Representing and Querying Uncertain Data

Prithviraj Sen

Dissertation submitted in partial fulfillment
of requirements for the degree of Doctor of Philosophy to the
Department of Computer Science
University of Maryland, College Park

Thesis Advisers: Dr. Amol V. Deshpande and Dr. Lise C. Getoor

Abstract

There has been a longstanding interest in building systems that can handle uncertain data. Traditional database systems inherently assume exact data and harbour fundamental limitations when it comes to handling uncertain data. In this dissertation, we present a probabilistic database model that can compactly represent uncertainty models in full generality. Our representation is associated with precise and intuitive semantics and we show that the answer to every user-submitted query can be obtained by performing probabilistic inference. To query large-scale probabilistic databases, we propose a number of techniques that help scale probabilistic inference. Foremost among these techniques is a novel lifted inference algorithm that determines and exploits symmetries in the uncertainty model to speed up query evaluation. For cases when the uncertainty model stored in the database does not contain symmetries, we propose a number of techniques that perform approximate lifted inference. Our techniques for approximate lifted inference have the added advantage of allowing the user to control the degree of approximation through a handful of tunable parameters. Besides scaling probabilistic inference, we also develop techniques that alter the structure of inference required to evaluate a query. More specifically, we show that for a restricted model of our probabilistic database, if each result tuple can be represented by a boolean formula with special characteristics, i.e., it is a read-once function, then the complexity of inference can be drastically reduced. We conclude the dissertation with a listing of directions for future work.

To Mummy, Baba and Shiny,
Dadu and Diddey,
Amma and Dada,
for always supporting me

Contents

1 Introduction 1
1.1 A Small Example . . . 2
1.2 Our Approach . . . 4
1.3 Outline and Contributions . . . 9

2 Related Work 12
2.1 Uncertainty and Databases . . . 12
2.2 First-Order Graphical Models . . . 14
2.3 Lifted Inference . . . 16
2.4 Query Evaluation in Probabilistic Databases . . . 17
2.5 Conclusion . . . 19

3 Representing Uncertain Data 21
3.1 Background: Probabilistic Graphical Models . . . 21
3.2 Probabilistic Databases with Probabilistic Graphical Models 24
3.2.1 Possible World Semantics . . . 24
3.2.2 Examples . . . 24
3.3 Query Evaluation . . . 28
3.3.1 Example . . . 31
3.3.2 General Relational Algebra Queries . . . 33
3.3.3 Complexity of probabilistic inference . . . 36
3.4 Experiments and Discussion . . . 37
3.4.1 Case Study: Querying Publication Datasets . . . 37
3.4.2 Experiments with TPC-H Benchmark . . . 39
3.4.3 Aggregation Queries . . . 40
3.5 Conclusion . . . 41

4 Bisimulation-based Lifted Inference for Probabilistic Databases 43
4.1 Motivating Example . . . 45

4.1.1 Limitations of Naive Inference Algorithms 47
4.2 Inference with Shared Factors 47
4.2.1 The ELIMRV operator 48
4.2.2 The RV-ELIM Graph 50
4.2.3 Identifying Shared Factors 50
4.2.4 Bisimulation for RV-ELIM Graphs 54
4.2.5 Inference with the Compressed RV-ELIM Graph . . . 56
4.3 Computing Elimination Orders 57
4.4 Experimental Evaluation 60
4.4.1 Car DB experiments 63
4.4.2 Experiments with uncertain join attributes 64
4.4.3 Experiments with TPC-H data 65
4.5 Discussion 66
4.6 Conclusion 69

5 Approximate Lifted Inference For Probabilistic Databases 70
5.1 Running Example 72
5.2 Approximate Lifted Inference with Approximate Bisimulation 73
5.3 Approximate Lifted Inference with Factor Binning 77
5.4 Unified Lifted Inference 82
5.4.1 Bounded Complexity Lifted Inference 82
5.4.2 Unified Lifted Inference Engine 83
5.5 Experimental Evaluation 83
5.5.1 Synthetic Bayesian Network Generator 84
5.5.2 Lifted Inference with Approximate Bisimulation . . . 85
5.5.3 Approximate Lifted Inference with Factor Binning . . 86
5.5.4 Unified Lifted Inference Engine 87
5.5.5 Experiments on Real-World Data 88
5.6 Conclusion and Future Work 90

6 Read-Once Functions and Probabilistic Databases 91
6.1 Preliminaries 93
6.1.1 Read-Once Functions: Unateness, P_4-Free and Normality 94
6.1.2 Hierarchical Queries and Read-Once Functions . . . 96
6.2 Generating Read-Once Expressions 98
6.2.1 A Simple Read-Once Function Aware Query Evaluator 98
6.2.2 Limitations of the Simple Query Evaluator 99
6.3 Read-Once expressions for Conjunctive Queries 100
6.3.1 Piece-Wise Building of Read-Once Functions 101

6.3.2 A Co-Tree Merging Algorithm 112
6.4 Experiments 118
6.5 Discussion and Conclusion 119

7 Conclusion **122**
7.1 Summary of Contributions 122
7.2 Avenues for Future Work 123
7.3 Conclusion 125

Chapter 1

Introduction

Many applications produce massive amounts of data that needs to be stored in an organized manner so that users can sift through and find information that is of interest. Database systems have become the de facto standard for storing such large amounts of data. At least from the end user's perspective, one of the most important reasons for the success of database systems is the declarative querying capabilities they offer through query languages such as SQL. The use of a declarative query language allows the lay user to pose complex queries against the underlying data without having to worry about algorithmic or efficiency issues associated with evaluating the query.

Unfortunately, current database systems are not well suited to store data with uncertainties. If we state that John's salary is $55,000 per annum, then John cannot have any salary other than $55,000. We cannot, for example, state that John's salary could lie anywhere between $55,000 to $60,000 per annum or that the temperature on the first floor measured through a sensor at 10:28AM this morning was more likely to be $52.6°$ F than $52.8°$ F. We refer to such data with uncertainties as *uncertain* data or *inexact* data.

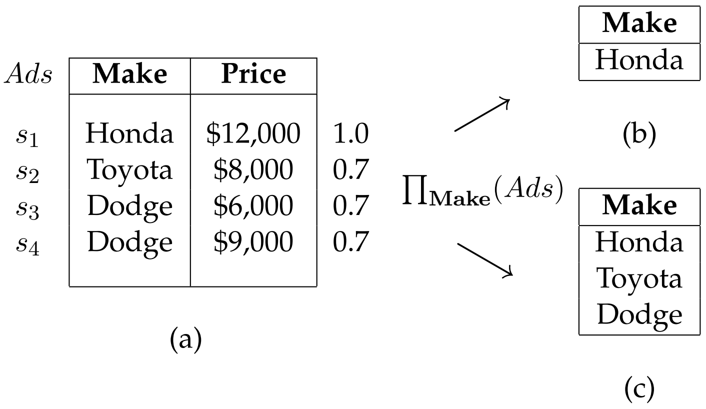
A number of real world applications produce large amounts of uncertain data. Examples include data collected from sensor networks [Deshpande et al., 2004], information extraction systems [Jayram et al., 2006] and mobile object tracking systems [Cheng et al., 2003]. Traditional database management systems are not suited for storing uncertain data, which means that the declarative querying capabilities offered by database systems are unavailable to people who need to deal with and search through such data to find information of interest.

In this dissertation, our aim is to develop a database system that can store and query uncertain data. To achieve our goal, we need to answer two fundamental questions: 1) How do we represent uncertainty in a database and 2) How do we use the uncertainty model along with the data to produce relevant answers to a user-submitted query? In the ensuing chapters, we will see how we answer the former question by combining traditional database ideas with uncertainty representation models from machine learning. Further, we also show how efficient query evaluation can be performed in such databases by developing novel algorithms based on ideas from graph-theory. But before we go any further, let us consider a small example that illustrates some of the differences between querying exact data using traditional database query processing techniques and handling uncertain data.

1.1 A Small Example

Figure 1.1(a) shows a small relation, Ads, where each row corresponds to an advertisement (ad) that we pulled off from a pre-owned car sales website. For simplicity, we depict only the **Make** and **Price** of the cars associated with each ad in Figure 1.1(a); a real pre-owned car sales database will likely contain many more attributes of interest. Ads contains four ads, the first (depicting a Honda for sale) being an instance of certain data and the last three (s_2, s_3 and s_4) being uncertain. Also shown in Ads against each tuple, are numbers depicting how certain/uncertain each tuple is. For now, we will refrain from specifying exactly what these numbers mean other than saying that they are a measure of how likely it is for the corresponding tuple to exist in the real world or their *degree of certainty*. Such numbers may be useful in capturing the fact that many ads posted on websites remain visible even after the car in question has been sold and thus with the passage of time it is less likely for a car being advertised to be still up for sale. For the purposes of presenting our example, a tuple with degree closer to 1 increases the chance of its being present in the real world, while a degree closer to 0 decreases the chance of that ad still being valid. Thus, in Figure 1.1(a), s_1 is an instance of exact data where we know that a car of make Honda is definitely up for sale, while s_2, s_3 and s_4 are uncertain, we are not quite sure if those cars are still available but there is a good chance (since 0.7 is closer to 1 than to 0) of them being still available for sale.

Suppose a user is interested in finding out the makes of the cars that are for sale and so wants to issue the query $\prod_{\textbf{Make}}(Ads)$. Now, we have a slight problem since we don't know how to deal with the degrees of tuples. We will consider two simple approaches. In the first approach, we



Ads

	Make	Price	
s_1	Honda	\$12,000	1.0
s_2	Toyota	\$8,000	0.7
s_3	Dodge	\$6,000	0.7
s_4	Dodge	\$9,000	0.7

(a)

$\prod_{\textbf{Make}}(Ads)$

Make
Honda

(b)

Make
Honda
Toyota
Dodge

(c)

Figure 1.1: (a) A small pre-owned car sales database. (b) $\prod_{\textbf{Make}}(Ads)$ ignoring uncertain data. (c) $\prod_{\textbf{Make}}(Ads)$ treating uncertain data just like exact data but with an extra attribute.

will simply ignore all uncertain data (s_2, s_3 and s_4) and not return them as query results. The result (shown in Figure 1.1(b)) contains only one result and is unsatisfactory because it seems to suggest that the only make available to the user is Honda and if she doesn't want to purchase a car of this make then s/he doesn't have any other cars to choose from which is not entirely true. There is a possibility that the other (uncertain) tuples in *Ads* are still valid ads and the user should be able to find out about these through her/his query. The main issue here is that when we throw away (uncertain) data we actually throw away information and this leads to query results of worse quality. In many domains, such as sensor networks, the bulk of data collected is uncertain due to reasons ranging from uncertainty associated with the sensing mechanism of the sensors to inadequate number of sensors being placed in the environment being measured. Throwing away the uncertain data, in such cases, leaves the database with precious little data to work with which, albeit certain, is still unlikely to be enough to ensure good quality query results.

A second approach is to treat the degrees of certainty of the tuples as an extra attribute and append it to list of attributes of *Ads*. Executing the same query under this paradigm returns the result shown in Figure 1.1(c) which now indicates that there are Toyotas and Dodges up for sale besides the Honda. Unfortunately, this result is not quite satisfactory due to two reasons:

- Honda vs. Toyota: Notice that the query result gives Honda and Toyota equal billing. However, the Honda result is derived from a *certain* tuple whereas the Toyota result is derived from an *uncertain* tuple which may not correspond to a valid ad.

- Toyota vs. Dodge: The query also gives Toyota and Dodge equal billing, even though, Toyota was derived from an uncertain tuple associated with a degree of 0.7 whereas Dodge was derived from *two* uncertain tuples each associated with a degree of 0.7. This suggests that it is more likely for the user to find a Dodge up for sale and the query result does not reflect this.

The first discrepancy (Honda and Toyota getting equal billing in the result) suggests that while evaluating a query we need to look at the degrees of the tuples, the second discrepancy (Toyota and Dodge getting equal billing) suggests that we may also need non-trivial reasoning mechanisms to combine and compare degrees if we are to return useful query results.

The above example should make it clear that handling uncertain data is quite different (and perhaps more challenging) than handling exact data. Uncertain data is typically richer than exact data and the richness is because of the quantitative expression of uncertainties which is something traditional database research has not considered in depth. Effectively storing and querying uncertain data requires that we use the information present in the uncertainties appropriately so that we can help users sift through and arrive at answers of interest. We next describe, at a high level of abstraction, the basic ideas used in this dissertation to design such a database system.

1.2 Our Approach

Any system that deals with uncertain data has to begin by describing a representation scheme that allows users to compactly yet flexibly represent the uncertainties present in the data. In this dissertation, we borrow extensively from the machine learning literature and use *probability theory* coupled with the language of *probabilistic graphical models* to augment databases so that they can represent uncertain data. For this reason, henceforth, we will refer to a database containing uncertain data as a *probabilistic database*. Probabilistic graphical models [Cowell et al., 1999; Pearl, 1988] are compact representations of joint probability distributions involving a large number of random variables. By redefining a probabilistic database in terms of a probabilistic graphical model, we inherit all of their nice compactness properties. Additionally, we show how to use probabilistic graphical models to represent all the different kinds of uncertainty that a user might want to express in a probabilistic database along with *correlations*. Correlations allow one to couple uncertainties among multiple random variables. For instance, relating back to the example from the previous section, suppose Ads contained another attribute **Color**. Also, suppose that for some ad we neither knew the color nor the make but we know that if the make of the car in the ad is Honda then its color can be one from a restricted set of colors, say red or black. Then, this coupling between color and make attributes can be represented in a probabilistic database by expressing it as a correlation. A number of applications produce uncertain data with known correlations, such as sensor networks for habitat monitoring where it has been shown that utilizing spatial and temporal correlations can drastically improve the quality of query results [Deshpande et al., 2004]. In short, our formulation of a probabilistic database can represent:

- attribute uncertainty: tuples with uncertain attribute values,
- tuple uncertainty: tuples whose existence we are unsure of,
- intra-tuple attribute-attribute correlation: tuples whose attribute values are uncertain and correlated,
- inter-tuple attribute-attribute correlation: attribute values from different tuples that are both uncertain and correlated (note that these tuples can belong to different relations),
- inter and intra attribute value-tuple existence correlations.

In the previous section, when we discussed simple ways of handling uncertain data using traditional database systems, we showed how such techniques led to query results that were qualitatively unsatisfactory. However, we did not discuss what the correct query result should look like. This, in part, relates to the question of assigning semantics to a probabilistic database. What does a probabilistic database actually mean? *Possible worlds semantics* [Halpern, 1990] is one set of semantics that has formed the basis of numerous probabilistic models proposed in the machine learning literature and it is known that databases based on possible worlds semantics are associated with particularly intuitive and precise query evaluation semantics [Dalvi and Suciu, 2004; Fuhr and Rolleke, 1997]. Essentially, under possible worlds semantics, a probabilistic database is simply a distribution over many traditional databases each referred to as a possible world.

Query evaluation under possible worlds semantics means evaluating the query against each possible world (which we know how to do since each possible world is a database devoid of any uncertainty) and for each result adding up the probabilities of all possible worlds that produce the result. Fortunately, our probabilistic graphical models based formulation of probabilistic databases lends itself naturally to possible worlds semantics thus defining precise semantics for the query evaluation problem.

Of course, defining the query evaluation problem by associating it with precise semantics is one thing and efficiently evaluating queries is another. Even though possible worlds semantics precisely defines what the result of posing a query to a probabilistic database should be, it does not provide an efficient means of computing it. To this end, we develop an approach to evaluating a user-submitted query by reformulating it as probalistic inference problem in an appropriately constructed graphical model. More precisely, given a query q (expressed in some declarative query language such as relational algebra or SQL) to be evaluated against a probabilistic database with an underlying probabilistic graphical model, we show how to augment the probabilistic graphical model on the fly to construct an augmented probabilistic graphical model from which we can compute the result of q by solving a probabilistic inference problem. Exactly how we augment the probabilistic graphical model underlying the probabilistic database depends on q and the operators appearing in it. This reformulation in terms of probabilistic inference has two clear benefits:

1. The general problem of evaluating queries on probabilistic databases is known to be #P-complete [Dalvi and Suciu, 2004], but we also know that for some queries this problem is solvable. By expressing a query evaluation problem as a probabilistic inference problem to be evaluated on an appropriately constructed probabilistic graphical model, we can now identify exactly which queries lead to hard problems since the hardness of running probabilistic inference is well understood and can be determined by measuring the *treewidth* [Arnborg, 1985] of the probabilistic graphical model.

2. By reformulating the query evaluation problem as a probabilistic inference problem, we allow access to using the host of probabilistic inference algorithms developed in the machine learning literature, and by appropriately choosing the inference algorithm, we can obtain various time vs. space vs. accuracy trade-offs depending on the requirements of the user.

