1}(t_1.\mathbf{B}) \underbrace{\sum_{s_2.e} f_{s_2}(s_2.e) f_{i_2}(i_2.e|s_2.e,t_1.\mathbf{B})}_{m_{s_2.e}(i_2.e,t_1.\mathbf{B})}\end{aligned}$$

Fig. 7 Variable elimination on the simplified running example

we were computing $\mu_{i_2}(i_2.e)$ for the example in Fig. 7, we first had to multiply the collection of factors $\{f_{s_2.e}(s_2.e), f_{i_2}(i_2.e|s_2.e, t_1.\mathbf{B})\}$, and then sum out $s_2.e$ from the product to generate the new intermediate factor $m_{s_2.e}(i_2.e, t_1.\mathbf{B})$. Note that $\mathbf{F}$ may contain intermediate factors produced by earlier applications of elimrv.

We first note a few properties of the elimrv operator. The order in which the factors appear in $\mathbf{F}$ may be important. For instance, suppose we want to sum over X_2 from the collection: $\{f_a(X_1, X_2), f_b(X_2, X_3)\}$. Then we would generate 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. Although f_d and f'_d are *numerically equivalent*, they are not *symbolically equivalent*; as we will see later, our algorithm looks for symbolic equivalence which makes this a crucial point. 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. To clarify these points about elimrv and to control its behavior, we feed the operator an explicit label that specifies the above described information.

Example 4 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. For each *elimrv* operation, we go through the list of factors in $\mathbf{F}$ assigning each argument a unique id if it has not been seen before. Then we construct the label by traversing the list of factors again, writing the id of the argument that appears, enclosing the lists of arguments in square braces and finally, appending the label 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 5.1 The *elimrv*$(Y, \mathbf{F}, l)$ operator takes a random variable Y, an ordered list of factors $\mathbf{F}$ 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 $\mathscr{F}$ 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. In this case, the labels do not affect the operations performed, and can be chosen arbitrarily.

5.2 The RV- ELIM graph

For the purposes of introducing our graph-based data structure, we will assume that we are given, besides X and $\mathscr{P} = \langle\mathscr{F}, \mathscr{X}\rangle$, an elimination order $\mathscr{O}$ that contains all random variables involved in $\mathscr{X}$ except for X. In the next section (Sect. 6), we discuss in detail how to construct such an elimination order that suits our purposes. The *rv-elim* graph essentially encodes the sequence of elimrv operations encountered during the run of inference using a labeled graph.

Definition 5.2 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 $\mathscr{P} = \langle\mathscr{F}, \mathscr{X}\rangle$ according to elimination order $\mathscr{O}$ such that:

- Every $v \in V$ represents a factor. If v is a source vertex, then it represents a factor from $\mathscr{F}$ and $l(v) = \mathrm{id}(f)$; if v is not a source vertex 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 $\mathbf{F}$, 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 edge label.

Figure 8a shows the rv-elim graph for our running example using the same elimination order $\mathscr{O} = \{s_1.e, s_2.e, t_1.\mathbf{B}\}$. 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 DAG).

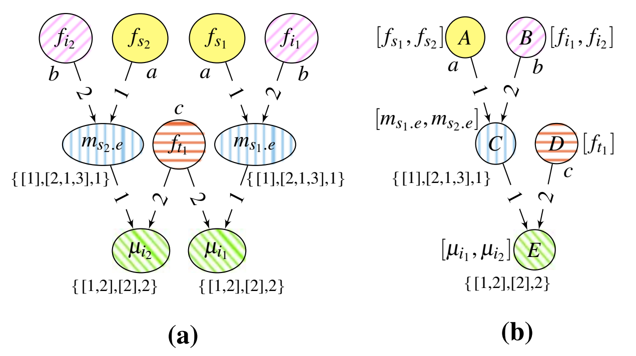


Fig. 8 **a** rv-elim graph for the example from Fig. 7, **b** its compressed version obtained using bisimulation. The rv-elim graph shown in **a** is a vertex-labeled, edge-labeled graph. The edges are labeled with integers (in this case, 1 or 2) and denote the order in which the parent factors are present in the elimrv operation. The vertices are labeled with strings and these are shown alongside the vertex, if the vertex is a source vertex then the label is a letter (e.g., *a* for the first source vertex 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 sink vertex in the rv-elim graph). The compressed rv-elim graph shown in **b** 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 five vertices whereas the rv-elim graph itself contains nine vertices, a significant reduction considering we have such a small running example

5.3 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 5.3 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:

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

Note that for a given internal vertex in the rv-elim graph all incoming edges from parents are assigned distinct edge labels; this is because 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 5.2).

We can now use Claim 5.3 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 Fig. 7 we showed that during the run of inference for our running example, $m_{s_1.e}$ and $m_{s_2.e}$ were intermediate factors that turned out to be shared (shown with vertical shading in Fig. 8a). By looking at the rv-elim graph (Fig. 8a) this is now easy to see since:

– They have the same vertex label {[1], [2, 1, 3], 1}.
– Both $m_{s_1.e}$ and $m_{s_2.e}$ have parents f_{s_1} and f_{s_2}, resp., via edges labeled 1, and $f_{s_1} \cong f_{s_2}$ since they have the same vertex label (viz., a) and are source vertices.
– Both $m_{s_1.e}$ and $m_{s_2.e}$ have parents f_{i_1} and f_{i_2}, resp., via edges labeled 2, and $f_{i_1} \cong f_{i_2}$ since they have the same vertex label (viz., b) and are also source vertices.

Thus by Claim 5.3, $m_{s_1.e} \cong m_{s_2.e}$.

Given a graph (like the rv-elim graph shown in Fig. 8a) and a property (such as the one specified in Claim 5.3), there exist 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* [29]. Given the special case of the graph being a DAG, there exist algorithms that run in time linear in the size of the graph.

Dovier et al. [18], 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 in 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 8b shows the compressed rv-elim graph returned as a result of running bisimulation on the rv-elim graph shown in Fig. 8a. Notice how vertex A represents both factors f_{s_1} and f_{s_2}. We show this in Fig. 8b using shadings and by indicating A's extent in square braces next to it. More interestingly, the pair of intermediate shared factors that we identified earlier have also been collapsed into one single vertex in the compressed rv-elim graph: C represents $m_{s_1.e}$ and $m_{s_2.e}$. Finally, for our example, it turns out that the final marginal

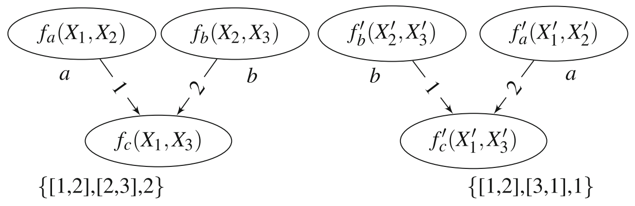


Fig. 9 A poor ordering of parent vertices

probability distributions for i_1 and i_2 are also identical and these have also been collapsed to vertex E in Fig. 8b.

Unfortunately, we cannot apply the bisimulation algorithm described in Dovier et al. directly to our problem. Even though factor multiplication is a commutative operation and we usually have a number of choices regarding the order in which to multiply the parent factors during an elimrv operation, the particular order we choose affects the results. Traditional exact inference algorithms simply choose an order for multiplying the factors arbitrarily. However, in our case, Claim 5.3 actually uses the order of the parents of the vertices in the rv-elim graph to determine which ones represent shared factors. Figure 9 illustrates what could happen if we ordered the parents in the rv-elim graph injudiciously. In Fig. 9, $f_a \cong f_a'$ and $f_b \cong f_b'$, thus if we compute $f_c(X_1, X_3) = \sum_{X_2} f_a(X_1, X_2) f_b(X_2, X_3)$ and $f_c'(X_1', X_3') = \sum_{X_2'} f_a'(X_1', X_2') f_b'(X_2', X_3')$ then f_c and f_c' turn out to be shared and we can detect this by using Claim 5.3. However, if we instead chose to order the elimrv operation differently, i.e., multiply f_b' first and f_a' second as shown in Fig. 9, then the results turn out to have the arguments ordered differently, with possibly different input-output mappings, meaning we have just lost 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 algorithm, and then the heuristic we developed for ordering parents.