Besides utilizing probabilistic inference algorithms and the various optimizations they come with to evaluate queries on probabilistic databases, another aspect that affects the complexity of the query evaluation problem is the data stored in the probabilistic database. We can reduce the complexity of query evaluation by exploiting special properties of the data stored in the probabilistic database at hand. One such property is the presence of *shared correlations*, where the same correlations and uncertainties repeatedly occur in the data many times over. For instance, in the example from the previous section, if we had two ads concerning Honda vehicles and we didn't know their respective colors then it is likely that the **Color** attribute values of the two corresponding tuples would be governed by the same uncertainties and probability distributions. Essentially, uncertainties and probability distributions rarely vary on a tuple-to-tuple basis and usually come from general statistics, which leads to repeated probability factors and correlations. Besides occurring naturally in the data, shared correlations are also introduced when we augment the probabilistic graphical model defined on the base data to evaluate a query. In the presence of shared correlations, any standard inference algorithm would treat each copy of a shared correlation separately and perform the same computation steps repeatedly. We develop an inference algorithm based on bisimulation [Kanellakis and Smolka, 1983] that helps identify such shared correlations and avoid repetitive computations. We validate our algorithm by showing that even in the presence of a few shared correlations our algorithm does significantly better than standard inference algorithms.

We further develop our approach to leveraging shared correlations while evaluating queries by developing approximate versions of the above inference algorithm. For many applications, perfect accuracy in query results may not be a requirement and some errors can be tolerated; our approximate inference techniques are aimed towards such applications where we make more aggressive use of shared correlations and trade-off accuracy to reduce time spent to run inference. More specifically, we propose two different ways to implement approximate inference, both closely related to bisimulation. Both of these techniques can be combined for more aggressive exploitation of shared correlations. Further, our techniques can be combined with bounded complexity inference techniques such as mini-buckets [Dechter and Rish, 2003]. We report experiments on both synthetic and real data to show that in the presence of symmetries, run-times for inference can be improved significantly, with approximate lifted inference providing orders of magnitude speedup over standard inference algorithms and the previously developed shared correlations-aware exact inference algorithm.

In the last part of the dissertation, our focus remains on efficient query evaluation but the questions we ask are slightly different. Recall that while evaluating queries, we first take the (uncertain) data from the database and the user submitted query, and generate a probabilistic graphical model on which we need to run inference to compute the result of the query. Note that, for the same query, many different query plans are possible. Further, different query plans of the same query may result in different probabilistic graphical models, all of which are equivalent with respect to the results of inference. The obvious question to be asked in such a scenario is: are all of these graphical models similar in complexity or is there a graphical model/query plan on which it is easier to run inference, in other words, is there a low-treewidth graphical model? Previous attempts to answer this question led to the concept of *hierarchical queries*. Hierarchical queries represent the class of queries for which there exists a particular query plan that lets us generate a tree-structured probabilistic graphical model (which is easy to run inference on) for any (tuple-independent) probabilistic database. However, because of their query-centric definition that does not involve the database, hierarchical queries represent an overly pessimistic way of defining the class of tractable queries. It is easy to construct examples where a non-hierarchical query run on an appropriate database gives rise to a tractable query evaluation problem. In the final part of the dissertation, we go beyond the notion of hierarchical queries. Our goal is to develop query evaluation algorithms that, given the database and the query, generate a tree-structured graphical model (if it exists) leveraging both the data *and* the query. For a tuple-level probabilistic database, it is easy to show that every result tuple is associated with a boolean formula and query evaluation reduces to computing the marginal probability for the boolean formula holding true. It is also easy to see that if the result tuple is such that its associated boolean formula can be factorized into a form where every boolean variable (or tuple-existence variable, in our case) appears not more than once, then its marginal probability can be computed efficiently. We propose novel approaches that generate such factorizations of result tuples produced by evaluating queries. By doing so, we leverage both data and query to solve queries on probabilistic databases in the most efficient manner possible.

This dissertation forms the first few steps in developing a full-fledged database system that can manage and store uncertain data. Given the level of interest in probabilistic databases and the wide array of applications that can benefit from developments in this area of research, it should come as no surprise that much work still needs to be done before we see a viable, useful system being released and that the number of possible directions of future work far exceeds than what can be accomodated in a few pages of this dissertation. Still, some of these directions are more compelling and require more urgent attention than others. We conclude the dissertation with a summary of contributions made and a listing of these possible avenues for future work.

1.3 Outline and Contributions

The rest of the dissertation is organized as follows:

- In the next chapter, we begin by discussing prior related work. The work described in this dissertation contributes and is related to a number of different fields of research, and in Chapter 2 we review the more relevant references organized according to different areas of research to help the reader place our contributions in context.

- Chapter 3 describes the basic representation scheme which can express all types of uncertainties that one may want to express in a relational database. This chapter is based on work that appeared in Sen and Deshpande [2007]; Sen et al. [2007, 2009b]. More precisely, in this chapter:

  - We define probabilistic databases in terms of probabilistic graphical models.
  - We show how our formulation naturally lends itself to possible worlds semantics.
  - We show that the query evaluation problem can be recast as a probabilistic inference problem in an appropriately constructed probabilistic graphical model.
  - We show how to construct the probabilistic graphical model for a query on the fly at query time.
  - We show how standard probabilistic inference algorithms, along with various optimizations, can be used to answer queries in our probabilistic database.

- In Chapter 4, we develop the first inference algorithm that exploits shared correlations which is the first inference algorithm of its kind that can be applied to any probabilistic graphical model (even ones that do not arise out of query evaluation for probabilistic databases). This chapter is based on work that appeared in Sen et al. [2008a, 2009b]. More precisely, in this chapter:

  - We define shared correlations and motivate their presence in uncertain data using examples.
  - We develop an inference algorithm based on bisimulation that exploits shared correlations to avoid repetitive computation.
  - We develop an effective heuristic to construct elimination orders (a key step in most exact inference algorithms) and show that our heuristic produces orders that work well with our inference algorithm.
  - We validate our inference algorithm by running experiments and comparing against standard inference algorithms.

- In Chapter 5, we develop approximate versions of the previously developed shared correlation-aware inference algorithm. This chapter is based on work that appeared in Sen et al. [2009a]. More precisely, in this chapter:

  - We devise two different ways to implement approximate inference with shared correlations: one based on approximate bisimulation and another based on factor binning.
  - We show that these two approaches can be combined together for more aggressive exploitation of shared correlations.
  - We also show how these techniques can be combined with bounded complexity inference mechanisms.
  - We develop a *unified* inference engine that, through the use of a handful of tunable parameters, allows the user to control the degree of approximation and to what extent we want to exploit shared correlations, thus allowing the user to achieve a trade-off between accuracy of inference and time spent running inference.
  - We demonstrate through experiments on both synthetic and real data how the approximate inference procedures can provide orders of magnitude speedup over standard inference algorithms and our previously developed shared correlation-aware exact inference algorithm.

- In Chapter 6, we develop query evaluation algorithms that generate tree-structured graphical models (if possible) given the query and the database. More precisely, in this chapter:
  - We review the concept of hierarchical queries.
  - We review the relationship between hierarchical queries, tree-structured graphical models and read-once functions.
  - We propose a very simple query evaluation algorithm that makes use of previous work on read-once functions and generates tree-structured graphical models whenever possible given any query to be run on a probabilistic database.
  - We consider the special case of conjunctive queries and show that read-once functions can be generated more efficiently for this case.
- We conclude the dissertation with Chapter 7 which contains a summary of contributions and a listing of possible avenues of future work.

Chapter 2

Related Work

The broader field of uncertainty management in databases has seen a lot of work in recent years. In this chapter, we attempt to list the more relevant works and contrast them with the contributions made in this dissertation. Moreover, the work described in the ensuing chapters relates to various different fields of research besides database systems such as machine learning. In what follows, we attempt to divide the related work according to the various fields of research and within each sub-division, we mention how our work relates to relevant prior work.

2.1 Uncertainty and Databases

The topic of representing and modeling uncertainty has been in the collective conscience of the database community for a fairly long period of time. Consequently, a wide array of approaches have been proposed. Very early on, the subject of dealing with *null* values or logical uncertainty in a principled manner received a fair amount of attention [Imielinski and Lipski, Jr., 1984]. More recently, there has been more work along these lines that attempt to concisely represent such databases by employing vertical partitioning methods [Antova et al., 2007]. Das Sarma et al. [2006] explore various different models of logical uncertainty with varying representation power.

When it comes to dealing with uncertainty involving a measure of uncertainty or beliefs, a number of different approaches have been proposed. However, the consensus seems to be that probability theory has the right balance of power and tractability, which is why the bulk of research in this area falls under the sub-area known as *probabilistic databases*. Barbara et al.

[1992] is one of the earliest works along these lines which explores attribute uncertainty models focussing on intra-tuple correlations. Fuhr and Rolleke [1997] is perhaps one of the earliest works that proposed a coherent, albeit simplistic, model of a probabilistic database; the application in focus was combining information retrieval and database techniques into one single system. *ProbView* [Lakshmanan et al., 1997] posits that each tuple is associated not with a point estimate of probability but a range, and goes on to develop query evaluation techniques based on linear programming. In recent developments, Dalvi and Suciu [2004] present a probabilistic database model based on simple semantics (possible worlds) and show how query rewriting techniques can help solve intractable queries under this model.

In Chapter 3, we develop compact yet powerful models of probabilistic databases based on probability theory and factored representations of joint probability distributions. Our approach is closely related to representing uncertainty with probabilistic graphical models from the machine learning literature. Our techniques allow the user to express all kinds of uncertainty within a relational database, along with correlations. We also show that our model of a probabilistic database is associated with precise and intuitive semantics, possible worlds, and query evaluation can be performed by running standard probabilistic inference algorithms on an appropriately constructed probabilistic graphical model. The work described in Chapter 3 illustrates that it is possible to go beyond simplistic tuple-level uncertainty models that assume complete independence and still be able to come up with models of probabilistic databases that have desirable properties such as simple semantics and tractable querying.

Chapter 3 is based on previously published works [Deshpande et al., 2008; Sen and Deshpande, 2007; Sen et al., 2007, 2009b]. In Sen and Deshpande [2007], we introduced models of probabilistic databases that allowed tuple-level uncertainty with correlations, of both intra-relation and inter-relation varieties, and was perhaps one of the first works to include correlations. This was a significant departure from prior work, both Fuhr and Rolleke [1997] and Dalvi and Suciu [2004] worked with tuple-level uncertainty models assuming complete independence among tuples. Since most applications produce data which requires modeling correlations, our work significantly broadened the applicability of probabilistic databases. Subsequently, in Sen et al. [2007, 2009b], we made our model of probabilistic databases more general by including attribute and tuple level uncertainty, and also by including first-order graphical models based on shared correlations. The concept of shared correlations is introduced in Chapter 4 wherein we represent numerous identical correlations together instead of representing them separately. This allows our uncertainty model to become even more compact. Shared correlations are the basis of state-of-the-art first-order uncertainty representation models from machine learning (e.g., probabilistic relational models [Friedman et al., 1999] and Markov logic networks [Richardson and Domingos, 2006], reviewed in more detail below).

There are a number of other works that also come under the umbrella of probabilistic databases and have been tried over the years. Fuhr and Rolleke [1996] proposed an extension based on NF2 relational algebra. *Trio* [Benjelloun et al., 2006] proposes the concept of *x-tuples* which is basically an uncertain tuple represented by its various alternatives. *MystiQ* [Re et al., 2006] proposes the *block-independent disjoint* model which is similar to x-tuples. *SPROUT* [Koch and Olteanu, 2008] employs a model referred to as a *world-set tree* and Li and Deshpande [2009] employ a similar *and/xor* tree. None of these approaches discuss concisely describing the uncertainty model using shared correlations and first-order graphical models like we do in Chapter 4.

Among the various models that go beyond the use of probability theory, there are models based on fuzzy logic [Bosc and Pivert, 2005; Buckles and Petry, 1982] and models based on Dempster-Shafer theory [Choenni et al., 2006].

2.2 First-Order Graphical Models

On the topic of representing uncertainty, researchers in machine learning have devoted a lot of thought and time to developing concise models that possess the requisite representation power. The result is the development of *probabilistic graphical models* (PGM), that contain as special cases *Bayesian networks* [Pearl, 1988] and *Markov networks* [Cowell et al., 1999]. As reviewed in Chapter 3, a probabilistic graphical model represents a joint distribution among many random variables by representing it in little pieces called *factors*. Bayesian networks include only directed dependencies and Markov networks only allow undirected dependencies. There exist generalizations that allow a mix of directed and undirected dependencies but disallow directed cycles such as *chain graphs* [Lauritzen, 1996] and *factor graphs* [Frey, 2003]. Further, generalizations that allow directed cycles have also been studied [Richardson, 1997]. Our approach outlined in Chapter 3 can make use of any of these approaches.

However, PGMs are not without limitations. These models are easier to visualize, reason about and deal with when the number of random variables range in a few hundreds or less. Even in a small-to-moderately sized probabilistic database we are likely to exceed this number which makes it unreasonable to assume that having an uncertainty model in terms of a PGM will be easy to handle. Machine learning researchers, specifically *statistical relational learning* researchers (SRL), in the past decade or so, have paid cognisance to this fact and have come up with a new class of PGMs frequently referred to as *first-order graphical models* (FO-models). FO-models are essentially PGMs with an additional layer of specification that uses first-order rules to specify correlations among classes of random variables. The same first-order rule applies to all random variables belonging to the respective classes, and these are, essentially, *shared correlations* (Chapter 4). This allows FO-models to be compact, easier to maintain, visualize and also, statistically easier to estimate from data. Listing the various FO-models produces a veritable alphabet soup: PRMs [Friedman et al., 1999] (probabilistic relational models), RMNs [Taskar et al., 2002] (relational Markov networks), MLNs [Richardson and Domingos, 2006] (Markov logic networks), BLOGs [Milch et al., 2005] (Bayesian logic) etc. We refer the interested reader to Getoor and Taskar [2007] for a more extensive and detailed survey.

Our use of shared correlations and FO-models to specify models of uncertainty in probabilistic databases means we have close connections to this area of work although there are some differences. Most of the work on FO-models has concentrated on how to specify and learn a class-level probabilistic model for relational data; and answering queries expressed in a standard query language (e.g., relational algebra or SQL) was not their main focus as is the case in research on probabilistic databases. In fact, very few FO-models proposed in the literature even consider querying with a structured query language. ProbLog [De Raedt et al., 2007], which uses Prolog, is perhaps the only exception. We believe that the best way to view the work described in this dissertation is to look upon it as taking the best of both FO-models and probabilistic databases, since the representation schemes we develop in Chapter 3 allow us to represent shared correlations in databases while the query evaluation algorithms we develop later (in Chapter 4 and Chapter 5) can exploit the same shared correlations to allow the user to efficiently and declaratively query the probabilistic database.

2.3 Lifted Inference

Even though probabilistic inference can be used to evaluate queries in probabilistic databases, there may still be cases when probabilistic inference is inefficient. In Chapter 4 and Chapter 5, we show how to exploit special properties of the uncertain data, i.e., shared correlations, to speed up inference during query evaluation. Shared correlations occur in the data when the same uncertainties and probability distributions occur repeatedly. In such a case, standard inference algorithms treat each instance of these shared correlations separately and repeatedly perform the same computation steps. We develop an approach based on the graph-theoretic concept of *bisimulation* [Kanellakis and Smolka, 1983; Paige and Tarjan, 1987] that avoids such repeated computation. In Chapter 4, we present an exact inference algorithm based on these ideas and, in Chapter 5, we extend the techniques in multiple different ways to perform approximate inference.

The inference algorithms presented in Chapter 4 and Chapter 5 are closely related to *lifted inference* algorithms [de Salvo Braz et al., 2005; Poole, 2003] developed by the SRL community. Lifted inference aims to exploit the symmetry provided by FO-models in the form of shared correlations to achieve more efficient inference. The basic idea behind lifted inference is to develop inference algorithms that instead of summing over random variables and multiplying factors, sum over sets of random variables and multiply sets of factors, thus reducing redundant computation. Most works in lifted inference assume that they are provided a PGM expressed as an FO-model and that the symmetry of shared correlations is explicitly provided in first-order logic. In Chapter 4, we make no such assumptions. This is mainly because to evaluate queries in probabilistic databases one first needs to build the PGM on which we need to perform inference (described in detail in Chapter 3) and it is not straightforward to obtain a PGM expressed as an FO-model via this approach. Instead, our bisimulation-based approach to lifted inference discovers the symmetry due to shared correlations in the constructed PGM on the fly. To the best of our knowledge, our approach is the first general lifted inference approach that can be applied to any PGM.

Some attempts have been made by the probabilistic database community to make more direct use of existing lifted inference work. In Wang et al. [2008], the authors state that among the various issues complicating the use of *Parameterized Variable Elimination* [Poole, 2003] for query evaluation in probabilistic databases is the presence of evidence and, presumably, handling joins among different relations; Wang et al. only report experiments on single-relation selection queries.