5.4 Bisimulation for RV- ELIM graphs

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

Algorithm 1: Bisimulation for rv-elim graphs

```
input: rv-elim graph $G = (V, E)$ with source vertices labeled
$rank(v) = \begin{cases} 0 & \text{if } v \text{ is a source} \\ 1 + \max\{rank(v') | v' \to v \in E\} & \text{o.w.} \end{cases}$
$\rho \leftarrow \max\{rank(v) | v \in V\}$
$B_{0,l} = \{v \in V | v \text{ is a source } \wedge l(v) = l\}$ $\forall l$ labels on sources 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 $C$
```

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). Then the algorithm starts by assigning the source vertices 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 5.3, the partitioning of a vertex only depends on its label and its parents' partitioning. The nested for loops achieve this. They take all vertices at rank i, choose orders for each vertices' parents (we will discuss how this is done shortly), form the label and the key based on this ordering, and partition these vertices based on the constructed key. See [18] 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 5.3 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. We illustrate this by revisiting the example in Fig. 9. Before partitioning f_c and f_c', we first order their parents by their block-ids. Assuming we follow the ordering $a < b$, we would order f_a followed by f_b and f_a' followed by f_b'. The labels $l(f_c)$ and $l(f_c')$ would then turn out to be {[1,2],[2,3],2}, in both cases. Finally, we form the keys on f_c and f_c' using their labels and parents' block-ids, and in this case, concatenating the jth parent's block-id (j, b_j) along with the vertex label gives us

(1, a), (2, b), {[1, 2], [2, 3], 2} in both cases thus allowing us to assign f_c and f_c' to the same block of the partition.

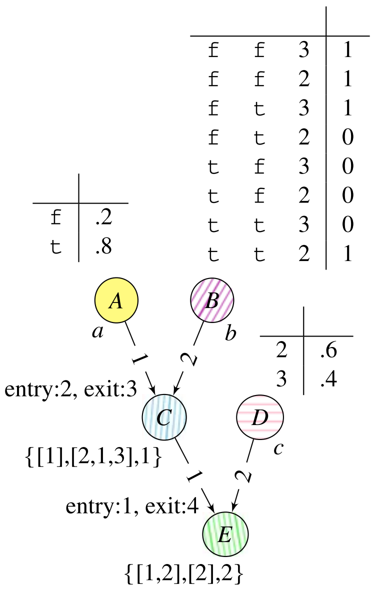
Algorithm 1, by itself, is reasonably efficient. Its time complexity, assuming we use the heuristic that orders based on block-ids, is $O(|V| + |E|)$ (to compute ranks in step 1) $+ \sum_{v\in V} d_v \log d_v + d_v$ (to order the parents and form the key) where d_v is the in-degree of v (ignoring the 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.

5.5 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 Sect. 5.3 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 keys constructed by Algorithm 1 which contain the vertex and edge labels, and all vertices within a 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 source vertices of the rv-elim graph to the source vertices in the compressed rv-elim graph. Then we call a depth-first search procedure (dfs) from the sink vertex 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 parents' functions have not 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 sink vertex of the compressed rv-elim graph. Figure 10 shows the inference procedure for the running example.

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; in that case the compressed rv-elim graph may have multiple sink vertices. Another extension is to use it to compute maximum-a-posteriori (MAP) assignments instead of marginal probabilities, by switching from the sum-product elimrv operator to the max-product operator.



$$C : \text{elimrv}\left(-, \left\{ \begin{array}{c|c} - & \\ \hline \mathtt{f} & .2 \\ \mathtt{t} & .8 \end{array} \begin{array}{ccc|c} & - & & \\ \hline \mathtt{f} & \mathtt{f} & 3 & 1 \\ \mathtt{f} & \mathtt{f} & 2 & 1 \\ \mathtt{f} & \mathtt{t} & 3 & 1 \\ \mathtt{f} & \mathtt{t} & 2 & 0 \\ \mathtt{t} & \mathtt{f} & 3 & 0 \\ \mathtt{t} & \mathtt{f} & 2 & 0 \\ \mathtt{t} & \mathtt{t} & 3 & 0 \\ \mathtt{t} & \mathtt{t} & 2 & 1 \end{array} \right\}, \{[1],[2,1,3],1\} \right) = \begin{array}{cc|c} & & \\ \hline \mathtt{f} & 3 & 1 \\ \mathtt{f} & 2 & .2 \\ \mathtt{t} & 3 & 0 \\ \mathtt{t} & 2 & .8 \end{array}$$

$$E : \quad \text{elimrv}\left(-, \left\{ \begin{array}{cc|c} & - & \\ \hline \mathtt{f} & 3 & 1 \\ \mathtt{f} & 2 & .2 \\ \mathtt{t} & 3 & 0 \\ \mathtt{t} & 2 & .8 \end{array}, \begin{array}{c|c} - & \\ \hline 2 & .6 \\ 3 & .4 \end{array} \right\}, \{[1,2],[2],2\} \right) = \begin{array}{c|c} & \\ \hline \mathtt{f} & .52 \\ \mathtt{t} & .48 \end{array}$$

Fig. 10 Inference on the compressed rv-elim graph. Depicted against each internal vertex is the entry and exit times for the dfs procedure. At the *bottom*, we have shown the elimrv operations that need to be computed where '−' depicts the argument being summed over. `t` `true`, `f` `false`

6 Elimination order generation and optimizations

One of the important steps in performing probabilistic inference is to choose a good elimination order. The best elimination order is defined to be the one that runs inference by minimizing the size of the largest factor (in terms of the number of arguments) encountered during inference over all elimination orders. 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. Finding the best elimination order is known to be NP-Hard [2], even without considering how to optimize inference with shared factors. Thus, as is often done while performing exact inference, to compute practical elimination orders we resort to the use of heuristics. In our case where we are interested in exploiting shared factors, we note the following:

```
Algorithm 2: Minimum size heuristic
  input: PGM $\langle \mathscr{F}, \mathscr{X} \rangle$, query random variables $\mathbf{X} \subseteq \mathscr{X}$
  $\mathscr{O} \leftarrow$ empty list
  // construct adjacency lists
  $\text{Adj}(X) = \{X' | \exists f \in \mathscr{F}$ s.t. $X, X' \in \arg(f)\}, \forall X \in \mathscr{X}$
  while $\exists Y \in \mathscr{X}$ s.t. $Y \notin \mathbf{X}$ do
      // pick random variable with smallest
         neighborhood
      $Y \leftarrow \text{argmin}_{Y \in \mathscr{X}, Y \notin \mathbf{X}} |\text{Adj}(X)|$
      add $Y$ to $\mathscr{O}$
      // update Adj
      for $X \in Adj(Y)$ do
          $\text{Adj}(X) \leftarrow \text{Adj}(X) \setminus \{Y\}$
      for $X, X' \in Adj(Y)$ do
          $\text{Adj}(X) \leftarrow \text{Adj}(X) \cup \{X'\}$
          $\text{Adj}(X') \leftarrow \text{Adj}(X') \cup \{X\}$
      delete $\text{Adj}(Y)$
      delete $Y$ from $\mathscr{X}$
  return $\mathscr{O}$
```

1. On one hand, the elimination order defines the rv-elim graph.
2. On the other hand, given a set of factors, ideally, we would like to eliminate arguments of shared factors earlier since such an operation helps save computation. This, in turn, means that the rv-elim graph affects the best choice of random variables to be eliminated next, since we determine shared factors through the rv-elim graph.

This tight coupling of the elimination order and the rv-elim graph motivates the use of a unified approach that does both, determines the sharing among factors and chooses which random variables to eliminate next. In this section, we describe such an approach. But first we review a popular elimination order determination technique referred to as the minimum size heuristic (MSH) [30] that is often used to construct elimination orders for traditional exact inference algorithms.

6.1 Minimum size heuristic

Traditional MSH is a greedy heuristic that returns a list of the random variables that need to be eliminated for inference. The main goal of MSH is to pick the next random variable such that the intermediate factor produced by eliminating it produces the smallest factor (in terms of number of joint assignments to its arguments). The manner in which MSH operates is easily described by visualizing its operations on the *moral graph.*

Definition 6.1 Given a collection of factors $\mathbf{F}$, the *moral graph* is defined to be an undirected graph where vertices correspond to random variables and two random variables X_1 and X_2 are connected by an edge if and only if there exists $f \in \mathbf{F}$ which involves both X_1, X_2 as arguments.

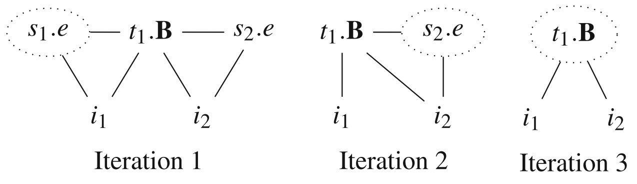


Fig. 11 Minimum size heuristic run on our running example where we would like to compute marginal probabilities for i_1 and i_2. The edges between $s_1.e$ and $t_1.\mathbf{B}$, and between $s_2.e$ and $t_1.\mathbf{B}$ are due to factors $f_{i_1}(i_1.e|s_1.e, t_1.\mathbf{B})$ and $f_{i_2}(i_2.e|s_2.e, t_1.\mathbf{B})$, respectively. Note that in the first iteration we could have also chosen $s_2.e$. In each iteration, the *dotted node* represents the random variable chosen for elimination. The resulting elimination order is: $\{s_1.e, s_2.e, t_1.\mathbf{B}\}$

Consider eliminating X from $\mathbf{F}$. Let $\{f_1(\mathbf{X}_1), \ldots f_m(\mathbf{X}_m)\} \subseteq \mathbf{F}$ denote the set of factors that involve X as an argument. Then, after eliminating X, we will generate a new factor m_X whose argument set is $\mathbf{X}_1 \cup \cdots \cup \mathbf{X}_m \setminus \{X\}$. In terms of the moral graph, the arguments of the new factor m_X are exactly the neighbors of X. Thus we can now implement MSH iteratively. In each iteration, we simply pick the random variable with the smallest neighborhood, update the moral graph by adding edges to represent the new factor m_X introduced by eliminating X, delete X from the moral graph and proceed until we have eliminated all random variables other than the ones whose marginals we are interested in. MSH can be made to run efficiently by maintaining the moral graph in adjacency list format. Algorithm 2 shows the complete algorithm where $\mathbf{X}$ denotes the set of random variables whose marginals need to be computed. Figure 11 shows what happens when we run MSH on our running example where we want to compute the marginal probabilities of i_1 and i_2. In the first iteration, $s_1.e$ is chosen since this has the smallest neighborhood (note that we could have also chosen $s_2.e$), in the second iteration $s_2.e$ is chosen and finally, $t_1.\mathbf{B}$ is chosen providing the final elimination order.

What we just described is the simplest version of MSH. More advanced versions of MSH compute the size of a factor by counting the number of rows in it (product of the domains of the factors' arguments). We refer the interested reader to [26] for more details.

6.2 Augmented rv-elim graph: generation and coloring

Unfortunately, applying MSH naively is unlikely to work well in our case. As we indicated earlier, ideally, we would like to construct elimination orders that lead to rv-elim graphs with symmetry and, in turn, eliminate random variables from shared factors early on while keeping the sizes of the factors in check. To this end, we will now describe an augmented version of the rv-elim graph that not only helps determine shared factors but also helps determine the elimination order that preserves and exploits sharing. Once we have introduced our new data structure then we will describe how to perform inference on it.

Definition 6.2 The *augmented rv-elim graph* $G = (V_f, V_{rv}, E_{f \to f}, E_{rv \to f}, L_v, L_e)$ is an edge-labeled and vertex-labeled graph such that:

- Every $v \in V_f$ represents a factor.
- Every $v \in V_{rv}$ represents a random variable.
- Every $(v_1 \xrightarrow{i} v_2) \in E_{f \to f}$ such that $v_1, v_2 \in V_f$ denotes that the factor represented by v_1 was the i^{th} factor multiplied to produce the factor represented by v_2 by the corresponding elimrv operation.
- Every $(v_1 \xrightarrow{i} v_2) \in E_{rv \to f}$ such that $v_1 \in V_{rv}$ and $v_2 \in V_f$ denotes that the random variable represented by v_1 is the i^{th} argument of the factor represented by v_2.
- $L_v(v), \forall v \in V_f$ denotes the label such that:
  - If v is not the head of any edge in $E_{f \to f}$ then $L_v(v) = id(f)$ where f is the factor represented by v.
  - If v is the head of some edge in $E_{f \to f}$ then $L_v(v) = l$ where l is the label of the corresponding elimrv operation that produced the factor corresponding to v.

Essentially, the augmented rv-elim graph, as the name suggests, is the rv-elim graph with extra vertices representing random variables added to it. These random variable vertices (or rv-vertices) are connected via edges in $E_{rv \to f}$ to the factors where they appear as arguments and the label on the edge is simply the position of the argument in the factor. Figure 12 shows the complete aug. rv-elim graph for our running example where the edges in $E_{rv \to f}$ are denoted via dotted lines to differentiate them from edges in $E_{f \to f}$, the original edges in the rv-elim graph defined in the previous section. Thus,

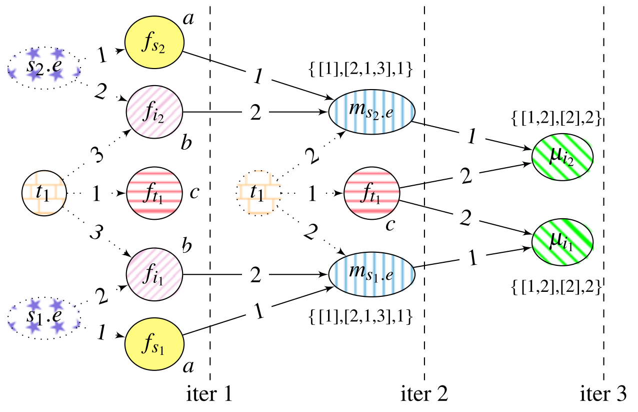


Fig. 12 Augmented rv-elim graph for our running example and how the coloring/generation algorithm works in this case. The *dotted rv-vertices* depict which random variables are picked for elimination in each iteration. The edges in $E_{rv \to f}$ are denoted by *dotted lines*. Note that we have two vertices representing $t_1.\mathbf{B}$ purely to maintain legibility

for example, $s_2.e$ connects to f_{i_2} via a dotted edge labeled 2 since $s_2.e$ is the second argument in $f_{i_2}(i_2.e|s_2.e, t_1.\mathbf{B})$.