Poole [2003] was one of the first to show that variable elimination [Zhang and Poole, 1994] can be modified to directly work with FO-models to avoid propositionalization during inference. Subsequently, de Salvo Braz et al. [2005] further developed on Poole's work and referred to it as *inversion elimination*. They also introduce another technique for lifted inference known as *counting elimination* which is more expensive than inversion elimination (since it requires considering all possible combinations of assignments to a set of random variables [de Salvo Braz et al., 2005]) but can help in certain situations where the complexity of the ground model renders ground inference infeasible. It is straightforward to show that our bisimulation-based approach to lifted inference subsumes inversion elimination (and partial inversion [de Salvo Braz et al., 2006]). We provide more discussion illustrating this connection, along with an example, in Section 4.6.

Lifted inference is still a very young field, but there has been some work on designing approximate lifted inference algorithms. Jaimovich et al. [2007]; Kersting et al. [2009]; Singla and Domingos [2008] all, essentially, propose to use a bisimulation-like algorithm on the factor graph [Kschischang et al., 2001] representing the probabilistic model to find clusters of random variables that send and receive identical messages which helps speed up inference with loopy belief propagation (LBP) [Yedidia et al., 2000], a ground approximate inference algorithm. Our work on approximate lifted inference described in Chapter 5 differs from lifted LBP on two distinct counts. First, except for Kersting et al., the above works depend on receiving the FO-model as input, whereas our approximate lifted inference techniques, in effect, determine the first-order representation on the fly. Second, as Singla and Domingos acknowledge, LBP often has problems with convergence, whereas the approaches we describe Chapter 5 are always guaranteed to converge.

2.4 Query Evaluation in Probabilistic Databases

Keeping with the wide array of representation schemes proposed, a number of diverse schemes for query evaluation in probabilistic databases have also been tried. Until the last decade, there were mainly two competing schools of thought: *Intensional* and *Extensional* query evaluation. Intensional evaluation always provides coherent results adhering to possible worlds semantics. Extensional evaluation, however, does not always come with guaranteed semantics, so in that sense, the results may be wrong, even though extensional evaluation is always cheaper than intensional evaluation. Dalvi and Suciu [2004] illustrated that these two approaches were not completely at loggerheads, and that there exists a subset of SQL whose queries are such that when run under extensional semantics with a particular plan lead to results in accordance with possible worlds semantics. This subset of relational algebra has since been referred to as queries with *safe plans* [Dalvi and Suciu, 2004] or *hierarchical queries* [Dalvi and Suciu, 2007].

Interestingly, it is easy to show that safe plans always give rise to tree-structured PGMs when expressed in our formulation. This means that our approach to evaluating queries is also quite efficient when extensional evaluation provides correct query results (since tree-structured PGMs are easy to run inference on), besides always adhering to possible world semantics. In Chapter 6, we take this idea one step further. Instead of looking at the query to find out if it is tractable or not, as is done in most other works on tractable queries [Dalvi and Suciu, 2007, 2004; Olteanu and Huang, 2009, 2008], we ask if the PGM constructed for query evaluation can be re-ordered into a tree-structured graphical model. Essentially, the definition of hierarchical queries [Dalvi and Suciu, 2007] does not take into account the data contained in the database. One way to describe this tractable class of queries is to say that if a query is tractable for all possible databases then it belongs to this class. However, this represents a very pessimistic way of defining tractable queries. Since the PGM on which we need to run inference to compute the results is a combination of the query and the database, we need to look at both aspects in order to determine tractability. In Chapter 6, we explore these issues and make connections to literature in graph theory on factorizing boolean formulas [Golumbic et al., 2006]. We develop algorithms that take each result tuple and explore whether the corresponding PGM can be converted to a tree-structured one, if so then we proceed to building this tree-structured PGM and running inference on it. We also show that for a large class of queries, conjunctive queries without self-joins, some of the checks that need to be performed in the most general case can be avoided, resulting in more efficiency. By doing so, we show that both data and query can be leveraged to the fullest to evaluate queries over probabilistic databases.

The original work on hierarchial queries [Dalvi and Suciu, 2004] mainly considered queries involving equality join predicates. Since then, there have been other works along these lines extending the notion to various other operators. In recent work, there have been attempts to show that, at least in some cases, inequality predicates, $\neq$ [Olteanu and Huang, 2008] and $>, <$ [Olteanu and Huang, 2009], also allow for tractable query evaluation. These approaches are currently out of the scope for our framework since they may not lead to tree-structured PGMs. As regards the assumption of no self-joins in the query, Dalvi and Suciu [2007] is the only work we are aware of that attempts to remove this assumption. From the machine learning community, Darwiche [2002] proposes utilizing boolean formula factorization algorithms so that a given probabilistic model can be compiled into a more tractable form usually referred to as an *arithmetic circuit*. This is advantageous because performing inference using the compiled arithmetic circuit is more efficient than performing inference with the original probabilistic model. More importantly, Darwiche can handle attribute uncertainty (they consider general PGMs). However, Darwiche relies on the use of an exponential-sized intermediate representation called *multi-linear formula*. In Chapter 6, we consider the simpler case of a probabilistic database with tuple-level uncertainty. Developing techniques that can handle attribute-level uncertainty is delegated to future work.

Since our work illustrating that query evaluation requires probabilistic inference (Chapter 3), a number of other works have utilized different inference procedures. In this dissertation, we propose the use of exact inference procedures such as variable elimination [Zhang and Poole, 1994] and the junction tree algorithm [Pearl, 1988], Benjelloun et al. [2006]; Fuhr and Rolleke [1997] have utilized the inclusion-exclusion principle for boolean formulas, Re et al. [2007] proposed the use of a Markov chain Monte Carlo technique, Koch and Olteanu [2008] use ordered binary decision diagrams and as mentioned earlier, Wang et al. [2008] makes direct use of existing work on lifted inference [Poole, 2003] developed in the SRL community.

Other techniques to improve efficiency of evaluating queries in probabilistic databases include Trio's memoization techniques [Das Sarma et al., 2008], index structures for uncertain data retrieval [Singh et al., 2007] and index structures for junction trees [Kanagal and Deshpande, 2009]. These techniques are fairly generic and can be used in conjunction with the techniques proposed in this dissertation.

2.5 Conclusion

Having surveyed the relevant related work, we are now ready to proceed with the rest of the dissertation. In the next chapter we propose models for representing uncertainty to be used in conjunction with probabilistic databases, develop a technique that expresses a query evaluation problem as an inference problem on a PGM and illustrate the connection between query evaluation and probabilistic inference.

Chapter 3

Representing Uncertain Data

The database community has seen a lot of work on managing uncertain data, and the search for an ideal representation scheme has been a topic of constant interest. In the past, a number of different approaches have been proposed to represent uncertainty (we surveyed some of these in Chapter 2). Among these, perhaps the most frequently proposed approach has been the use of probability theory, perhaps due to its balance between power and simplicity; probability theory is general enough to represent most kinds of uncertainty we encounter in various applications in practice and is still simple enough to be amenable to algebraic manipulation so that we can use it to perform various operations such as query evaluation.

In this chapter, we describe our approach to representing uncertainty in databases. We use probability theory in conjunction with probabilistic graphical models (PGMs) to develop a compact scheme to represent uncertain data with correlations. In the next section, we provide background on PGMs. In Section 3.2, we formally define a probabilistic database in terms of PGMs and describe their semantics, in addition to providing a few examples that illustrate how correlations can be represented and affect the distribution represented by a probabilistic database. In Section 3.3, we discuss query evaluation and optimizations that can lead to efficient query evaluation, especially for aggregate computation. We conclude the chapter with Section 3.5, after describing experimental results in Section 3.4.

3.1 Background: Probabilistic Graphical Models

Probabilistic graphical models (PGMs) form a powerful class of approaches that can compactly represent and reason about complex dependency pat-

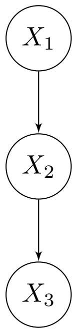
$$Pr(x_1, x_2, x_3) = \frac{1}{\mathcal{Z}} f_1(x_1) f_{12}(x_1, x_2) f_{23}(x_2, x_3)$$

x_1	f_1
1	0.5
2	0.3
3	0.2

x_1	x_2	f_{12}
1	1	0.8
1	2	0.1
1	3	0.1
2	1	0.1
2	2	0.8
2	3	0.1
3	1	0.1
3	2	0.1
3	3	0.8

x_2	x_3	f_{23}
1	1	0.7
1	2	0.2
1	3	0.1
2	1	0.2
2	2	0.7
2	3	0.1
3	1	0.2
3	2	0.1
3	3	0.7

(a)



(b)

x_1	x_2	x_3	Pr
1	1	1	0.280
1	1	2	0.080
1	1	3	0.040
1	2	1	0.010
1	2	2	0.035
1	2	3	0.005
1	3	1	0.010
1	3	2	0.005
1	3	3	0.035

x_1	x_2	x_3	Pr
2	1	1	0.021
2	1	2	0.006
2	1	3	0.003
2	2	1	0.048
2	2	2	0.168
2	2	3	0.024
2	3	1	0.006
2	3	2	0.003
2	3	3	0.021

x_1	x_2	x_3	Pr
3	1	1	0.014
3	1	2	0.004
3	1	3	0.002
3	2	1	0.004
3	2	2	0.014
3	2	3	0.002
3	3	1	0.032
3	3	2	0.016
3	3	3	0.112

(c)

Figure 3.1: Example involving three dependent random variables each with a ternary domain: (a) factored representation (b) graphical model representation (c) resulting joint probability distribution.

terns involving large numbers of correlated random variables [Cowell et al., 1999; Pearl, 1988]. The key idea behind PGMs is exploiting *conditional independence* [Pearl, 1988]. Most random variables only show local interactions or correlations with other random variables, and in many cases there are only a few of such correlations that need to be captured to represent the joint probability distribution defined over the collection of random variables. PGMs allow the specification of such correlations by defining small functions we refer to as *factors*[*], the joint probability distribution over the

[*] Factors are a generalization of *conditional probability tables* in Bayesian networks [Pearl,

collection of random variables can then be defined as a normalized product of all factors.

Let X denote a random variable with a domain $dom(X)$ and let $Pr(X)$ denote a probability distribution over it. Similarly, let $\mathbf{X} = \{X_1, X_2, X_3 \ldots, X_n\}$ denote a set of n random variables each with its own associated domain $dom(X_i)$, and $Pr(\mathbf{X})$ denote the joint probability distribution over them.

Definition 1. *A factor $f(\mathbf{X})$ is a function over a (small) set of random variables $\mathbf{X} = \{X_1, \ldots, X_n\}$ such that $0 \leq f(\mathbf{x}),\ \forall \mathbf{x} \in dom(X_1) \times \ldots \times dom(X_n)$.*

Definition 2. *A* probabilistic graphical model *(PGM) $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$ defines a joint distribution over the set of random variables $\mathcal{X}$ via a set of factors $\mathcal{F}$, where $\forall f(\mathbf{X}) \in \mathcal{F},\ \mathbf{X} \subseteq \mathcal{X}$. Given a complete joint assignment $\mathbf{x} \in \times_{X \in \mathcal{X}} dom(X)$, the joint distribution is defined by $Pr(\mathbf{x}) = \frac{1}{\mathcal{Z}} \prod_{f \in \mathcal{F}} f(\mathbf{x}_f)$ where $\mathbf{x}_f$ denotes the assignments restricted to arguments of f and $\mathcal{Z} = \sum_{\mathbf{x}'} \prod_{f \in \mathcal{F}} f(\mathbf{x}'_f)$*[†]*.*

Figure 3.1 shows a small example of a PGM expressing a joint probability distribution over three random variables each with domain {1,2,3}. The complete joint distribution is shown in Figure 3.1(c); note that representing this requires storing 27 real numbers (26, if you exploit the fact that the distribution should add upto 1). However, if we are willing to exploit conditional independence among X_1, X_2 and X_3, then we can represent the joint probability distribution with far fewer numbers. For instance, the distribution is such that X_3 is conditionally independent of X_1 given the value of X_2; in terms of correlations, X_1 only directly affects X_2's value and X_2 only affects X_3's values. Exploiting these properties, we can represent the same distribution using three factors (shown in Figure 3.1(a)). Note that the factors only require storing 21 real numbers which is 5 less compared to storing the joint distribution described earlier. The savings usually increase with more random variables and larger domains. In Figure 3.1(b) we show a "graphical" representation of the PGM where vertices represent random variables and edges depict correlations.

1988].

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

3.2 Probabilistic Databases with Probabilistic Graphical Models

We are now ready to define a probabilistic database in terms of a PGM. The basic idea is to use random variables to depict uncertain attribute values and factors to represent correlations. Let R denote a probabilistic relation or simply, relation, and let $attr(R)$ denote the set of attributes of R. A relation R consists of a set of probabilistic tuples or simply, tuples, each of which is a mapping from $attr(R)$ to random variables. Let $t.a$ denote the random variable of tuple $t \in R$ such that $a \in attr(R)$. Besides mapping each attribute to a random variable, every tuple t is also associated with a boolean-valued random variable which captures the existence uncertainty of t and we denote this by $t.e$.

Definition 3. *A* probabilistic database *or simply, a* database, *$\mathcal{D}$ is a pair $\langle \mathcal{R}, \mathcal{P} \rangle$ where $\mathcal{R}$ is a set of relations and $\mathcal{P}$ denotes a PGM defined over the set of random variables associated with the tuples in $\mathcal{R}$.*

3.2.1 Possible World Semantics

We now define semantics for our formulation of a probabilistic database. Let $\mathcal{X}$ denote the set of random variables associated with database $\mathcal{D} = \langle \mathcal{R}, \mathcal{P} \rangle$. Possible world semantics defines a database $\mathcal{D}$ as a probability distribution over deterministic databases or possible worlds [Dalvi and Suciu, 2004] each of which is obtained by assigning $\mathcal{X}$ a joint assignment $\mathbf{x} \in \times_{X \in \mathcal{X}} dom(X)$[‡]. The probability associated with the possible world obtained from the joint assignment $\mathbf{x}$ is given by the distribution defined by the PGM $\mathcal{P}$ (Definition 2).

3.2.2 Examples

We now present a few examples to further explain our notion of a probabilistic database. Consider the two-relation database shown in Figure 3.2(a). In this database, every tuple has an uncertain attribute value (the

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

S	**A**	**B**
s_1	a_1	{1: 0.6, 2: 0.4}
s_2	a_2	{1: 0.6, 2: 0.4}

T	**B**	**C**
t_1	{2: 0.5, 3: 0.5}	c

(a)

$s_1.\mathbf{B}$	$f_{s_1.\mathbf{B}}$
1	0.6
2	0.4

$s_2.\mathbf{B}$	$f_{s_2.\mathbf{B}}$
1	0.6
2	0.4

$t_1.\mathbf{B}$	$f_{t_1.\mathbf{B}}$
2	0.5
3	0.5

(b)

Figure 3.2: A small database with independent uncertain attribute values.

B attributes) and these are indicated in Figure 3.2(a) by specifying their respective domains with each entry from the domain followed by the probability with which the attribute value can take the assignment. In a database, we represent the uncertainty associated with each uncertain value using a random variable and the corresponding probability distribution using a factor (assuming complete independence). For instance, $s_2.\mathbf{B}$ can be assigned the value 1 with probability 0.6 and the value 2 with probability 0.4 and we would represent this using the factor $f_{s_2.\mathbf{B}}$ shown in Figure 3.2(b). We show all three required factors $f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$, $f_{s_2.\mathbf{B}}(s_2.\mathbf{B})$ and $f_{t_1.\mathbf{B}}(t_1.\mathbf{B})$ in Figure 3.2(b). In addition to the random variables which denote uncertain attribute values, we can introduce tuple *existence* random variables $s_1.e$, $s_2.e$, and $t_1.e$, which capture tuple uncertainty. These are boolean-valued random variables and can have associated factors. In Figure 3.2, we assume the tuples are certain, so we do not show the existence random variables for the base tuples. We next explain semantics of our example database in terms of possible worlds.

The database shown in Figure 3.2 represents a distribution over many deterministic databases (possible worlds), and each possible world is obtained by assigning all three random variables $s_1.\mathbf{B}$, $s_2.\mathbf{B}$ and $t_1.\mathbf{B}$ assignments from their respective domains. Since the three random variables depicted in Figure 3.2 each have domain with size 2, there are $2^3 = 8$ possible worlds. Figure 3.3 shows all 8 possible worlds with the corresponding probabilities listed under the column "prob.(ind.)". 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 $s_1.\mathbf{B} = 1$, $s_2.\mathbf{B} = 2$, $t_1.\mathbf{B} = 2$ (D_3 in Figure 3.3) the probability is $0.6 \times 0.4 \times 0.5 = 0.12$.

possible world	prob. (ind.)	prob. (implies)	prob. (diff.)	prob. (pos.corr.)
$D_1 = \begin{array}{ccc} S & \textbf{A} & \textbf{B} \\ s_1 & a_1 & 1 \\ s_2 & a_2 & 1 \end{array} \quad \begin{array}{ccc} T & \textbf{B} & \textbf{C} \\ t_1 & 2 & c \end{array}$	0.18	0.50	0.30	0.06
$D_2 = \begin{array}{ccc} S & \textbf{A} & \textbf{B} \\ s_1 & a_1 & 1 \\ s_2 & a_2 & 1 \end{array} \quad \begin{array}{ccc} T & \textbf{B} & \textbf{C} \\ t_1 & 3 & c \end{array}$	0.18	0.02	0.06	0.30
$D_3 = \begin{array}{ccc} S & \textbf{A} & \textbf{B} \\ s_1 & a_1 & 1 \\ s_2 & a_2 & 2 \end{array} \quad \begin{array}{ccc} T & \textbf{B} & \textbf{C} \\ t_1 & 2 & c \end{array}$	0.12	0	0.20	0.04
$D_4 = \begin{array}{ccc} S & \textbf{A} & \textbf{B} \\ s_1 & a_1 & 1 \\ s_2 & a_2 & 2 \end{array} \quad \begin{array}{ccc} T & \textbf{B} & \textbf{C} \\ t_1 & 3 & c \end{array}$	0.12	0.08	0.04	0.20
$D_5 = \begin{array}{ccc} S & \textbf{A} & \textbf{B} \\ s_1 & a_1 & 2 \\ s_2 & a_2 & 1 \end{array} \quad \begin{array}{ccc} T & \textbf{B} & \textbf{C} \\ t_1 & 2 & c \end{array}$	0.12	0	0	0.24
$D_6 = \begin{array}{ccc} S & \textbf{A} & \textbf{B} \\ s_1 & a_1 & 2 \\ s_2 & a_2 & 1 \end{array} \quad \begin{array}{ccc} T & \textbf{B} & \textbf{C} \\ t_1 & 3 & c \end{array}$	0.12	0.08	0.24	0
$D_7 = \begin{array}{ccc} S & \textbf{A} & \textbf{B} \\ s_1 & a_1 & 2 \\ s_2 & a_2 & 2 \end{array} \quad \begin{array}{ccc} T & \textbf{B} & \textbf{C} \\ t_1 & 2 & c \end{array}$	0.08	0	0	0.16
$D_8 = \begin{array}{ccc} S & \textbf{A} & \textbf{B} \\ s_1 & a_1 & 2 \\ s_2 & a_2 & 2 \end{array} \quad \begin{array}{ccc} T & \textbf{B} & \textbf{C} \\ t_1 & 3 & c \end{array}$	0.08	0.32	0.16	0

Figure 3.3: Possible worlds for example in Figure 3.2(a).

$$Pr^{implies}(s_1.\mathbf{B}, s_2.\mathbf{B}, t_1.\mathbf{B}) = f^{implies}_{t_1.\mathbf{B}}(t_1.\mathbf{B}) f^{implies}_{t_1.\mathbf{B},s_1.\mathbf{B}}(t_1.\mathbf{B}, s_1.\mathbf{B}) f^{implies}_{t_1.\mathbf{B},s_2.\mathbf{B}}(t_1.\mathbf{B}, s_2.\mathbf{B})$$

$t_1.\mathbf{B}$	$f^{implies}_{t_1.\mathbf{B}}$
2	0.5
3	0.5

$t_1.\mathbf{B}$	$s_1.\mathbf{B}$	$f^{implies}_{t_1.\mathbf{B},s_1.\mathbf{B}}$
2	1	1
2	2	0
3	1	0.2
3	2	0.8

$t_1.\mathbf{B}$	$s_2.\mathbf{B}$	$f^{implies}_{t_1.\mathbf{B},s_2.\mathbf{B}}$
2	1	1
2	2	0
3	1	0.2
3	2	0.8

(a)

$$Pr^{diff}(s_1.\mathbf{B}, s_2.\mathbf{B}, t_1.\mathbf{B}) = f^{diff}_{t_1.\mathbf{B},s_1.\mathbf{B}}(t_1.\mathbf{B}, s_1.\mathbf{B}) f^{diff}_{s_2.\mathbf{B}}(s_2.\mathbf{B})$$

$t_1.\mathbf{B}$	$s_1.\mathbf{B}$	$f^{diff}_{t_1.\mathbf{B},s_1.\mathbf{B}}$
2	1	0.5
2	2	0
3	1	0.1
3	2	0.4

$s_2.\mathbf{B}$	$f^{diff}_{s_2.\mathbf{B}}$
1	0.6
2	0.4

(b)

$$Pr^{pos.corr}(s_1.\mathbf{B}, s_2.\mathbf{B}, t_1.\mathbf{B}) = f^{pos.corr.}_{t_1.\mathbf{B},s_1.\mathbf{B}}(t_1.\mathbf{B}, s_1.\mathbf{B}) f^{pos.corr.}_{s_2.\mathbf{B}}(s_2.\mathbf{B})$$

$t_1.\mathbf{B}$	$s_1.\mathbf{B}$	$f^{pos.corr.}_{t_1.\mathbf{B},s_1.\mathbf{B}}$
2	1	0.1
2	2	0.4
3	1	0.5
3	2	0

$s_2.\mathbf{B}$	$f^{pos.corr.}_{s_2.\mathbf{B}}$
1	0.6
2	0.4

(c)

Figure 3.4: Factors for the probabilistic databases with dependencies (we have omitted the normalization constant $\mathcal{Z}$ because the numbers are such that distribution is already normalized) (a) *implies* correlation (b) *different* correlation (c) *positive* correlation.

Let us now try to modify our example to illustrate how to represent correlations in a probabilistic database. In particular, we will try to construct three different databases each containing the following dependencies respectively:

- *implies*: $t_1.\mathbf{B} = 2$ implies $s_1.\mathbf{B} \neq 2$ and $s_2.\mathbf{B} \neq 2$, in other words, $(t_1.\mathbf{B} = 2) \implies (s_1.\mathbf{B} = 1) \wedge (s_2.\mathbf{B} = 1)$.
- *different*: $t_1\mathbf{B}$ and $s_1.\mathbf{B}$ cannot have the same assignment, in other words, $(t_1.\mathbf{B} = 2) \Leftrightarrow (s_1.\mathbf{B} = 1)$ or $(s_1.\mathbf{B} = 2) \Leftrightarrow (t_1.\mathbf{B} = 3)$.
- *positive correlation*: High positive correlation between $t_1.\mathbf{B}$ and $s_1.\mathbf{B}$, if one is assigned 2 then the other is also assigned the same value with high probability.

Figure 3.3 shows a distribution each over the possible worlds that satisfies each of the above correlations (the columns are labeled with abbreviations of the names of the correlations, e.g., the column for positive correlation is labeled "pos. corr.").

To represent the possible worlds of our example database with the new correlations, we simply redefine the factors in the database. However, in this case, since we need to represent correlations, we will need to use factors defined over multiple random variables. Figure 3.4 represents the three sets of factors each corresponding to a database with each of the previously defined dependencies that depict the required distribution over possible worlds from Figure 3.3. For instance, Figure 3.4 (a) shows the factors required to define the possible worlds distribution depicted in column "implies" in Figure 3.3, and this is achieved by defining factors $f^{implies}_{t_1.\mathbf{B},s_1.\mathbf{B}}$ and $f^{implies}_{t_1.\mathbf{B},s_2.\mathbf{B}}$ which denote the implication dependencies defined earlier. Similarly, notice how factor $f^{diff}_{t_1.\mathbf{B},s_1.\mathbf{B}}$ (Figure 3.4 (b)) enforces that $t_1.\mathbf{B}$ and $s_1.\mathbf{B}$ be assigned different values. Lastly, $f^{pos.corr.}_{t_1.\mathbf{B},s_1.\mathbf{B}}$ enforces the positive correlation between $t_1.\mathbf{B}$ and $s_1.\mathbf{B}$ depicted in the third example.

Note that in Definition 3, we make no restrictions as to which random variables appear as arguments in a factor. Thus, if the user wishes, s/he may define a factor including random variables from the same tuple, different tuples, tuples from different relations or tuple existence and attribute value random variables, which means that in our formulation we can express any kind of correlation that one might think of representing in a probabilistic database.

3.3 Query Evaluation

Having defined our representation scheme, we now move our discussion to query evaluation. The main 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) and a database $\mathcal{D}$, then the result of evaluating q against $\mathcal{D}$ is defined to be the set of results obtained by evaluating q against each possible world of $\mathcal{D}$ augmented with the probabilities of the possible worlds. Relating back to our earlier examples, suppose we want to run the query $q = \prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$. Figure 3.5(a) shows the set of results obtained from each set of possible worlds augmented by the corresponding probabilities depending on which database we ran the query against.

Now, even though query evaluation under possible world semantics is clear and intuitive, it still has some issues that prevent us from executing it directly. First and foremost among these issues, is the size of the result. Since the number of possible worlds is exponential in the number of random variables in the database (product of domain sizes of all random variables to be more precise), in the case that every possible world returns a different result, returning the result to the user or storing it is only going to be feasible for the smallest of databases. To get around this issue, it is traditional to compress the result before returning it to the user. One way of doing this is to collect all tuples from the set of results returned by possible world semantics and return these along with the sum of probabilities of the possible worlds that return the tuple as a result [Dalvi and Suciu, 2004]. In Figure 3.5(a), there is only one tuple that is returned as a result and this tuple is returned by possible worlds D_3, D_5 and D_7. In Figure 3.5(b), we show the resulting probabilities obtained by summing across these three possible worlds for each example database.

The second issue is, of course, related to the complexity of computing the results of a query from first principles. Since the number of possible worlds is going to be large for any non-trivial database, evaluating results directly by enumerating all of its possible worlds is going to be infeasible. To get around this issue we first make the connection between computing query results for a probabilistic database and the marginal probability computation problem for probabilistic graphical models.

Definition 4. *Given a PGM $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$ and a random variable $X \in \mathcal{X}$, the* marginal probability *associated with the assignment $X = x$, where $x \in dom(X)$, is defined as $\mu_X(x) = \sum_{\mathbf{x} \sim x} Pr(\mathbf{x})$, where $Pr(\mathbf{x})$ denotes the distribution defined by the PGM and $\mathbf{x} \sim x$ denotes a joint assignment to $\mathcal{X}$ where X is assigned x.*

Since each possible world is obtained by assigning all random variables

possible world	query result	prob. (ind.)	prob. (implies)	prob. (diff.)	prob. (pos.corr.)
D_1	$\emptyset$	0.18	0.50	0.30	0.06
D_2	$\emptyset$	0.18	0.02	0.06	0.30
D_3	**C** c	0.12	0	0.20	0.04
D_4	$\emptyset$	0.12	0.08	0.04	0.20
D_5	**C** c	0.12	0	0	0.24
D_6	$\emptyset$	0.12	0.08	0.24	0
D_7	**C** c	0.08	0	0	0.16
D_8	$\emptyset$	0.08	0.32	0.16	0

(a)

query result	$Pr(D_3) + Pr(D_5) + Pr(D_7)$			
	ind.	implies	diff.	pos.corr.
C c	0.32	0	0.20	0.40

(b)

Figure 3.5: Results running the query $\prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$ on the different example databases.

in the database with a joint assignment, at least intuitively, it does seem like we are computing marginal probabilities when we sum over all possible worlds to evaluate a query. However, we have yet to express the result tuples using random variables (the random variables in the database are the ones associated with the base tuples). Therefore, to cast the query evaluation problem into a marginal probability computation problem, we have to first show how to augment the PGM underlying the database such that the augmented PGM contains random variables representing result tuples. We can then express the probability computation associated with evaluating the query as a standard marginal probability computation problem and thus allow us to use any of the host of probabilistic inference algorithms designed to perform marginal probability computations to solve the query evaluation problem. We next present an example to illustrate the basic ideas underlying our approach to augmenting the PGM underlying the database given a query; after that we discuss how to augment the PGM in the general case given any relational algebra query.

3.3.1 Example

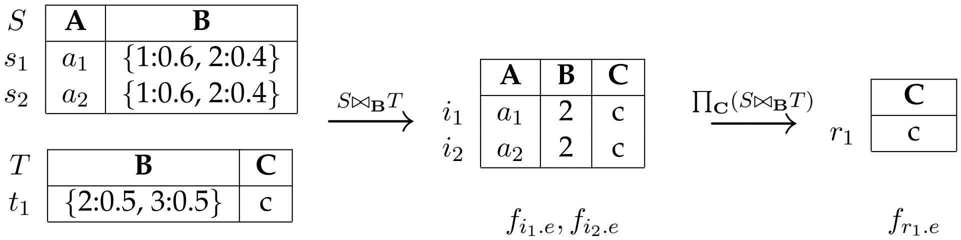
Consider running the query $\prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$ on the database presented in Figure 3.2(a). Our query evaluation approach is very similar to query evaluation in traditional database systems and is depicted in Figure 3.6. Just as in traditional database query processing, in Figure 3.6, we introduce intermediates tuples produced by the join (i_1 and i_2) and produce a result tuple (r_1) produced 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 that we need to maintain:

- 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 $s_1.\mathbf{B}$ and $t_1.\mathbf{B}$ are assigned the value 2.

- Similarly, $i_2.e$ is `true` only in those possible worlds where both $s_2.\mathbf{B}$ and $t_1.\mathbf{B}$ are assigned the value 2.

- Finally, r_1 (the result tuple produced by the projection) exists or $r_1.e$ is `true`, only in those possible worlds that produce at least one of i_1 or i_2 or both.

To enforce these correlations, during query evaluation we introduce intermediate factors defined over appropriate random variables. For our example, we introduce the following three correlations:



S	**A**	**B**
s_1	a_1	{1:0.6, 2:0.4}
s_2	a_2	{1:0.6, 2:0.4}

T	**B**	**C**
t_1	{2:0.5, 3:0.5}	c

	A	**B**	**C**
i_1	a_1	2	c
i_2	a_2	2	c

	C
r_1	c

$S \bowtie_{\mathbf{B}} T$
$\prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$
$f_{i_1.e}, f_{i_2.e}$
$f_{r_1.e}$

Figure 3.6: Evaluating $\prod_{\mathbf{C}}(S \bowtie_{\mathbf{B}} T)$ on the database from Figure 3.2(a).

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

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

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

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

- For the correlation among $r_1.e$, $i_1.e$ and $i_2.e$, we introduce a factor $f_{r_1.e}$ capturing the `or` semantics. In other words, we would like to enforce that $r_1.e$ is `true` when at least one of $i_1.e$ *or* $i_2.e$ hold true:

$$f_{r_1.e}(r_1.e, i_1.e, i_2.e) = \begin{cases} 1 & \text{if } r_1.e \Leftrightarrow (i_1.e \vee i_2.e) \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.6 depicts the full run of the query along with the introduced factors.

Now, to compute the probability of existence of r_1 (which is what we did in Figure 3.5 by enumerating over all possible worlds), we simply need to compute the marginal probability associated with the assignment $r_1.e$ = `true` from PGM formed by the set of factors in the base data and the factors introduced during query evaluation. For instance, for the example where we assumed complete independence among all uncertain attribute values (Figure 3.2(b)), our augmented PGM is given by the collection $f_{s_1.\mathbf{B}}, f_{s_2.\mathbf{B}}, f_{t_1.\mathbf{B}}, f_{i_1.e}, f_{i_2.e}$ and $f_{r_1.e}$, and to compute the marginal probability, we can simply use any of the exact inference algorithms available in the machine learning literature such as variable elimination [Dechter, 1996; Zhang and Poole, 1994] or the junction tree algorithm [Huang and Darwiche, 1994].

3.3.2 General Relational Algebra Queries

Query evaluation for general relational algebra 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 result tuples of interest. We next describe the modified σ, $\times$, $\prod$, δ, $\cup$, $-$ 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, besides being 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 is $f^{\sigma}_{r.e}$, which 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 $c(\{t.a\}_{a\in attrR})$ is `true` if a joint assignment to the attribute value random variables of t satisfies the predicate c and `false` otherwise.

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, besides being 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, besides being associated with $r.e$. The factor for $r.e$, denoted by $f^{\prod}_{r.e}$, 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 `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 $dom(y^r_t) = \{$`true`, `false`$\}$ such that y^r_t is `true` iff t gives rise to r and `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:

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

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

Aggregation operators: Aggregation operators are also easily handled using factors. Suppose we want to compute the `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 $\{t.a\}_{t \in attr(R)}$ 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:

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

Optimizations: For the above operator modifications, we have attempted to be completely general and as such, the factors introduced may look slightly 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 which 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 the 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} which is inefficient. By exploiting decomposability of $\vee$ we can implement the same factor using a linear number of constant sized (3-argument) factors which may lead to significant speedups. We refer the interested reader to Rish [1999]; Zhang and Poole [1996] for more details. The only aggregation operator that is not decomposable is `avg`, but even in this case we can exploit the same ideas by implementing `avg` in terms of `sum` and `count`, both of which are decomposable.