The advantage of using the aug. rv-elim graph to track sharing is that not only does it help identify shared factors, but it also helps identify *random variables appearing as arguments in the same positions in shared factors* (which we will refer to as shared random variables). In a moment we will see why identifying shared random variables is important to determine the elimination order, but first let us see how shared random variables are identified. Going back to Fig. 12, recall that we can easily identify shared pairs of factors f_{s_2} and f_{s_1} (filled solid), and f_{i_1} and f_{i_2} (north-east stripes) using Claim 5.3 (for the aug. rv-elim graph Claim 5.3 does not change except that the edges used to find shared factors are only edges from $E_{f \to f}$). Having found these shared factors, we can now define a different property that helps determine shared random variables:

Property 6.3 *For augmented rv-elim graph $G = (V_f, V_{rv}, E_{f \to f}, E_{rv \to f}, L_v, L_e)$, two vertices $v_1, v_2 \in V_{rv}$ represent shared random variables if:*

- *if v_1 has k i-labeled edges then v_2 has k i-labeled edges*
- $\forall (v_1 \xrightarrow{i} u_1), (v_2 \xrightarrow{i} u_2) \in E_{rv \to f} : u_1 \cong u_2$

The property basically says that two rv-vertices are shared if they appear as arguments in the same positions to shared factors. Using this property it is now easy to see that in Fig. 12, rv-vertices $s_1.e$ and $s_2.e$ represent shared random variables since they both appear as first arguments to f_{s_1} and f_{s_2} (which are shared factors) and, as second arguments to f_{i_1} and f_{i_2} (shared factors). This implies that if we were to now construct the compressed rv-elim graph, then both f_{s_1} and f_{s_2}, and f_{i_1} and f_{i_2} would collapse to the same vertices, and eliminating $s_1.e$ and $s_2.e$ would be possible in one single operation, which would lead to savings since we would eliminate 2 random variables in one single operation. By using Property 6.3 on the aug. rv-elim graph we have now managed to identify this advantageous operation and we can now proceed by eliminating $s_1.e$ and $s_2.e$. The other option would have been to eliminate $t_1.\mathbf{B}$, but this would not lead to much savings in computation since this random variable is not shared with any other random variable.

The basic strategy behind coloring (and constructing) the aug. rv-elim graph is as follows:

- We first begin by collecting all factors from $\mathscr{F}$ and forming the first layer of the aug. rv-elim graph. Note that since we do not know which (shared) random variables to eliminate we do not know what the intermediate factors generated during inference will be. Going back to Fig. 12, this means all we have at this stage is the graph until the first dashed vertical line.

- Then we invoke Claim 5.3 to determine shared factors.
- We then invoke Property 6.3 to determine shared random variables.
- Now we invoke MSH. We first collect all random variables with the minimum sized neighborhood in the moral graph and then we organize them by their colors (assigned in the previous step by invoking Property 6.3). We choose the random variables of the color that has the greatest extent size.
- We then proceed by eliminating the chosen random variables, generating the next set of intermediate factors, generating the next layer of the aug. rv-elim graph and repeating the whole process.

Figure 12 shows the whole process for our running example. In the first iteration, we eliminate $s_1.e$ and $s_2.e$ to generate the first set of intermediate factors ($m_{s_1.e}$ and $m_{s_2.e}$). In the second iteration, we have no option but to eliminate $t_1.\mathbf{B}$ thus producing the next layer (μ_{i_1} and μ_{i_2}). The whole process ends in 3 iterations. Algorithm 3 describes the full algorithm in detail.

Having generated and colored the aug. rv-elim graph, we can simply extract the subgraph corresponding to the factors comprising of V_f and $E_{f\to f}$ and generate the compressed rv-elim graph based on their colors. Then we can perform inference on the compressed rv-elim graph just as we described in Sect. 5.

6.3 Further optimizations

Query evaluation in probabilistic databases, in general, allow for a number of optimizations. One such optimization that we describe here is possible because very often during query evaluation, we are interested only in marginal probabilities of individual random variables separately. For instance, in our running example, we are interested in computing the marginal probabilities $\mu_{i_1}(i_1.e)$ and $\mu_{i_2}(i_2.e)$, not $\mu(i_1.e, i_2.e)$. This is true in most other works in probabilistic databases including [11,13,21,31] (the last reference considers computing single random variable marginal probabilities conditioned on some evidence). In such a case, one simple optimization would be to make sure that at no point during inference do we generate a factor with more than one query random variable as its argument. Even though this does not change the inherent hardness of the general problem of query evaluation for probabilistic databases, it should, nevertheless, help keep the sizes of factors small.

To see how this relates to the running example, let us go back to Fig. 12. Recall that, in the second iteration, we need to eliminate $t_1.\mathbf{B}$ from the set of factors $f_{t_1}(t_1.\mathbf{B})$, $m_{s_1.e}(i_1.e, t_1.\mathbf{B})$ and $m_{s_2.e}(i_2.e, t_1.\mathbf{B})$. If performed naively, this would give rise to a single factor with both $i_1.e$ and $i_2.e$ as arguments; but this is not necessary for our purposes of computing the marginal probabilities for $i_1.e$ and $i_2.e$ (even though it is possible to retrieve the required marginal probabilities from that factor). Instead it would be more convenient if we simply generate two factors, one containing the marginals for $i_1.e$ and the

Algorithm 3: Generate & Color Aug. RV-Elim Graphs

```
input: PGM $\langle \mathscr{F}, \mathscr{X} \rangle$, query random variables $\mathbf{X} \subseteq \mathscr{X}$
Adj$(X) = \{X' | \exists f \in \mathscr{F}$ s.t. $X, X' \in \arg(f)\}, \forall X \in \mathscr{X}$
$V_{rv} \leftarrow \mathscr{X}$; $V_f \leftarrow \mathscr{F}$; $E_{f\to f} \leftarrow \emptyset$
$E_{rv\to f} \leftarrow \{(v_r \xrightarrow{i} v_f) |$s.t. $r$ is $i^{th}$ argument of $f\}$
$L_v(v) \leftarrow$ id$(f), \forall v \in V_f$ where $f$ is the factor denoted by $v$
$L_e(v_r \xrightarrow{i} v_f) \leftarrow i, \forall (v_r \xrightarrow{i} v_f) \in E_{rv\to f}$
$G \leftarrow (V_{rv}, V_f, E_{f\to f}, E_{rv\to f}, L_v, L_e)$
$B^f_{0,l} \leftarrow \{v \in V_f | L_v(v) = l\}, \forall l$ labels in $G$
$C^f \leftarrow \{B^f_{0,l}\}$
while $\exists Y \in \mathscr{X}$ s.t. $Y \notin \mathbf{X}$ do
    for $v \in V_{rv}$ do
        $k^{rv}_v \leftarrow \{(i, B_f) | (v \xrightarrow{i} v_f) \in E_{rv\to f}\}$ where $B_f$ denotes
        $v_f$'s color
    // color rv-vertices
    $B^{rv}_k \leftarrow \{v \in V_{rv} | k^{rv}_v = k\}, \forall k$
    /* pick blocks of rv-vertices with the
       smallest neighborhood                        */
    $C^{rv} \leftarrow \{\text{argmin}_{B^{rv}_k, \exists Y \in B^{rv}_k \wedge Y \notin \mathbf{X}} |Adj(Y)|\}$
    $B^{rv} \leftarrow \text{argmax}_{B \in C^{rv}} |B \setminus \mathbf{X}|$
    $V_{\text{new}} \leftarrow \emptyset$
    for $v \in B^{rv}$ do // generate next layer of $G$
        // collect factors for $v$
        $\mathbf{F}_v \leftarrow \{v_f | (v \xrightarrow{i} v_f) \in E_{rv\to f}\}$
        $o \leftarrow$ order $\mathbf{F}_v$ based on their colors
        construct $L_v(v)$ (see Sect. 5)
        Let $v_{\text{new}}$ denote the new factor produced by
        elimrv$(v, \mathbf{F}_v, L_v(v))$
        $V_{\text{new}} \leftarrow V_{\text{new}} \cup \{v_{\text{new}}\}$
        $V_f \leftarrow V_f \cup \{v_{\text{new}}\}$
        for $v_f \in \mathbf{F}_v$ do
            $E_{f\to f} \leftarrow E_{f\to f} \cup \{(v_f \xrightarrow{i} v_{\text{new}})\}$ where $i$ denotes
            the position of $v_f$ in $o$
        $\mathbf{Y} \leftarrow \cup_{f \in \mathbf{F}_v} \mathbf{Y}_f$ where $\mathbf{Y}_f$ denotes arguments of $f$
        $\mathbf{Y} \leftarrow \mathbf{Y} \setminus v$
        for $Y \in \mathbf{Y}$ do
            $E_{rv\to f} \leftarrow E_{rv\to f} \cup \{(Y \xrightarrow{i} v_{\text{new}})\}$ where $i$ denotes
            the position of argument $Y$ in factor denoted by $v_{\text{new}}$
        // update Adj
        for $Y \in Adj(v)$ do
            Adj$(Y) \leftarrow$ Adj$(Y) \setminus v$
        for $Y, Y' \in Adj(v)$ do
            Adj$(Y) \leftarrow$ Adj$(Y) \cup \{Y'\}$
            Adj$(Y') \leftarrow$ Adj$(Y') \cup \{Y\}$
        delete Adj$(v)$; delete $v$ from $\mathscr{X}$
    // color the constructed vertices
    $B^f_k \leftarrow \{v \in V_{\text{new}} | k^f_v = k\}$
    $C^f \leftarrow C^f \cup \{B^f_k\}_{\forall k}$
return $C^f$
```

```
Algorithm 4: Eliminate a random variable
    input: F, X to be eliminated, X query random variables, graph
           $G = (V, E)$ whose vertices denote random variables and
           edges denote correlations
    rv-tags($Y$) $\leftarrow \begin{cases} \{Y\} & \text{if } Y \in \mathbf{X} \\ \cup_{Y' \text{ s.t. } (Y \to Y) \in E} \text{rv-tags}(Y') & \text{o.w.} \end{cases}$
    factor-tags($f$) $\leftarrow \bigcap_{Y \in \arg(f)} \text{rv-tags}(Y)$
    for $Y$ s.t. $\exists f \in \mathbf{F} \wedge Y \in$ factor-tags($f$) do
        factors($Y$) $\leftarrow \{f \in \mathbf{F} | Y \in \text{factor-tags}(f)\}$
    /* partition query random variables based
       on the factors they appear as tags for
       */
    $B_{\text{factors}} \leftarrow \{Y | \text{factors}(Y) = \text{factors}\}$
    $C \leftarrow \{B_{\text{factors}}\}_{\forall \text{factors}}$
    $\mathbf{F}_{\text{new}} \leftarrow \{\sum_X \prod_{f \in B} f\}_{B \in C}$
    return $\mathbf{F}_{\text{new}}$
```