3.3.3 Complexity of probabilistic inference

The above operators will help generate the augmented PGM given any relational algebra query to be executed on a database, after generating the augmented PGM, the last step of query evaluation requires that we run probabilistic inference. Exact probabilistic inference is known to be NP-hard in general [Cooper, 1990]. More specifically, the complexity of exact probabilistic inference is exponential in a quantity known as the *treewidth* [Arnborg, 1985] which depends on the structure of the graph depicting the PGM (where vertices denote random variables and edges denote correlations, see Figure 3.1(b) for an example). However, many applications provide PGMs with sparse graph structures that allow efficient probabilistic computation [Zhang and Poole, 1994]. Variable elimination, also known as *bucket elimination*, [Dechter, 1996; Zhang and Poole, 1994] and the junction tree algorithm [Huang and Darwiche, 1994] are two exact inference algorithms (among others) that have the ability to exploit such structure. In particular, the inference problem is easy if the PGM is or closely resembles a tree and the problem becomes progressively harder as the PGM deviates more from being a tree.

3.4 Experiments and Discussion

We performed three sets of experiments. In the first set of experiments, we illustrate the importance of modeling correlations. Here, we chose the problem of querying publication datasets as a case study and show that if we do not model natural mutual exclusivity correlations then results can be counter-intuitive. In the second set of experiments, we experiment on the TPC-H benchmark [TPC-H Benchmark] with slight modifications (to add probabilities) to demonstrate the scalability of query evaluation with probabilistic inference. In the third set of experiments, we demonstrate the range of queries that can be evaluated with probabilistic inference by evaluating aggregation queries.

3.4.1 Case Study: Querying Publication Datasets

Most applications produce data that requires modeling uncertainty and usually, assuming one wants to be faithful to the underlying distribution then, complex correlations need to be represented. However, the complexity of managing uncertain databases increases with increasingly correlated models. In this section, we present some experiments that indicate the need for modeling correlations, that unless we do this the quality of results obtained from query evaluation can be exceedingly poor.

Consider a *publications* database containing two relations: (1) `PUBS(PID, Title)`, and (2) `AUTHS(PID, Name)`, where PID is the unique publication id, and consider the task of retrieving all publications with title y written by an author with name x. Assuming that the user is not sure of the spellings x and y, we might use the following query to perform the above task:

$$\prod_{Title}(\sigma_{Name\approx x}(\mathtt{AUTHS}) \bowtie \sigma_{Title\approx y}(\mathtt{PUBS}))$$

One way to handle uncertain predicates used above is to interpret them in terms of probabilities. Given a predicate of the form $R.a \approx k$, where a is a string attribute, and k is a string constant, the system assigns a probability to each tuple t, based on how *similar* $t.a$ is to k. Following Dalvi and Suciu [2004], we compute the *3-gram distance* [Ukkonen, 1992] between $t.a$ and k, and convert it to a posterior probability by assuming that the distance is normally distributed with mean 0, and variance σ (σ is a parameter fed to the system). For the above query, the similarity predicates will cause both the relations `PUBS` and `AUTHS` to be converted into probabilistic relations, $\mathtt{AUTHS}^p$ and $\mathtt{PUBS}^p$. However, note that $\mathtt{AUTHS}^p$ contains natural mutual exclusion dependencies with respect to this query. Since the user is looking for publications by a single author with name x, it is *not possible for x to match two* AUTHS^p *tuples corresponding to the same publication in the same possible world*. Thus, any two AUTHS^p tuples with the same PID exhibit a mutual exclusion dependency, and a possible world containing both of them should be assigned zero probability.

To illustrate the drawbacks of ignoring such mutual exclusion dependencies, we ran the above query with x = "T. Michel" and y = "Reinforment Leaning hiden stat" on two probabilistic databases, one assuming complete independence among tuples (*IND_DB*) and another that models the dependencies (*MUTEX_DB*). We ran the query on an extraction of 860 publications from the real-world CiteSeer dataset [Giles et al., 1998b]. We report results across various settings of σ. Figure 3.7 shows the top three results obtained from the two databases at three different settings of σ (we also list the author names to aid the reader's understanding). *MUTEX_DB* returns intuitive and similar results at all three values of σ. *IND_DB* returns reasonable results only at $\sigma = 10$, whereas at $\sigma = 50, 100$ it returns very odd results ("Decision making and problem solving" does not match the string "Reinforment Leaning hiden stat" very closely and yet it is assigned the highest rank at $\sigma = 100$). Figure 3.8 (i) shows the cumulative recall graph for *IND_DB* for various values of σ, where we plot the fraction of the top N results returned by *MUTEX_DB* that were present in the top N results returned by *IND_DB*. As we can see, at $\sigma = 50$ and 100, *IND_DB* exhibits poor recall.

Figure 3.7 shows that *IND_DB* favors publications with long author lists. This does not affect the results at low values of σ (=10) because, in that case, we use a "peaked" gaussian which assigns negligible probabilities to possible worlds with multiple AUTHS^p from the same publication. At larger settings of σ, however, these possible worlds are assigned larger probabilities and *IND_DB* returns poor results. *MUTEX_DB* assigns these possible worlds zero probabilities by modeling dependencies on the base tuples. We would like to note that, although setting the value of σ carefully may have resulted in a good answer for *IND_DB* in this case, choosing σ is not easy in general and depends on various factors such as user preferences, distributions of the attributes in the database, etc. Modeling mutual exclusion dependencies explicitly using our approach naturally alleviates this problem.

Title
Reinforcement learning with hidden states (by L. Lin, T. Mitchell)
Feudal Reinforcement Learning (by C. Atkeson, P. Dayan, . . .)
Reasoning (by C. Bereiter, M. Scardamalia)
. . .

(i) *MUTEX_DB* results at $\sigma = 10, 50, 100$

Title
Reinforcement learning with hidden states (by L. Lin, T. Mitchell)
Feudal Reinforcement Learning (by C. Atkeson, P. Dayan, . . .)
Reasoning (by C. Bereiter, M. Scardamalia)
. . .

(ii) *IND_DB* results at $\sigma = 10$

Title
Feudal Reinforcement Learning (by C. Atkeson, P. Dayan, . . .)
Decision making and problem solving (G. Dantzig, R. Hogarth, . . .)
Multimodal Learning Interfaces (by U. Bub, R. Houghton, . . .)
. . .

(iii) *IND_DB* results at $\sigma = 50$

Title
Decision making and problem solving (G. Dantzig, R. Hogarth, . . .)
HERMES: A heterogeneous reasoning and mediator system (by S. Adali, A. Brink, . . .)
Induction and reasoning from cases (by K. Althoff, E. Auriol, . . .)
. . .

(iv) *IND_DB* results at $\sigma = 100$

Figure 3.7: Top three results for a similarity query: (i) shows results from *MUTEX_DB*; (ii), (iii) and (iv) show results from *IND_DB*.

3.4.2 Experiments with TPC-H Benchmark

We also show scalability results for our proposed query execution strategies using a randomly generated TPC-H dataset of size 10MB. For simplicity, we assume complete independence among the base tuples (though the intermediate tuples may still be correlated). Figure 3.8 (iii) shows the execution times on TPC-H queries Q2 to Q8 (modified to remove the top-level aggregations). The first bar on each query indicates the time it took for our implementation to run the full query including all the database operations and the probability computations. The second bar on each query indicates the time it took to run only the database operations using our JAVA-based implementation. Here are the summary of the results:

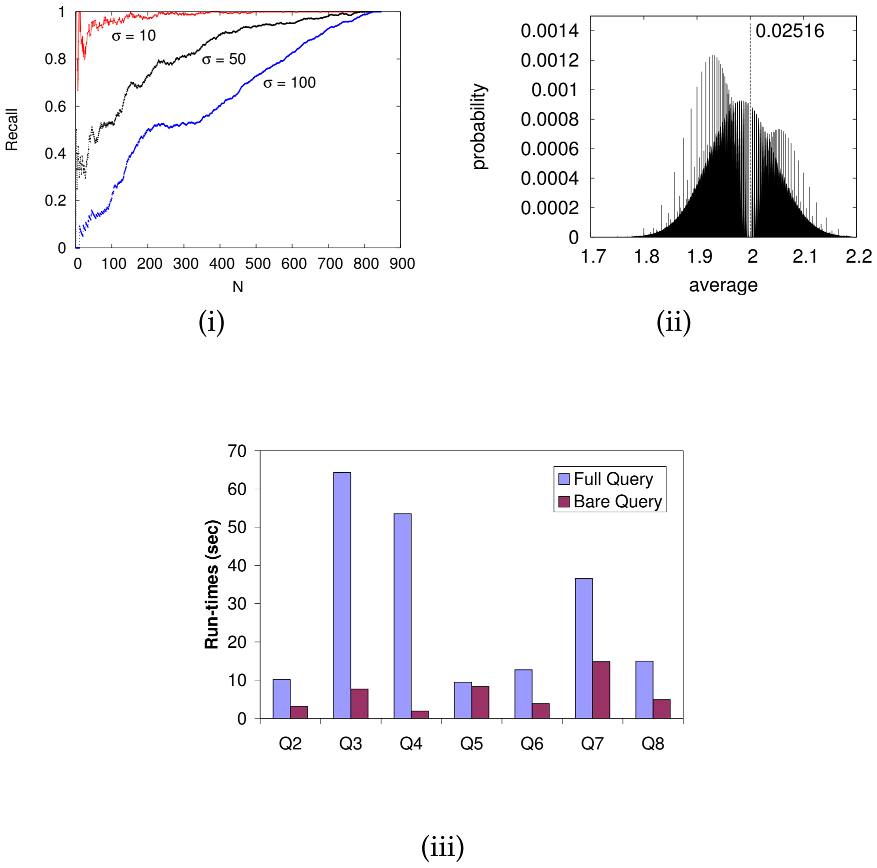
- As we can see in Figure 3.8 (iii), for most queries the additional cost of probability computations is comparable to the cost of normal query processing.

- The two exceptions are Q3 and Q4 which severely tested our probabilistic inference engine. By removing the aggregate operations, Q3 resulted in a relation of size in excess of 60,000 result tuples. Although Q4 resulted in a very small relation, each result tuple was associated with a probabilistic graphical model of size exceeding 15,000 random variables. Each of these graphical models are fairly sparse but bookkeeping for such large data structures took a significant amount of time.

- Q7 and Q8 are supposed to be intractable queries (i.e., are not *hierarchical queries* [Dalvi and Suciu, 2004]) yet their run-times are surprisingly fast. By taking a closer look, we noticed that both these queries gave rise to tree-structured graphical models for which treewidth is low justifying our belief that there are may be databases where the data allows query evaluation to be tractable even if query compilation techniques [Dalvi and Suciu, 2004; Olteanu and Huang, 2009, 2008] suggest otherwise.

3.4.3 Aggregation Queries

Our approach also naturally supports efficient computation of a variety of aggregate operators over probabilistic relations using the decomposition techniques described in Section 3.3. Figure 3.8 (ii) shows the result of running an *average* query over a synthetically generated dataset containing 500 tuples. As we can see, the final result can be a fairly complex probability distribution, which is quite common for aggregate operations.



(i) (ii)

(iii)

Figure 3.8: (i) Cumulative recall graph comparing results of *IND_DB* and *MUTEX_DB* for $\sigma = 10, 50, 100$. (ii) AVG aggregate computed over 500 randomly generated tuples with attribute values ranging from 1 to 5. (iii) Run-times on TPC-H data.

3.5 Conclusion

In this chapter, we described an approach to represent uncertain data with arbitrary correlations in a probabilistic database using probabilistic graphical models. Probabilistic graphical models allow us exploit conditional independence present in the data to provide a compact scheme that can represent both attribute and tuple level uncertainty in the same database. We showed how our representation scheme naturally lends itself to possible world semantics thus associating precise semantics with the query evaluation problem. We further showed that it is possible to recast the query evaluation problem into a marginal probability computation problem on an appropriately constructed probabilistic graphical model that can be generated on the fly. Our approach allows us to use a host of probabilistic inference algorithms (exact and approximate) developed in the machine learning community to evaluate queries, however there are certain aspects regarding query evaluation in probabilistic databases that make it unique and different from the inference problems traditionally considered in machine learning research. In the next chapter we show how these aspects can be exploited to speedup query evaluation for probabilistic databases.

Chapter 4

Bisimulation-based Lifted Inference for Probabilistic Databases

In the last chapter, we described our representation scheme for uncertain data and showed that query evaluation reduces to probabilistic inference in such databases. In reality, most probabilistic database formulations (whether the one described in the previous chapter or other formulations based on tuple-level uncertainty such as Benjelloun et al. [2006] or Re and Suciu [2007]) require general probabilistic inference at some level of abstraction. Thus it is imperative that we design efficient inference approaches to make probabilistic databases a feasible and viable option. Given that we already know general inference is a #P-complete problem [Dalvi and Suciu, 2004], the only way we can achieve this is to utilize the special properties of the data at hand. In this chapter, we motivate the presence of one such property that we refer to as *shared correlations*, and show how to exploit it to speed up inference during query evaluation for probabilistic databases.

Consider the example database containing pre-owned car sales ads from the last chapter which we used to contrast between tuple level uncertainty databases and databases that can express uncertainty at both tuple and attribute levels (shown again in Figure 4.1 for convenience). Recall that, the first tuple shows an ad with the color of the car missing, the third tuple shows one with the make missing and the second tuple represents an ad with both attributes missing. Figure 4.1 also shows the probability distributions associated with these missing values, more specifically, f_{make} defines the distribution over missing make values in the database (assuming our

AdID	Make	Color	Price
1	Honda	?	\$9,000
2	?	?	\$6,000
3	?	Beige	\$8,000
⋮	⋮	⋮	⋮

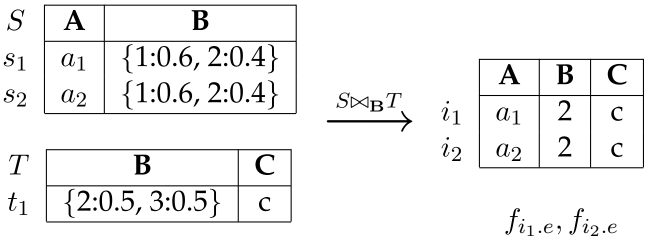
Color	f_{color}
Black	0.75
Beige	0.25

Make	f_{make}
Honda	0.55
Toyota	0.45

Figure 4.1: Pre-owned car ads with missing values.

universe can contain only two makes Honda and Toyota) and f_{color} defines the distribution over missing color values (assuming our universe contains only black and beige cars). Note that the distributions make no reference to any tuple specific information. In other words, no matter how many tuples with missing color are present in the relation, their uncertainty will still be defined by the same distribution represented by f_{color} and, along with f_{make}, these distributions are examples of *shared correlations* (more precisely defined in Section 4.2.

In many cases, the uncertainty in the data is defined using general statistics that *do not* vary on a per-tuple basis, and this, in turn, leads to shared correlations. Various earlier works have also described applications with shared correlations. For instance, Andritsos et al. [2006] describe a customer relationship management application where the objective is to merge data from two or more source databases and each source database is assigned a probability value based on the quality of the information it contains. Even here, probabilities do not change from tuple to tuple, since tuples from the same source are assigned the same source probability. Another source of shared correlations in probabilistic databases is the query evaluation approach itself. Recall from the previous chapter that while evaluating queries we first build an augmented PGM by introducing small factors that depict probability distributions and correlations on the fly. For instance, if tuples t_1 and t_2 join to produce join tuple r then one needs to introduce a factor that encodes the correlation that r exists iff both t_1 and t_2 exist ($f^{\times}_{r.e}(r.e, t_1.e, t_2.e)$ defined in the last chapter). More importantly, such a factor is introduced whenever *any* pair of tuples join, thus leading to repeated copies of the same factor, thus introducing additional shared correlations. Our aim, in this chapter, is to exploit such shared correlations to make exact probability computation for query evaluation in probabilistic



S	**A**	**B**
s_1	a_1	{1:0.6, 2:0.4}
s_2	a_2	{1:0.6, 2:0.4}

T	**B**	**C**
t_1	{2:0.5, 3:0.5}	c

$\xrightarrow{S \bowtie_{\mathbf{B}} T}$

	A	**B**	**C**
i_1	a_1	2	c
i_2	a_2	2	c

Figure 4.2: Running example for this chapter.

databases more efficient.