rv	rv-tags
$s_1.e$	$\{i_1.e\}$
$s_2.e$	$\{i_2.e\}$
$t_1.\mathbf{B}$	$\{i_1.e, i_2.e\}$
$i_1.e$	$\{i_1.e\}$
$i_2.e$	$\{i_2.e\}$

factor	factor-tags
$f_{t_1}(t_1.\mathbf{B})$	$\{i_1.e, i_2.e\}$
$m_{s_1.e}(i_1.e, t_1.\mathbf{B})$	$\{i_1.e\}$
$m_{s_2.e}(i_2.e, t_1.\mathbf{B})$	$\{i_2.e\}$

tags	factor set	factor produced
$\{i_1.e\}$	$\{m_{s_1.e}(i_1.e, t_1.\mathbf{B}), f_{t_1}(t_1.\mathbf{B})\}$	$\mu_{i_1}(i_1.e)$
$\{i_2.e\}$	$\{m_{s_2.e}(i_2.e, t_1.\mathbf{B}), f_{t_1}(t_1.\mathbf{B})\}$	$\mu_{i_2}(i_2.e)$

Fig. 13 Optimizing random variable elimination for single random variable marginal probability computation. Applying Algorithm 4 to our running example while trying to eliminate $t_1.\mathbf{B}$ from the set of factors $f_{t_1}(t_1.\mathbf{B})$, $m_{s_1.e}(i_1.e, t_1.\mathbf{B})$ and $m_{s_2.e}(i_2.e, t_1.\mathbf{B})$

other for $i_2.e$ (which is what we have shown in Fig. 12). To achieve this, we first need to *tag* each random variable with the query random variables for whose marginal probability computation they are required. Let G denote the graphical representation of the PGM where vertices denote random variables and edges denote correlations (see Fig. 5c for an example). Now define tags(Y) $= \{X \in \mathbf{X} | Y \rightsquigarrow X \text{ in } G\}$, where $\mathbf{X}$ denotes the query random variables whose marginals we require and $Y \rightsquigarrow X$ denotes that there exists a path from Y to X. The tags can be computed by running a simple depth-first search on G which begins at the source vertices and collects tags from the children until we reach the query random variables which appear as sink vertices and whose tags are themselves. Having computed tags for random variables, we will also need tags for factors: tags(f) $= \bigcap_{X \in \arg(f)}$ tags(X). At this point, we could simply collect all factors f that contain a particular query random variable X in its tags and perform one elimination operation over the collected factors. One such operation for each query random variable should achieve our goal. But this may lead to repeated computation when there is more than one query random variable present in the tags for the same set of factors. To avoid such a situation, we first partition the query random variables that appear in the factor tags into sets such that two query random variables that appear in tags of the same set of factors are partitioned together. After this, for each partition, we collect the corresponding factors and perform one elimination operation each. Algorithm 4 provides the full procedure. Note that instead of recomputing rv-tags and factor-tags separately for each elimination operation as suggested in Algorithm 4, it may be advantageous to maintain these data structures globally to save computation.

Figure 13 shows how applying the procedure before eliminating $t_1.\mathbf{B}$ helps produce two factors $\mu_{i_1}(i_1.e)$ and $\mu_{i_2}(i_2.e)$ as opposed to constructing one single factor with both query random variables as arguments.

7 Experimental evaluation

Our experiments were designed to answer the following questions:

- Does exploiting shared factors help achieve faster query evaluation?
- When is it worthwhile to apply our bisimulation-based approach to a query evaluation problem?
- What factors of speedup can we hope to gain by exploiting shared factors?
- Does evaluating more complex queries, as opposed to simple queries used in the previous sections, also lead to shared factors?
- Is our approach equally effective on databases with both attribute and tuple uncertainty?

We compared approaches on two different scenarios; one based on our earlier car example, and another based on the TPC-H benchmark.

Approaches We compare three inference techniques and report their query evaluation times. For each approach, we report wall clock times divided into two parts: time spent to perform arithmetic operations (**arith. ops.**), which includes the time spent to multiply factors and sum over random variables; and time spent to perform all other remaining operations (**rem.**), which includes time spent to find the elimination order, time spent running bisimulation(s), if the inference technique requires any, etc. The three inference approaches we compare are:

- **BatchVE** variable elimination [47], a standard inference algorithm, modified so that we can compute the marginal probabilities of all random variables of interest in one pass. This approach does not exploit shared factors.
- **SharedInf (2P)** an earlier approach we developed [42] where we used a two phase approach for exploiting shared

factors. In this approach, we first determine the elimination order by compressing the PGM using bisimulation and by transferring the ids on the factors to their corresponding random variables. This approach has its limitations since it is not straightforward to extend the approach to cases where there is not a one-to-one correspondence between factors and random variables. Moreover, in this approach, we partition the random variables once, as opposed to potentially doing it multiple times as in the new approach we proposed in this article.
- **SharedInf** the approach proposed in this article.

For each experiment we also report the time required to perform the relational algebra operations **rel. alg. ops.**, which includes the time to run the query and to introduce the factors needed to run inference. All the reported parameters are also explained in the legend at the top of Fig. 14.

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

7.1 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 three inference algorithms varies with different characteristics of the data. In addition to the relation containing the advertisements (Ad) described in Fig. 1a, we added another relation which denotes the source websites from which the ads were extracted (S). Each tuple in S is an uncertain tuple with an associated probability of existence which depends on the reliability of the website's information. We also added a **Color** attribute to Ad. For these experiments, we ran the following query: $\prod_{\mathbf{Ad}}((\sigma_{\mathbf{Color=c}} Ad) \bowtie_{\mathbf{SourceID}} S)$ where c denotes a specific color and **SourceID** is a primary key in S and acts as a foreign key in Ad. In addition to the uncertain tuples in S, we set the **Color** attribute values to be uncertain and correlated with the **Model** attributes. A car of a certain **Model** can have one of four distinct **Colors**. The parameters that we varied for these experiments are d (domain size of **Model**,

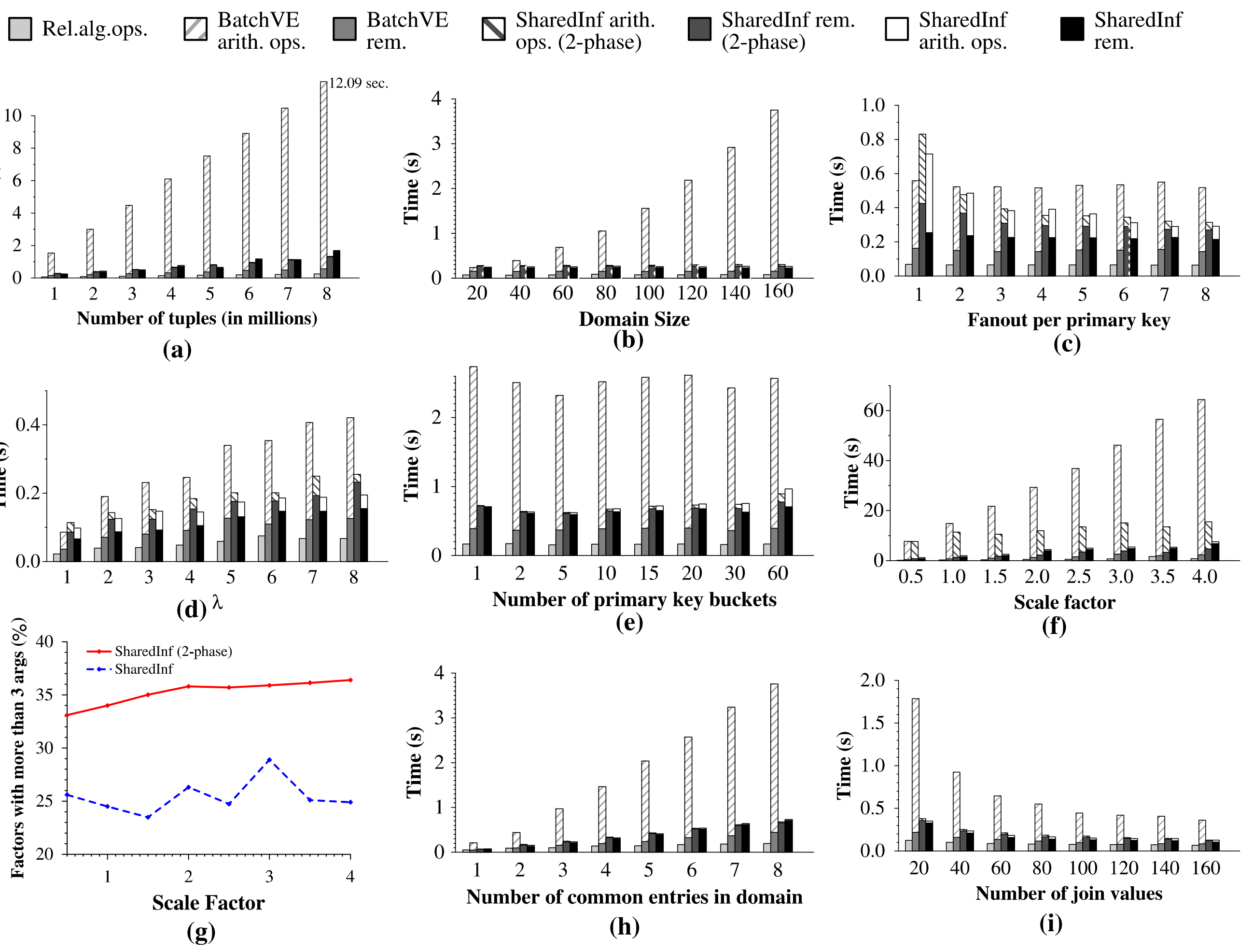


Fig. 14 Plots for experiments on synthetic and TPC-H data. The legend is shown on *top*

default was 50), n (the number of attribute uncertainty tuples in Ad, default value is 100) and fanout (the number of tuples in Ad that each tuple from S joins with, default value is 100).

In Fig. 14a, we show how BatchVE, SharedInf (2P) and SharedInf perform when we vary n from 100 to 800. Notice that both SharedInf and SharedInf (2P) significantly reduce the time spent performing arihtmetic operations (almost invisible in the plot for both). Note that on the x-axis in Fig. 14a, we report the size of Ad in terms of the 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, we use an approach proposed by Dalvi and Suciu [10] to convert data with attribute-level uncertainty to tuple-level uncertainty. This approach computes 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 $n \times d_1 \times d_2 \times \cdots d_{|attr(R)|}$ 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. To express the effects of this transformation (and the explosion in size it leads to) in numbers, we ran the same queries on MystiQ [4], which is a publicly available tuple-level uncertainty probabilistic database. We ran MystiQ with its default settings. In Table 1(a), we compare the times for SharedInf against the times it took to run the same queries on MystiQ. As should be clear, the sheer explosion in number of tuples produced by the transformation causes a blowup in terms of execution times. In this case, since we were experimenting with a foreign-key primary-key join, MystiQ actually found a safe plan, so this represents a best case scenario for MystiQ.

In Fig. 14, in some cases SharedInf required more time to run operations other than arithmetic operations compared to SharedInf (2P). For instance, at $n = 800$ SharedInf rem. is about 1.7 s while SharedInf rem. (2-phase) is 1.33 s. This is to be expected as we discussed earlier. However, it is unclear if this extended preprocessing led to any gains in inference since all the arith. ops. times (except BatchVE's) are close to zero. We compare SharedInf and SharedInf (2P) in more detail in some of the experiments in the next section.

Figure 14b shows the performance of the three inference algorithms with varying domain sizes. Notice how at $d = 20$, SharedInf (2P) actually performs worse (because small domain sizes means small factors and therefore, less time spent on arithmetic operations), but the difference between its time and BatchVE's time is not large. On the other hand, SharedInf's total inference time is almost the same as BatchVE's. In Table 1b, we compare MystiQ's execution times against SharedInf's for the same queries.

The third experiment we ran (Fig. 14c) is the most interesting experiment in this subsection. Here we varied the fanout from 1 to 8 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 and SharedInf (2P) perform worse than BatchVE, but not by a huge margin; in fact, SharefInf fares slightly better at this extreme parameter setting. 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 and SharedInf (2P) are already doing better than BatchVE. At fanout 8 they both perform significantly better than BatchVE. Note that most of the SharedInf arith. ops. times in this plot may look larger than SharedInf (2P)'s, which is disconcerting. The differences, however, are not very large in absolute terms, for example, at fanout = 1 the difference between SharedInf arith. ops and SharedInf (2P) arith. ops. is about a 100 milliseconds. We took a closer look at the reason behind this and found that if we simply compare the number of multiplications and summations, then these turn out to be identical. We hypothesize that these longer arith. ops. times for SharedInf could be because of the additional bookkeeping we need to do; SharedInf (2P)'s bookkeeping is not nearly as complex, since for SharedInf we need to keep track of shared random variables and shared factors simultaneously. In the experiments with TPC-H data (reported later), as we increased the size of the relations, the increased scale removes these discrepancies.

In Fig. 14d, instead of using identical fanouts for all tuples in S, we sampled them from a Poisson distribution with parameter λ. 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, ..., i.e., 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 have the same

Table 1 Comparison with a tuple uncertain probabilistic database

#Tuples	SharedInf (s)	MystiQ (s)
1 million	0.332	6.54
8 million	1.956	94.623
Domain size	**SharedInf (s)**	**MystiQ (s)**
20	0.314	1.09
160	0.341	128.429

existence probability. The number of tuples in S was fixed to 60, so at 60 buckets (right end of the plot), we have exactly 1 tuple belonging to each bucket. Figure 14e shows how SharedInf's performance deteriorates when the number of buckets increase.