Our motivation for shared correlations and more efficient inference in this context closely ties in with recent work done in the machine learning community. In the past decade or so, machine learning researchers have devised approaches to exploit shared correlations to come up with more compact ways of describing PGMs. These models are sometimes referred to as *first-order graphical models*. Lifted inference is the sub-field that aims to devise more efficient inference techniques for first-order models that exploit such shared correlations. In fact, the inference approach we devise in this chapter is a novel lifted inference algorithm that automatically determines symmetries in the uncertainty model denoted by shared factors. We surveyed these related areas of research along with various works on lifted inference in Chapter 2. The rest of this chapter is organized as follows: In the next section, we introduce a motivating example that shows how standard inference algorithms fail to exploit shared correlations; in Section 4.2 and Section 4.3 we formally define shared correlations and describe our approach to inference with shared correlations; in Section 4.4 we describe our experimental results comparing our approach to standard inference algorithms; and finally, we conclude with Section 4.6 after a discussion in Section 4.5.

4.1 Motivating Example

For the purposes of this chapter, we will use a slightly simplified version of the example from the last chapter (Figure 4.2). In this modified version, we have the same two relations S and T, but we run a simple join query, $S \bowtie_{\mathbf{B}} T$, in this case. The inference task remains the same, i.e., we need to compute the marginal probabilities of the two result tuples produced. In

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

Figure 4.3: How variable elimination proceeds to solve the query evaluated in Figure 4.2.

other words, we need to compute marginal probabilities corresponding to the assignments $i_1.e$ = `true` and $i_2.e$ = `true` from the augmented PGM comprising of factors $f_{s_1.\mathbf{B}}, f_{s_2.\mathbf{B}}, f_{t_1.\mathbf{B}}, f_{i_1.e}$ and $f_{i_2.e}$ (see previous chapter for full definitions of the factors). Now let us try to see how standard inference algorithms such as variable elimination (VE) [Dechter, 1996; Zhang and Poole, 1994] and the junction tree algorithm [Huang and Darwiche, 1994] would proceed to solve this problem. Here we take the example of VE. Recall that marginal probability computation basically means that we simply sum over all the other random variables from the PGM except for the random variable whose marginal probability we need to compute (Definition 4). VE runs by first choosing an elimination order which specifies the order in which to sum over (eliminate) the random variables. It then repeatedly picks the next random variable from the order, pushes the corresponding summation as far into the product of factors as possible, sums it out and proceeds in this fashion. In Figure 4.3 we show the first few steps of how VE would proceed when used to compute the probability of $i_1.e$ and $i_2.e$ using the elimination order $\mathcal{O} = \{s_1.\mathbf{B}, s_2.\mathbf{B}, t_1.\mathbf{B}\}$ (variables are eliminated left to right).

4.1.1 Limitations of Naive Inference Algorithms

The main issue with VE (or any other standard exact probabilistic inference algorithm, for that matter) is that it does not exploit shared correlations. For instance, in Figure 4.3, in the process of computing the probabilities for $i_1.e$ and $i_2.e$, we produce intermediate factors $m_{s_1.\mathbf{B}}(i_1.e, t_1.\mathbf{B})$ and $m_{s_2.\mathbf{B}}(i_2.e, t_1.\mathbf{B})$. If we take a closer look at both of these factors then we will notice that they both map exactly the same inputs to the same outputs:

$i_1.e$	$t_1.\mathbf{B}$	$m_{s_1.\mathbf{B}}$
True	2	0.4
True	3	0
False	2	0.6
False	3	1

$i_2.e$	$t_1.\mathbf{B}$	$m_{s_2.\mathbf{B}}$
True	2	0.4
True	3	0
False	2	0.6
False	3	1

This indicates that we went through the exact same multiplication and summation steps to compute both $m_{s_1.\mathbf{B}}(i_1.e, t_1.\mathbf{B})$ and $m_{s_2.\mathbf{B}}(i_2.e, t_1.\mathbf{B})$. In fact, these are *shared factors* and represent *shared correlations* (which will be defined more precisely in the next section), and this repeated computation is what we would like to avoid. In hindsight, it is not really surprising that $m_{s_1.\mathbf{B}}(i_1.e, t_1.\mathbf{B})$ and $m_{s_2.\mathbf{B}}(i_2.e, t_1.\mathbf{B})$ turned out to be virtual copies of each other. If we look closely, $m_{s_1.\mathbf{B}}$ was computed by multiplying $f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$ with $f_{i_1.e}(i_1.e, s_1.\mathbf{B}, t_1.\mathbf{B})$ followed by a summation operation, whereas $m_{s_2.\mathbf{B}}$ was computed by multiplying $f_{s_2.\mathbf{B}}(s_2.\mathbf{B})$ with $f_{i_2.e}(i_2.e, s_2.\mathbf{B}, t_1.\mathbf{B})$ followed by a summation operation, and $f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$ and $f_{s_2.\mathbf{B}}(s_2.\mathbf{B})$, and $f_{i_1.e}(i_1.e, s_1.\mathbf{B}, t_1.\mathbf{B})$ and $f_{i_2.e}(i_2.e, s_2.\mathbf{B}, t_1.\mathbf{B})$ were pairs of shared factors. Often during the run of inference such intermediate shared factors multiply with each other and give rise to more intermediate shared factors thus making it imperative that we recognize and take advantage of such symmetry *before we actually compute these shared factors*. Devloping an approach that achieves this is the topic of the next section.

4.2 Inference with Shared Factors

We begin by formally defining shared factors, and for this we need to take a closer look at the definition of a factor (Definition 1). A factor consists of two distinct parts: the first part is the list of random variables it takes as arguments, and the second part is the function that maps input assignments to outputs. Thus, it may be possible for two factors f_1 and f_2 to have different arguments lists but use the same function to map inputs to outputs.

$$m_{s_1.\mathbf{B}} \qquad\qquad\qquad\qquad m_{s_2.\mathbf{B}}$$

$$\text{args.: } i_1.e, t_1.\mathbf{B} \qquad\qquad \text{args.: } i_2.e, t_1.\mathbf{B}$$

$$\text{func.: } \left\{ \begin{array}{r} \texttt{True}, 2 \rightarrow 0.4 \\ \texttt{True}, 3 \rightarrow 0 \\ \texttt{False}, 2 \rightarrow 0.6 \\ \texttt{False}, 3 \rightarrow 1 \end{array} \right. \qquad\qquad \text{func.: } \left\{ \begin{array}{r} \texttt{True}, 2 \rightarrow 0.4 \\ \texttt{True}, 3 \rightarrow 0 \\ \texttt{False}, 2 \rightarrow 0.6 \\ \texttt{False}, 3 \rightarrow 1 \end{array} \right.$$

Figure 4.4: Pair of shared factors.

Definition 5. *Let f_1 and f_2 denote two factors, $f_1.func$ and $f_2.func$ denote their function components, and dom$_1$ and dom$_2$ denote the domains of $f_1.func$ and $f_2.func$, respectively. f_1 and f_2 are* shared factors, *denoted $f_1 \cong f_2$, if dom$_1$ = dom$_2$ = dom and $f_1.func(d) = f_2.func(d), \forall d \in$ dom.*

Figure 4.4 shows two factors from the previous section where we have clearly separated their argument lists and function components.

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

The basic idea behind our approach to performing probabilistic inference with shared factors is to represent a run of the inference algorithm explicitly as a labeled graph. Once we do that, we will 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 will 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 [Zhang and Poole, 1994] and junction tree algorithm [Huang and Darwiche, 1994]).

4.2.1 The ELIMRV operator

The *elimrv* operator (which stands for ELIMinate a Random Variable) is the basic operator that is used repeatedly while running inference to compute marginal probabilities. It essentially takes a random variable Y and a collection of factors $\mathbf{F}$ each of which involves Y as an argument and sums Y out from the product of all factors in $\mathbf{F}$ to return a new factor. We denote the resulting (intermediate) factor produced by m_Y followed by its list of arguments, if they are not clear from the context. For instance, when we were computing $\mu_{i_1.e}(i_1.e)$ for the example in Section 4.1, to sum over $s_1.\mathbf{B}$ we had to first multiply the collection of factors formed by $f_{i_1.e}(i_1.e, s_1.\mathbf{B}, t_1.\mathbf{B})$ and $f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$ and then sum over $s_1.\mathbf{B}$ from the product to produce the new intermediate factor $m_{s_1.\mathbf{B}}(i_1.e, t_1.\mathbf{B})$. Note that $\mathbf{F}$ may contain intermediate factors produced by earlier applications of the elimrv operator.

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

Example 1. *For the examples that follow we use the following simple format for constructing labels that specify the argument order, how the arguments overlap and which argument is being summed over. 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 6. *The* elimrv$(Y, \mathbf{F}, l)$ *operation takes as input 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*.*

4.2.2 The RV-ELIM Graph

For the purposes of introducing our graph-based data structure, we will assume that we are given, besides X and $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$, an elimination order $\mathcal{O}$ that contains all random variables involved in $\mathcal{X}$ except for X. In the next section (Section 4.3), we discuss in detail how to construct such an elimination order that suits our purposes. Note that once we have an elimination order, we have the sequence of elimrv operations defined for our inference procedure. The inference procedure proceeds as follows: we collect all factors from $\mathcal{F}$ in a pool, pick the first random variable Y to be eliminated from $\mathcal{O}$, 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 from $\mathcal{O}$. The *rv-elim* graph (which stands for Random Variable ELIMination graph) essentially encodes this sequence of elimrv operations using a labeled graph.

Definition 7. *The* rv-elim *graph* $G = (V, E, \mathcal{L}_V, \mathcal{L}_E)$ *is a directed graph with vertex labels* $\mathcal{L}_V(v), \forall v \in V$*, and edge labels* $\mathcal{L}_E(e), \forall e \in E$*, that represents a run of inference on a PGM* $\mathcal{P} = \langle \mathcal{F}, \mathcal{X} \rangle$ *according to elimination order* $\mathcal{O}$ *such that:*

- *Every* $v \in V$ *represents a factor. If* v *is a root, then it represents a factor from* $\mathcal{F}$ *and* $\mathcal{L}_V(v) = id(f)$*; if* v *is not a root then it represents an intermediate factor* $m_Y = \text{elimrv}(Y, \mathbf{F}, l)$ *produced during the run of inference and* $\mathcal{L}_V(v) = l$*.*

- *For each* $m_Y = \text{elimrv}(Y, \mathbf{F}, l)$ *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 label on the edge.*

Figure 4.5 (a) shows the rv-elim graph for our running example using the same elimination order we defined in Section 4.1. One point to note about the rv-elim graph is that, in general, it can never contain a directed cycle (in other words, it has to be a directed acyclic graph (DAG)). Our example happens to be a tree; in general this is not always going to be the case; Figure 4.6 shows an rv-elim graph that is not a tree.

4.2.3 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 fac-

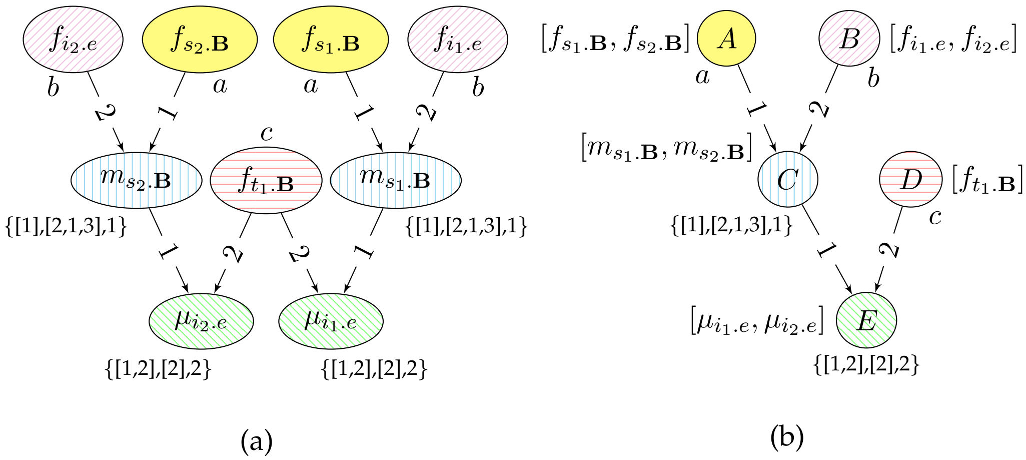


Figure 4.5: (a) rv-elim graph for the example from Figure 4.3, (b) its compressed version obtained using bisimulation. The rv-elim graph shown in (a) is a vertex-labeled, edge-labeled graph. The edges are labeled with integers (in this case, 1 or 2) and denote the order in which the parent factors are 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 vertices 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 5 vertices whereas the rv-elim graph itself contains 9 vertices, a significant reduction considering we have such a small running example.

tors. Denote by f_v the factor represented by vertex v in an rv-elim graph.

Claim 1. *For rv-elim graph $G = (V, E, \mathcal{L}_V, \mathcal{L}_E)$, two vertices $v_1, v_2 \in V$ are shared factors $f_{v_1} \cong f_{v_2}$ if:*

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

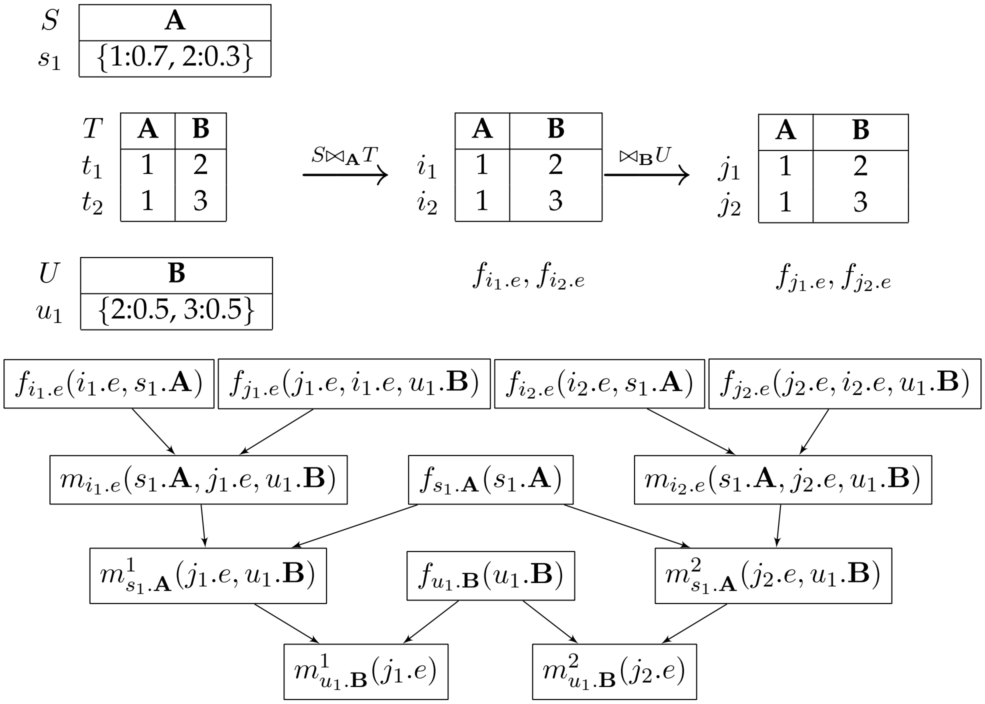

S
s_1
A
{1:0.7, 2:0.3}
T
t_1
t_2
A B
1 2
1 3
$S \bowtie_{\mathbf{A}} T$
i_1
i_2
A B
1 2
1 3
$\bowtie_{\mathbf{B}} U$
j_1
j_2
A B
1 2
1 3
U
u_1
B
{2:0.5, 3:0.5}
$f_{i_1.e}, f_{i_2.e}$
$f_{j_1.e}, f_{j_2.e}$
$f_{i_1.e}(i_1.e, s_1.\mathbf{A})$
$f_{j_1.e}(j_1.e, i_1.e, u_1.\mathbf{B})$
$f_{i_2.e}(i_2.e, s_1.\mathbf{A})$
$f_{j_2.e}(j_2.e, i_2.e, u_1.\mathbf{B})$
$m_{i_1.e}(s_1.\mathbf{A}, j_1.e, u_1.\mathbf{B})$
$f_{s_1.\mathbf{A}}(s_1.\mathbf{A})$
$m_{i_2.e}(s_1.\mathbf{A}, j_2.e, u_1.\mathbf{B})$
$m^1_{s_1.\mathbf{A}}(j_1.e, u_1.\mathbf{B})$
$f_{u_1.\mathbf{B}}(u_1.\mathbf{B})$
$m^2_{s_1.\mathbf{A}}(j_2.e, u_1.\mathbf{B})$
$m^1_{u_1.\mathbf{B}}(j_1.e)$
$m^2_{u_1.\mathbf{B}}(j_2.e)$

Figure 4.6: A three-relation join that produces a non-tree structured rv-elim graph (edge and vertex labels not shown for legibility). Note that to compute the marginal probabilities of $j_1.e$ we do not need to multiply all factors in the PGM, certain factors such as $f_{i_2.e}$ are only required to compute the marginal probabilities of the other result tuple's random variables ($j_2.e$'s) and we do this by "tagging" factors in the PGM with the random variables whose marginal probability computations they are involved in; subsequently, while performing inference we make sure that we multiply two factors only if they have atleast one tag in common.