7.2 Experiments with TPC-H data

We also ran experiments on a database with the TPC-H schema. We picked Q5 from the TPC-H specification since it involves a join among six relations, of which we made 4 relations (customer, lineitem, supplier and order) probabilistic. The query determines the volume of sales 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 randomly. For the lineitem and order relations, we made the discount attribute uncertain (domain size $4d$) and correlated with the type of the part being ordered (domain size d); we also made 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 Fig. 14f. 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.

Figure 14f shows that in this case of a four-relation join on uncertain data, SharedInf performs significantly better than not only BatchVE (which is to be expected) but also when compared to SharedInf (2P). The reason for this speedup is also captured in Fig. 14f: SharedInf saves significantly on arithmetic operations, whereas the time spent on running multiplications and summations in SharedInf (2P) shows up prominently. We investigated the reason behind this and compared the distributions of intermediate factors with respect to their sizes in the case of SharedInf and SharedInf (2P). Figure 14g shows the results. We find that if we just look at factors with more than three arguments then on an average, about 30–35% of the intermediate factors produced by SharedInf (2P) fall in this category whereas as SharedInf only produces such factors about 25% of the time. If we just look at factors with five arguments then the difference is more prominent. About 10% of the intermediate factors generated by SharedInf (2P) have five arguments where SharedInf has only 1%. Since dealing with large factors (especially when their arguments have large domains) is where inference algorithms usually spend the bulk of their time, this clearly shows SharedInf manages to determine elimination orders that produce smaller factors and lead to faster inference than SharedInf (2P).

7.3 Experiments with uncertain join attributes

The next two plots (Fig. 14h, i), show results for a two relation join between S and Ad where the join attribute **SourceID** itself was uncertain. This relates to the structure uncertainty where we are unsure about the primary/foreign key values in the data. For instance, as in Fig. 1b, 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 by generating uncertain attribute values for **SourceID** in both S and Ad in the following fashion. First we constructed k key values, then for each tuple in either relation we sampled 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 14h shows that increasing the value of m (k was held constant at 100), all three algorithms' times increase, although BatchVE has a more pronounced dependency on the value of m. Figure 14i shows how increasing the value of k (m was held constant at 2) helps reduce SharedInf's and SharedInf (2P)'s times more drastically than BatchVE's.

In almost all our experiments, SharedInf achieved significant speedups of up to 15 times compared to BatchVE and 7 times compared to SharedInf (2P). These numbers can easily increase if the random variables have larger domains. If we restrict our attention to time spent in multiplying factors and summing over random variables (arithmetic operations), then the speedups achieved by SharedInf range from 100 times, when compared to BatchVE, to 13 times, when compared to SharedInf (2P). More importantly, SharedInf showed an ability to reduce the number of large factors introduced during inference which means it determines better elimination orders for inference. Both SharedInf and SharedInf (2P) can waste time during query evaluation looking for symmetry when there is not much in the PGM. But in such cases, as our experimental study demonstrates, the time wasted is not exorbitant. SharedInf also performs more preprocessing and book-keeping in comparison to SharedInf (2P) but here also, the excess time seems worth the effort if it reduces the number of large factors introduced during inference.

8 Related work

The work described in this article builds on work from a number of different research areas. In what follows, we review the prior work from the areas most closely related to ours.

8.1 Structured large scale PGMs and lifted inference

Researchers in machine learning and artificial intelligence have spent a considerable amount of time developing models that enable them to express and reason with uncertainty and complex correlations. One of the issues they face when applying these approaches in the real-world is that of scale. Usually, modeling any non-trivial domain requires a large number of random variables resulting in a large model that is difficult to maintain and unwieldy to use. This has led to the development of a fairly long list of structured large-scale graphical models over the past decade, including (but not limited to) probabilistic relational models (PRMs) [20,23] and Markov logic networks [37]. We refer the interested reader to Getoor and Taskar [22] for an introduction to these and various other models. All of these approaches model shared correlations in some way, and developing new methods for exploiting them during inference and learning is an active research area; our bisimulation-based inference algorithm is a contribution to that line of work as well [43].

Among these, PRMs probably resemble PrDB most closely, because they too deal with relational data and shared factors. Most of the prior work on PRMs [20,23] has concentrated on how to specify and learn a schema-level probabilistic model for relational data. However, there are two important differences between large scale graphical models like PRMs and probabilistic databases like PrDB:

– Barring a few proposals such as Problog [14], most structured large scale graphical models restrict the user to ask queries that can be expressed only in terms of random variables and marginal and/or conditional probability distributions. Unlike probabilistic databases, they do not allow the use of a sophisticated, high-level querying language like SQL to enhance the usability of the system and the choice of posing more expressive queries.
– More importantly, even though most of these approaches exploit shared correlations to keep the model compact, during inference they usually ground or *unroll* the whole model and use standard inference algorithms to compute the required marginal/conditional probabilities.

More recently, researchers have started developing techniques for *lifted inference*, which try to exploit shared correlations for efficient inference; that line of work is closely related to our approach here. However, most of the work in lifted inference [15,32,34] assumes that the shared correlations are clearly specified using first-order logic that is subsequently used to speed up inference. In our work, we do not assume the presence of a first-order description. This is because the query evaluation approach for probabilistic databases does not provide such a description, and it is not clear how 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. Our techniques provide an alternative way to do inference over shared factors, and can be directly applied to that problem as well. We note that our approach subsumes a specific kind of lifted inference known as inversion elimination [15,34]. In fact, most of the operations defined in the lifted inference literature in conjunction with inversion elimination can be clearly seen in our framework too, e.g., *parameterized factors* [34] correspond to factors assigned the same color in rv-elim graphs and *parameterized random variables* [34] correspond to random variables assigned the same color in augmented rv-elim graphs.

Singla and Domingos [45] present another lifted inference approach that has superficial similarities to our approach. They propose an algorithm that lifts or compresses the factor graph corresponding to the PGM which consists of both vertices depicting random variables and factors. They then show that the compressed graph can be used to run inference approximately. This approach is similar to compressing our augmented rv-elim graph, however, our aug. rv-elim graph is more detailed because it not only contains vertices that denote factors in the PGM but also intermediate factors that get generated during inference. Because of this, through the use of the aug. rv-elim graph we can perform exact inference, whereas the algorithm by Singla and Domingos can only provide approximate results.

8.2 Probabilistic databases

As discussed in the introduction, much of the recent work on probabilistic databases is based on computing marginal probabilities to answer queries [11,13,21,31,36]. As such, making this part of query evaluation more efficient has been one of the main focuses of this area of research. Das Sarma et al. [13] describe memoization techniques in Trio where every time a tuple's existence probability is computed, Trio caches its result; the cached results are in turn used to compute existence probabilities of other tuples if required. 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 inference. Another distinction is that Trio does not attempt to exploit shared correlations. Other approaches to faster query processing include using traditional index structures [44] and special-purpose index structures [12], but these only help in data retrieval, and not in speeding up the inference process itself. Re et al. [36] present techniques for query processing with approximate inference but with guaranteed ranking. In a recent work, Wang et al. [46] observe that in many scenarios, users may want to query a probabilistic database at various levels at granularity. Query processing at coarse levels of granularity has many similarities to the lifted inference problem, and the authors draw upon the prior work in that area to efficiently process such queries. Our work in this article is orthogonal to their work, and our techniques can be applied to their problem as well.

In a related vein, Bravo and Ramakrishnan [5] suggest representing factors as relations so that we can use 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 three argument and-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 for representing large factors using relations can be combined with our current approach of representing factors as objects.

9 Conclusion

In this article, we presented PrDB, a flexible and efficient probabilistic database model. We showed how query evaluation corresponded to probabilistic inference in an appropriately constructed graphical model. In addition, we showed how to exploit shared correlations to speed up probabilistic inference during query evaluation. We introduced a new graph-based data structure, called rv-elim graph, that compactly captures a run of an inference process, and explained how to build it from the PGM given an elimination order. We then showed how the graph can be compressed using an algorithm based on bisimulation. We also presented a new algorithm for choosing an elimination order that attempts to maximize the sharing opportunities. We empirically evaluated our approach and demonstrated that, even with a few shared correlations, our approach does significantly better than naive inference approaches.