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

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

Note that for a given internal vertex in the rv-elim graph, all incoming edges from parents are assigned distinct edge labels since we label the edges with the index indicating the position of the factor represented by the parent in $\mathbf{F}$ of the corresponding elimrv operation and two factors cannot be at the same position (Definition 7).

We can now use Claim 1 to determine the intermediate shared factors that get generated during the inference process. The important thing to realize is that we can do this *without actually computing these intermediate factors*. For instance, recall that in Section 4.1, we showed that during the run of inference for our running example, $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ were intermediate factors that turned out to be shared (shown in dashed boxes in Figure 4.5(a)). By looking at the rv-elim graph (Figure 4.5(a)) this is now easy to see since:

- They have the same vertex label {[1],[2,1,3],1}.
- Both $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ have parents $f_{s_1.\mathbf{B}}$ and $f_{s_2.\mathbf{B}}$, respectively, via edges labeled 1, and $f_{s_1.\mathbf{B}} \cong f_{s_2.\mathbf{B}}$ since they have the same vertex label a and are roots.
- Both $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ have parents $f_{i_1.e}$ and $f_{i_2.e}$, respectively, via edges labeled 2, and $f_{i_1.e} \cong f_{i_2.e}$ since they have the same vertex label b and are also roots.

Thus by Claim 1, $m_{s_1.\mathbf{B}} \cong m_{s_2.\mathbf{B}}$.

Given a graph (like the rv-elim graph shown in Figure 4.5(a)) and a property (such as the one specified in Claim 1), we now need an algorithm for partitioning the vertices into collections of shared factors. It turns out that there exist reasonably fast algorithms that can partition the set of vertices into disjoint sets which, because of our construction, will satisfy this property. These algorithms generally go by the term *bisimulation* (also known as the *relational coarsest partition* problem [Paige and Tarjan, 1987]). 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 [Dovier et al., 2001] 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, is the result of taking the union of all edges between all vertices from the input graph in p_1 and all vertices from p_2. We will refer to each resulting disjoint set of the vertices of the rv-elim graph as an *extent* and the resulting graph returned as a result of running bisimulation on the rv-elim graph as the *compressed rv-elim graph*. Figure 4.5(b) shows the compressed rv-elim graph returned as a result of running bisimulation on the rv-elim graph shown Figure 4.5(a). Notice how vertex A represents both factors $f_{s_1.\mathbf{B}}$ and $f_{s_2.\mathbf{B}}$. We show this in Figure 4.5(b) by indicating A's extent in square braces next to it. More interestingly, the pair of intermediate shared factors that we identified earlier ($m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$) has also been collapsed into one single vertex denoted by C in the compressed rv-elim graph.

Unfortunately, we cannot apply the bisimulation algorithm described in Dovier et al. [2001] directly to our problem, and this is because we have not yet addressed an important issue. Recall that we discussed how the order in which the factors appear in the elimrv operator affects the results of applying the operator. We have not yet discussed how to choose an order. For traditional inference algorithms, when eliminating a random variable, any ordering of the factors works. However, in our case, Claim 1 actually uses the order of the parents of the vertices in the rv-elim graph to determine which ones represent shared factors. This means that for us the order matters. If we do not choose the correct order then we might end up with cases such the one shown in Figure 4.7, where instead of ordering the parents of $m_{s_2.\mathbf{B}}$ with $f_{s_2.\mathbf{B}}$ as the first parent and $f_{i_2.e}$ as the second, we have placed $f_{i_2.e}$ as the first parent and $f_{s_2.\mathbf{B}}$ as the second. A direct consequence of this is that the labels on the vertices representing $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ in the rv-elim graph are now different, which means that using Claim 1 we cannot decree them to form a pair of shared factors.

The problem is that we do not know the order in which we should present the factors to each elimrv operation, and some orders produce more symmetric rv-elim graphs (with more shared factors) than others and we need to choose these orders. 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. [2001]'s, requires a different interleaving of the steps, so for completeness we first present our bisimulation algoirthm, and then the heuristic we developed for ordering parents.

4.2.4 Bisimulation for RV-ELIM Graphs

We will assume that we are given an rv-elim graph $G = (V, E, \mathcal{L}_V, \mathcal{L}_E)$ for computing marginal probabilities of random variable X from PGM $\mathcal{P}$ using the elimination order $\mathcal{O}$. Each root $v \in V$ is labeled by the id(f_v)

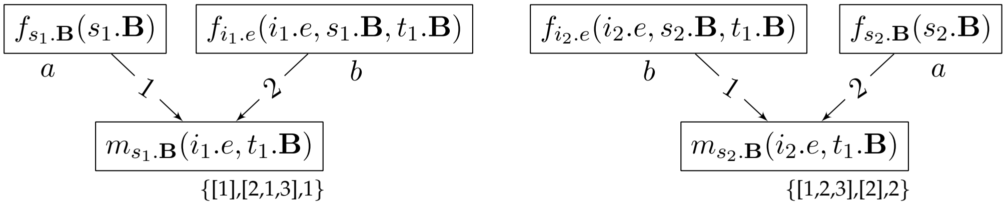

$f_{s_1.\mathbf{B}}(s_1.\mathbf{B})$
$f_{i_1.e}(i_1.e, s_1.\mathbf{B}, t_1.\mathbf{B})$
$f_{i_2.e}(i_2.e, s_2.\mathbf{B}, t_1.\mathbf{B})$
$f_{s_2.\mathbf{B}}(s_2.\mathbf{B})$
a
b
b
a
1
2
1
2
$m_{s_1.\mathbf{B}}(i_1.e, t_1.\mathbf{B})$
$m_{s_2.\mathbf{B}}(i_2.e, t_1.\mathbf{B})$
{[1],[2,1,3],1}
{[1,2,3],[2],2}

Figure 4.7: A poor ordering of parent vertices.

where f_v denotes the factor from $\mathcal{F}$ represented by v. We will assign the remaining vertex labels (for the internal vertices) and the the edge labels in G dynamically through the bisimulation algorithm we present.

A *partition* denotes a division of the set of vertices of the rv-elim graph into disjoint sets; each disjoint set is denoted a *block*. The full algorithm is described in Algorithm 1. The bisimulation algorithm starts by computing ranks for each vertex in the rv-elim graph (a simple depth-first search should do this). After computing ranks, the algorithm starts by assigning the roots in the rv-elim graph to the blocks formed by their labels. After this, it goes through the vertices at rank i, partitioning them into blocks. Note that when we are dealing with vertices at rank i, we only need the partitioning on the vertices at ranks $i' < i$, since according to Claim 1, the partitioning of a vertex only depends on its label and its parents' partitioning and the parents of vertices at rank i can only have ranks $i' < i$ (the rank computation scheme guarantees this). The nested for loops basically achieve this. They take all vertices at rank i, choose orders for each vertices' parents (we will discuss how this is done shortly), forms the label and the key based on this ordering and partitions these vertices based on the constructed key. See Dovier et al. [2001] for proof of correctness when the vertex and edge labels can be statically allocated.

Parent ordering heuristic To order the parents of each internal vertex v in the rv-elim graph before partitioning them, we simply order the parents based on their block-ids (assuming the block-ids can be ordered). We can do this using Algorithm 1 since when we are about to decide in which block to place v in, we have the blocks of its parents available. Recall that Claim 1 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. This heuristic helps align the parent vertices.

Algorithm 1, by itself, is reasonably efficient. Its time complexity, assuming we use the heuristic that orders based on block-ids, is $O(|V| + |E|)$

Algorithm 1: Bisimulation for RV-Elim Graphs.

```
input  : RV-Elim graph $G = (V, E, \mathcal{L}_V, \mathcal{L}_E)$ with roots labeled.
output : A disjoint partition over $V$.
$d(v) = \begin{cases} 0, \text{if } v \text{ is a root} \\ 1 + \max\{d(v')|v' \to v \in E\} \end{cases}$      /* compute depths */
$\rho \leftarrow \max\{d(v)|v \in V\}$
$B_{0,l} = \{v|v \text{ is root} \wedge \mathcal{L}_V(v) = l\}$      /* compute initial partition */
$C = \{B_{0,l}\}$
$B_i = \{v|d(v) = i\}, \forall i = 1 \ldots \rho$
for $i = 1 \ldots \rho$ do
    foreach $v \in B_i$ do      /* construct keys to partition on */
        order parents by block-ids
        construct label $\mathcal{L}_V(v)$
        construct key $k_v$ with $\mathcal{L}_V(v)$ and parents' blocks-ids
    end
    add $B_{i,k} = \{v \in B_i|k_v = k\}$ to $C$
end
return the final partition $C$
```

(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 $\mathcal{L}_V(v)$) + $O(|V|)$ to partition vertices at rank i into blocks based on their key. Adding up, this gives us $O(\sum_v d_v \log d_v + |V|) = O(|E| \log D + |V|)$ where D is the maximum in-degree of any vertex in the rv-elim graph.

4.2.5 Inference with the Compressed RV-ELIM Graph

Having computed the partition of the vertices using Algorithm 1, as indicated earlier, we can now construct the compressed rv-elim graph by constructing a graph where each block in the partition is represented by a vertex, the label on the block is the label on the vertices in the block, and two blocks have an edge labeled i between them if there exists a pair of vertices in the two blocks that have an edge labeled i. These definitions are consistent because the blocks of the partition correspond to particular keys constructed by Algorithm 1 which contain the vertex labels and edge labels, and all vertices in block have the same key.

We can now perform inference on the compressed rv-elim graph. To seed the inference, we simply copy the function components of the factors corresponding to roots of the rv-elim graph to the roots in the compressed rv-elim graph. Then we call a depth-first search procedure (dfs) from the leaf in the compressed rv-elim graph which 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 the parent's 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 leaf of the compressed rv-elim graph. If our inference required computing marginal probabilities of multiple random variables then this can also be done using our approach but in this case the compressed rv-elim graph may have multiple leaves. If the user requests marginal probabilities for random variable X, then we simply need to find the leaf in the compressed rv-elim graph that contains (unnormalized) $\mu(X)$ in its extent and return that (after normalization). This last step can be made faster if we maintain a mapping from random variables X to the leaves of the compressed rv-elim graph that contains the corresponding (unnormalized) marginal probability function.

4.3 Computing Elimination Orders

One of the important steps in performing probabilistic inference is to choose a good elimination order that helps run inference without producing too many large intermediate factors (in terms of number of arguments) during the run of inference. This can make the difference between inference being tractable or intractable since the size of a factor is proportional to the product of the domain sizes of its argument random variables. In our case, since we are interested in exploiting shared factors, and since the elimination order affects the rv-elim graph constructed, we would like elimination orders that produce smaller factors and, at the same time, produce rv-elim graphs that can be compressed using bisimulation. Unfortunately, even without consideration of shared factors, the problem is known to be NP-Hard [Arnborg, 1985]. Thus, as is done in traditional inference algorithms, we resort to heuristics. In particular, we introduce a novel version of the popular minimum size heuristic (MSH) [Kjaerulff, 1990] that is used with traditional exact inference algorithms to construct effective elimination orders that can help exploit shared factors[*].

Our elimination order generation heuristic works in two phases:

[*] For an alternate, more integrated, elimination order generation heuristic see Sen et al. [2009b].

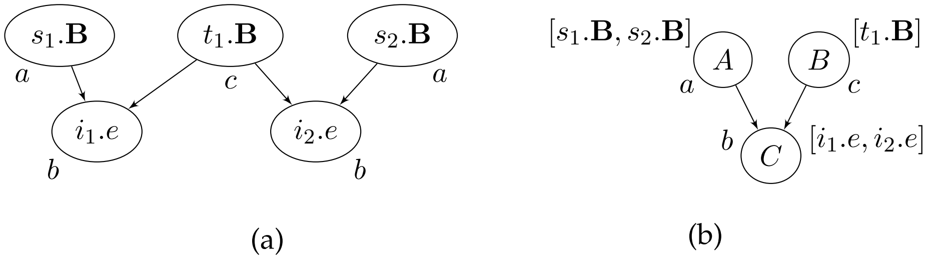


Figure 4.8: (a) Example PGM graph (b) its compressed version.

- We first identify sets of "similar" random variables, as we will explain shortly, this should help construct elimination orders that lead to rv-elim graphs which can be compressed better.
- Traditional MSH defines a notion of neighborhood for random variables. We show below that, for our purposes, this notion is no longer adequate and we introduce a novel version of MSH that helps avoid large intermediate factors.

We first explain the need to look for random variables that are "similarly" positioned in the PGM produced by query evaluation. Recall from our running example that we eliminated $s_1.\mathbf{B}$ and $s_2.\mathbf{B}$ one after another. If instead we had eliminated $t_1.\mathbf{B}$, then we risk combining shared factors into potentially one single factor and risk loss of symmetry in the resulting rv-elim graph. What we need to do here is find sets of random variables that occur in shared factors. Eliminating these one after another should help generate rv-elim graphs with better compression properties. Fortunately, we can easily represent a PGM as a graph where the random variables are represented using vertices and correlations are represented using edges (Figure 4.8(a) shows the PGM graph for our running example) and we can use this PGM graph to find similar random variables simply by labeling the vertices using the ids of the factors from the PGM (if the random variable is present in multiple factors then aggregate their ids using some operation such as max or sum, assuming the ids are numbers). Then we run a bisimulation on the PGM graph to compute a partition on the random variables of the PGM and the corresponding compressed PGM graph (Figure 4.8(b) shows the compressed PGM graph for our running example). Each extent thus obtained after bisimulation contains similar random variables. Note that, unlike rv-elim graphs which are guaranteed to be directed acyclic graphs, the PGM graph can be cyclic. Bisimulation algorithms for general graphs (with cycles) are available [Dovier et al., 2001; Paige and Tarjan, 1987].

We now explain the second step. Having constructed the sets of similar random variables we would now like to ensure that we eliminate those random variables one after another and that we avoid generating large factors in the process. One simple way to do this is to produce an ordering on the vertices of the compressed PGM graph and then expand the entries in that ordering using the extents. In addition, producing an ordering on the vertices of the compressed PGM graph is likely to be faster since the compressed graph is likely to contain less vertices compared to the number of random variables in the PGM. We now proceed towards applying (some suitable modification of) MSH on the compressed PGM graph, and for this we need some background on MSH. The basic tenet underlying traditional MSH is the notion of neighborhood of a random variable which is defined as the set of distinct random variables with which it appears as arguments to factors in the PGM. MSH works by greedily picking the random variable with the smallest neighborhood to be eliminated first, updating the neighborhoods of all random variables involved in the intermediate factor introduced by the elimination until all random variables to be eliminated have been picked.

However, the original MSH may not work on the compressed PGM graph. The problem here is that the neighborhood of a vertex in the compressed PGM graph is not a good indicator of the size of the intermediate factor produced by an elimination. This leads us towards defining a new neighborhood criterion that involves not only the neighborhood in the compressed PGM graph but also the extents of the vertices in the neighborhood. Define *avg. neighborhood size* to be $= \frac{\sum_{v' \in \mathcal{N}(v)} |\text{extent}(v')|}{|\text{extent}(v)|}$ where $\mathcal{N}(v)$ denotes the neighborhood of v in the compressed PGM graph. Essentially, avg. neighborhood assumes that there are as many neighbors to vertex v as there are random variables in all neighbors' extents summed up. It essentially tries to estimate the neighborhood of the vertex with respect to the uncompressed PGM graph, and it compensates for the case when v itself has a large extent by dividing by the extent size. Thus it tries to make MSH behave as if we are running it on the uncompressed PGM graph, but actually runs on the compressed PGM graph thus making it more efficient. Algorithm 2 shows the final modified minimum size heuristic algorithm.

Algorithm 2: Modified Minimum Size Heuristic

```
input : Compressed PGM graph $G = (V, E)$, and vertex $v_X$ that
        contains $X$ (whose marginals we need) in its extent.
output: Ordering over all random variables that need to be eliminated.

intialize empty list O
while $\exists v \in V$ s.t. $v \neq v_X, v \notin \mathbf{O}$ do
    pick vertex $v \neq v_X$ with the smallest avg. neighborhood
    add $v$ to O
    introduce an edge between every pair of neighbors of $v$
end
construct $\mathcal{O}$ by expanding entries in O with their extents
add extent($v_X$)$\setminus\{X\}$ to $\mathcal{O}$
return $\mathcal{O}$
```

4.4 Experimental Evaluation

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

- In most cases, our approach is significantly faster than the standard inference algorithm.
- In a small number of cases, our approach loses out to the baseline inference approach we compare against; but in these cases the difference between the time it took to run our approach and the baseline approach was not large.[†]

[†] Note that early stopping techniques are possible, such as once we run bisimulation on the PGM graph and find out that the extents of the compressed PGM graph are small then we can switch our inference engine and resort to standard

We compare against a baseline exact inference algorithm, denoted BatchVE, which is a modified version of variable elimination (VE) except that if the PGM contains multiple random variables whose marginal probabilities we are interested in, then it avoids multiple passes through the PGM like standard VE [Zhang and Poole, 1994] does. We refer to our approach, which constructs a compressed PGM to exploit shared factors, as Lifted Inference or LiftedInf, in short. For each experiment we report five numbers:

- **Relational algebra operations** (Rel. alg. ops): Reports the time taken to perform the relational algebra operations in the query to construct the PGM.