Even though we mainly focused on computing marginal probability distributions in this article, it is straightforward to extend our approach to speed up computation of joint or conditional probability distributions. Our work so far has identified a number of interesting directions for further research. Approximate inference algorithms are commonly used for large-scale probabilistic inference, and combining those with shared factors may result in significant speedups in query processing. Another direction for future work is to combine our approach which begins with a grounded probabilistic network, with the approaches developed in the lifted inference literature that begin with a first-order description of the probabilistic network. We believe 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.

Acknowledgments This work was supported in part by the National Science Foundation under Grants No. 0438866 and IIS-0546136. We would also like to thank Elena Zheleva for pointing us to the work on directed factor graphs [19] and the anonymous reviewers for their comments and suggestions.

References

1. Andritsos, P., Fuxman, A., Miller, R.J.: Clean answers over dirty databases. In: ICDE (2006)
2. Arnborg, S.: Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey. BIT **25**(1), 2–23 (1985)
3. Bosc, P., Pivert, O.: About projection-selection-join queries addressed to possibilistic relational databases. IEEE Trans. Fuzzy Syst. **13**(1), 124–139 (2005)
4. Boulos, J., Dalvi, N., Mandhani, B., Re, C., Mathur, S., Suciu, D.: Mystiq: a system for finding more answers by using probabilities. In: SIGMOD (2005)
5. Bravo, H., Ramakrishnan, R.: Optimizing MPF queries: decision support and probabilistic inference. In: SIGMOD (2007)
6. Buckles, B., Petry, F.: A fuzzy model for relational databases. Fuzzy Sets Syst. **7**(3), 213–226 (1982)
7. Cheng, R., Kalashnikov, D., Prabhakar, S.: Evaluating probabilistic queries over imprecise data. In: SIGMOD (2003)
8. Choenni, S., Blok, H.E., Leertouwer, E.: Handling uncertainty and ignorance in databases: a rule to combine dependent data. In: DASFAA (2006)
9. Cowell, R., Dawid, A., Lauritzen, S., Spiegelhater, D.: Probabilistic Networks and Expert Systems. Springer, Berlin (1999)
10. Dalvi, N., Suciu, D.: Management of probabilistic data: foundations and challenges. In: PODS (2007)
11. Dalvi, N., Suciu, D.: Efficient query evaluation on probabilistic databases. In: VLDB (2004)
12. Das Sarma, A., Agrawal, P., Nabar, S., Widom, J.: Towards special-purpose indexes and statistics for uncertain data. In: Workshop on Management of Uncertain Data (MUD), Auckland, New Zealand (2008)
13. Das Sarma, A., Theobald, M., Widom, J.: Exploiting lineage for confidence computation in uncertain and probabilistic databases. In: ICDE (2008)
14. De Raedt, L., Kimmig, A., Toivonen, H.: Problog: a probabilistic prolog and its application in link discovery. In: IJCAI (2007)
15. de Salvo Braz, R., Amir, E., Roth, D.: Lifted first-order probabilistic inference. In: IJCAI (2005)
16. Dechter, R.: Bucket elimination: a unifying framework for probabilistic inference. In: UAI (1996)
17. Deshpande, A., Guestrin, C., Madden, S., Hellerstein, J.M., Hong, W.: Model-driven data acquisition in sensor networks. In: VLDB (2004)

18. Dovier, A., Piazza, C., Policriti, A.: A fast bisimulation algorithm. In: International Conference on Computer Aided Verification, Paris, France (2001)
19. Frey, B.: Extending factor graphs so as to unify directed and undirected graphical models. In: UAI (2003)
20. Friedman, N., Getoor, L., Koller, D., Pfeffer, A.: Learning probabilistic relational models. In: IJCAI (1999)
21. Fuhr, N., Rolleke, T.: A probabilistic relational algebra for the integration of information retrieval and database systems. ACM Trans. Inf. Syst. **15**(1), 32–66 (1997)
22. Getoor, L., Taskar, B. (eds.): Introduction to Statistical Relational Learning. MIT Press, Cambridge (2007)
23. Getoor, L., Friedman, N., Koller, D., Taskar, B.: Learning probabilistic models of link structure. J. Mach. Learn. Res. **3**, 679–707 (2002)
24. Gupta, R., Sarawagi, S.: Creating probabilistic databases from information extraction models. In: VLDB (2006)
25. Halpern, J.: An analysis of first-order logics for reasoning about probability. Artif. Intell. **44**(1–2), 167–207 (1990)
26. Huang, C., Darwiche, A.: Inference in belief networks: A procedural guide. Int. J. Approx. Reason. **15**(3), 225–263 (1996)
27. Imielinski, T., Lipski, W. Jr.: Incomplete information in relational databases. J. ACM **31**(4), 761–797 (1984)
28. Jampani, R., Xu, F., Wu, M., Perez, L., Jermaine, C., Haas, P.: MCDB: a monte carlo approach to managing uncertain data. In: SIGMOD (2008)
29. Kanellakis, P., Smolka, S.: CCS expressions, finite state processes, and three problems of equivalence. In: ACM Symposium on Principles of Distributed Computing, Montreal, Canada (1983)
30. Kjaerulff, U.: Triangulation of graphs—algorithms giving small total state space. Technical report, University of Aalborg, Denmark (1990)
31. Koch, C., Olteanu, D.: Conditioning probabilistic databases. In: VLDB (2008)
32. Milch, B., Zettlemoyer, L., Kersting, K., Haimes, M., Kaelbling, L.: Lifted probabilistic inference with counting formulas. In: AAAI (2008)
33. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, Menlo Park (1988)
34. Poole, D.: First-order probabilistic inference. In: IJCAI (2003)
35. Re, C., Dalvi, N., Suciu, D.: Query evaluation on probabilistic databases. IEEE Data Eng. Bull. Spec. Issue Probab. Data Manag. **29**(1), 17–24 (2006)
36. Re, C., Dalvi, N., Suciu, D.: Efficient top-k query evaluation on probabilistic data. In: ICDE (2007)
37. Richardson, M., Domingos, P.: Markov logic networks. Mach. Learn. **62**(1–2), 107–136 (2006)
38. Richardson, T.: A characterization of Markov equivalence for directed cyclic graphs. Int. J. Approx. Reason. **17**(2–3), 107–162 (1997)
39. Rish, I.: Efficient Reasoning in Graphical Models. PhD thesis, University of California, Irvine (1999)
40. Sen P., Deshpande, A.: Representing and querying correlated tuples in probabilistic databases. In: ICDE (2007)
41. Sen, P., Deshpande, A., Getoor, L.: Representing tuple and attribute uncertainty in probabilistic databases. In: DUNE Workshop (ICDM) (2007)
42. Sen, P., Deshpande, A., Getoor, L.: Exploiting shared correlations in probabilistic databases. PVLDB **1**(1), 809–820 (2008)
43. Sen, P., Deshpande, A., Getoor, L.: Bisimulation-based approximate lifted inference. In: UAI (2009)
44. Singh, S., Mayfield, C., Prabhakar, S., Hambrusch, S., Shah, R.: Indexing uncertain categorical data. In: ICDE (2007)
45. Singla, P., Domingos, P.: Lifted first-order belief propagation. In: AAAI (2008)
46. Wang, D., Michelakis, E., Garofalakis, M., Hellerstein, J.: BayesStore: managing large, uncertain data repositories with probabilistic graphical models. In: VLDB (2008)
47. Zhang, N., Poole, D.: A simple approach to Bayesian network computations. In: Canadian Conference on Artificial Intelligence, Banff, Canada (1994)
48. Zhang, N., Poole, D.: Exploiting causal independence in Bayesian network inference. J. Artif. Intell. Res. **5**, 301–328 (1996)