- **BatchVE arithmatic operations** (BatchVE arith. ops.): Reports the time taken to multiply factors and sum over random variables during inference for BatchVE.

- **BatchVE remaining operations** (BatchVE rem.): Reports the time required to perform the remaining BatchVE operations such as determining the elimination order.

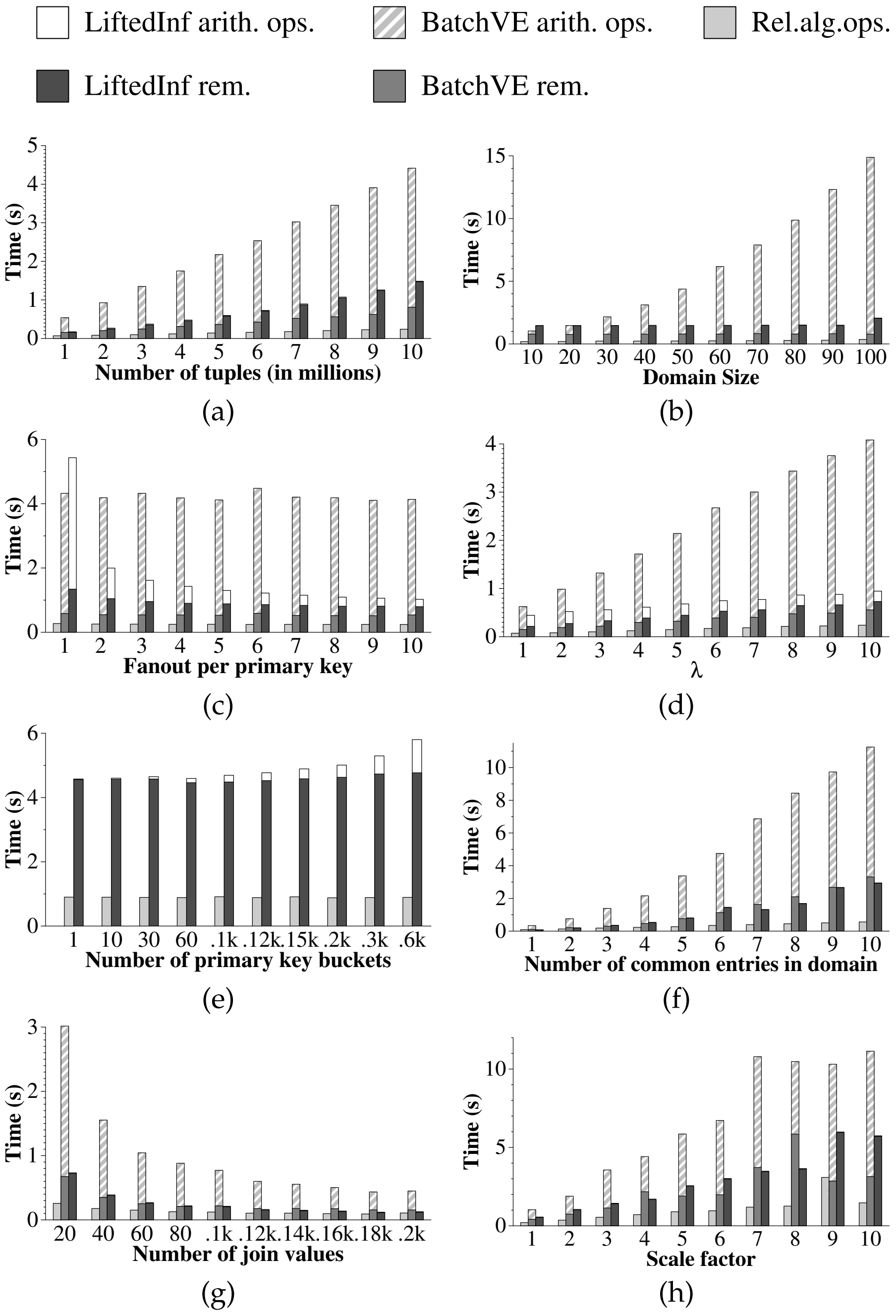
- **LiftedInf arithmatic operations** (LiftedInf arith. ops.): Reports the time spent multiplying functions of factors and summing over arguments (on the compressed rv-elim graph) for our approach.

- **LiftedInf remaining operations** (LiftedInf rem.): Reports time taken to perform the remaining operations for the approach we described in this chapter, this includes the various runs of bisimulation and the time spent to determine the elimination order from the compressed PGM graph.

For each experiment, we report three bars (except for Figure 4.9(e)): the first bar reporting the rel. alg. ops. time; the second, time spent by BatchVE; and the third, time spent by LiftedInf. See the legend (shown at the top in Figure 4.9) for more details. Note that no single bar reports the actual time to run the query. To find out the total time taken to run the query we need to add the rel. alg. ops. time to the second bar or the third bar, depending on the algorithm.

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

inference, but for our experiments we did not include this approach.



(a) (b)

(c) (d)

(e) (f)

(g) (h)

Figure 4.9: Plots for experiments on synthetic and TPC-H data. The legend is shown at the top.

4.4.1 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 performance of the two algorithms vary on various characteristics of the data. In addition to the relation containing the various advertisements (Ad) described in Figure 4.1, we added another relation which denotes the source websites from which the ads were pulled (S). Each tuple in S is an uncertain tuple with an associated probability of existence which depends on how reliable the website's information is. For these experiments, we ran the following query: $\prod_{\mathbf{AdID}}((\sigma_{\mathbf{Color=c}} Ad) \bowtie_{\mathbf{SID}} S)$ where c denotes a specific color and **SID** is a primary key in S and acts as a foreign key in Ad. Besides the uncertain tuples in S, we set the **Color** attribute values to be uncertain and these were correlated with the corresponding **Make** attributes. A car of a certain **Make** can have one of 4 distinct **Color**s. The parameters that we varied for these experiments are d (domain size of **Make**, default was 50), n (the number of attribute uncertainty tuples in Ad, default value is 1000) and fanout (the number of tuples in Ad that each tuple from S joins with, default value is 1000).

In Figure 4.9(a), we show how LiftedInf and BatchVE perform when we vary n from 100 to 1000. Notice that LiftedInf significantly reduces the time spent performing arithmatic operations. Note that on the x-axis in Figure 4.9(a), we report the size of Ad in terms of number of tuple uncertainty tuples to help the reader compare with previous work on probabilistic databases since our formulation can deal with both attribute uncertainty and tuple uncertainty but most recent work can handle tuple uncertainty only. A simple rule of thumb to compute the size of attribute uncertainty relations in tuple uncertainty format is $n \times d_1 \times d_2 \times \ldots d_{|attr(R)|}$, where n is the number of attribute uncertainty tuples and d_i is the domain size of the i^{th} uncertain attribute in the relation (assuming all i^{th} uncertain attribute values in the relation have the same domain size). For our experiment, this gives us $n \times d \times 4d$. See Section 4.5 for more details on this conversion from attribute uncertainty to tuple uncertainty.

Figure 4.9(b) shows the performance of the two inference algorithms with varying domain sizes. Notice how at $d = 10$, LiftedInf performs worse (because small domain sizes means small factors and therefore, less time spent on arithmatic operations), but the difference between its time and BatchVE's time is not large.

The third experiment we ran (Figure 4.9(c)) is the most interesting experiment in this subsection. Here we varied the fanout from 1 to 10 to vary the symmetry in the PGMs produced by the query (but kept the number of tuples in Ad fixed). At fanout 1, we have no symmetry and no shared factors in the base data, since every tuple from S has a unique existence probability, but the shared factors increase as we increase fanout. Thus, at fanout 1, LiftedInf should perform worse, and it does, but not by a huge amount. 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), LiftedInf is already doing better than BatchVE. At fanout 10, it does much better than BatchVE.

In Figure 4.9(d), instead of keeping the fanout constant for all tuples in S, we sampled it from a Poisson distribution with parameter λ. In this case however, we kept the number of tuples in S fixed. Note that at $\lambda = 1$, most fanouts sampled turn out to be 1, but some samplings produce numbers greater than 1 and LiftedInf utilizes this to do better than BatchVE, even at $\lambda = 1$. At $\lambda = 10$, LiftedInf 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 for existence probabilities by dividing the tuples in S into buckets. Two tuples in the same bucket had the same existence probability. The number of tuples in S were fixed to 600, so at 600 buckets (right end of the plot in Figure 4.9(e)), we have exactly 1 tuple belonging to each bucket. Figure 4.9(e) shows how LiftedInf's performance deteriorates when the number of buckets increase. Note that we do not show the time taken by BatchVE in this case since it would obscure the trend of LiftedInf (BatchVE took around 25 seconds for this experiment).

4.4.2 Experiments with uncertain join attributes

The next two plots (Figure 4.9(f) and (g)) relate to a two relation join between S and Ad where the join attribute **SID** itself was uncertain. This relates to the case of link uncertainty or structure uncertainty [Getoor et al., 2002], where we are unsure about the primary/foreign key values in the data. For instance, we may have another relation in our database which stores the id of the person who posted the pre-owned car ad and 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. However, 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 we wanted to keep some control over the complexity of the PGM. We setup this experiment in the following fashion: first we contructed k key values, then for each tuple in either relation, we polled from this pool 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 4.9(f) (varying m with k held constant at 100) and Figure 4.9(g) (varying k with m held constant at 2) show the results.

4.4.3 Experiments with TPC-H data

Following previous work, we also ran experiments based on the TPC-H schema. We picked Q5 from the TPC-H specification since this involves a join among six relations of which we made 4 relations (customer, lineitem, supplier and order) probabilistic. The query tries to determine how much volume of sales is being generated in various regions. Each customer makes 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 part being ordered's type (domain size d), and the orderdate attribute uncertain (domain size d). We set the parameters in the following manner: $k_1 \sim Poisson(2)$, $k_2 \sim Poisson(3)$, $p_1 = p_2 = 5$, $c = 3$, $d = 50$. We defined the scale factor to be the number of tuples in lineitem in tuple-uncertainty format divided by 6×10^6. The results are shown in Figure 4.9(h). The results showed similar trends when we tried other settings of the parameter values; for instance the execution time for LiftedInf went down when we decreased c and increased d and so on.

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

AdID	Make	Color	Price
1	Honda	?	\$9,000
2	?	?	\$6,000
3	?	Beige	\$8,000
⋮	⋮	⋮	⋮

Color	f_{color}
Black	0.75
Beige	0.25

Make	f_{make}
Honda	0.55
Toyota	0.45

AdID	Make	Color	Price	prob.
1	Honda	Black	\$9,000	0.75
1	Honda	Beige	\$9,000	0.25
2	Honda	Black	\$6,000	0.4125
2	Honda	Beige	\$6,000	0.1375
2	Toyota	Black	\$6,000	0.3375
2	Toyota	Beige	\$6,000	0.1125
3	Honda	Beige	\$8,000	0.55
3	Toyota	Beige	\$8,000	0.45
⋮	⋮	⋮	⋮	⋮

Figure 4.10: Database with pre-owned cars for sale (a) attribute-uncertainty format (b) pure tuple-uncertainty format.

shared factors.

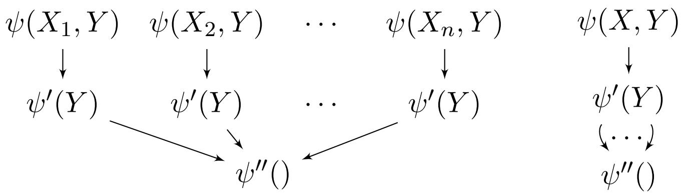
4.5 Discussion

Recall that, in Chapter 2 we surveyed a number of related works proposing different ways of modeling uncertainty in probabilistic databases. Broadly speaking, the various models can be categorized as models that associated uncertainty at the tuple level (tuple-level uncertainty) and models that represent uncertainty at both tuple and attribute levels. Especially in recent times, a number of tuple-level uncertainty models have been proposed in the probabilistic database community. These include (but are not limited to) MystiQ's *block-independent disjoint formalism* [Re et al., 2006] and Trio's *x-tuples* [Benjelloun et al., 2006; Das Sarma et al., 2006]. In contrast, our approach based on PGMs and shared correlations (first-order graphical models) allows expressing uncertainty at the tuple and/or attribute levels. Having reviewed both of these approaches, one question that begs asking is whether we are any closer to choosing one single way of modeling uncertainty. Both approaches have advantages and disadvantages, and to compare them we first need to understand how to represent the same fragment of uncertain data in either approach. To this end, we discuss a simple transformation that takes a database with attribute and tuple uncertainty and returns its representation in pure tuple uncertainty format. After that, we discuss the pros and cons of one representation scheme over the other.

Figure 4.10 shows an example where the database contains ads for pre-owned vehicles up for sale. In Figure 4.10 (a), some of the attribute values are missing; in particular, the tuple with **AdID** 1 has its **Color** attribute missing, tuple with **AdID** 3 has its **Make** attribute missing and tuple with **AdID** 2 has both attributes missing. Figure 4.10 (a) also shows the probability distributions associated with the missing attributes (common across all tuples) in the bottom. To represent such data with attribute uncertainty in pure tuple uncertainty, one approach is to compute all possible joint instantiations of every tuple present in the attribute-level uncertainty database. For instance, the first tuple in Figure 4.10 (a) can be instantiated to two tuples $\boxed{1 \mid \text{Honda} \mid \text{Black} \mid \$9{,}000}$ and $\boxed{1 \mid \text{Honda} \mid \text{Beige} \mid \$9{,}000}$, where the first instantiation's probability of existence is 0.75 while the second instantiation's is 0.25 (given by the distribution on the **Color** attribute from Figure 4.10 (a)). Note that these two instantiations cannot exist together since they come from the same attribute-level uncertainty tuple, in other words, they are correlated with a mutually exclusive dependency. In Figure 4.10 (b), we show all three tuples from Figure 4.10 (a) represented with tuple-level uncertainty and tuples present in the same block are mutually exclusive (note that this is the same representation used in other works on probabilistic databases such as x-tuples [Benjelloun et al., 2006] and block-independent disjoint formalism [Re and Suciu, 2007]).

It is difficult to see how approaches that only allow representing tuple-level uncertainty can exploit shared correlations for efficient query evaluation. As the example shows, the shared factors for **Color** and **Make** in Figure 4.10 (a) get completely obscured once we convert to tuple-level uncertainty, so much so that no pair of tuples in Figure 4.10 (b) have the same probability of existence. Moreover, the tuple-level uncertainty representation (Figure 4.10 (b)) requires 8 tuples to represent the same information that required only 3 tuples using attribute-level uncertainty (Figure 4.10 (a)) which means representing data using tuple-level uncertainty requires more space. In the above example, if the color attribute had n_c values in its domain and the make attribute n_m, then the tuple with **AdID** 2 in Figure 4.10 (a) would blow up into $n_c \times n_m$ tuples in pure tuple-level uncertainty format. Not only does this imply that the tuple-level uncertainty format requires more space to represent the same data, it also means that this form of representation involves more random variables which is another reason why query evaluation may be slower under this approach.

The problem with using tuple-level uncertainty for data that contains attribute uncertainty is that it requires computing joint distributions and this becomes an expensive operation in terms of size of the representation when we have many uncertain attribute values connected via correlations.



(a) (b)

Figure 4.11: Inversion elimination is a special case of bisimulation-based inference: (a) the rv-elim graph and (b) its compressed version.

This observation has been made in other contexts also, such as selectivity estimation in databases [Getoor et al., 2001], and is one of the main reasons why researchers in machine learning prefer working with factored representations of joint probability distributions such as probabilistic graphical models. Note that many domains produce uncertain data that can be naturally modeled using attribute-level uncertainty rather than tuple-level uncertainty such as mobile object databases [Cheng et al., 2003] and sensor network data [Deshpande et al., 2004], and these can be easily read into a database that can represent both attribute and tuple uncertainty, whereas we need to perform some transformation before we can read them into a database that can only represent tuple-level uncertainty which may lead to loss of its natural structure.

On a side note, recall that while discussing related work on lifted inference in Section 2.3, we mentioned that *inversion elimination* [de Salvo Braz et al., 2005; Poole, 2003] is one popular approach. We are now ready to demonstrate that inversion elimination is a special case of our bisimulation-based lifted inference. Given a computation of the form $\sum_Y \sum_{X_i} \prod_i \psi(X_i, Y)$ (all ψ's are shared factors), inversion elimination avoids the complexity of eliminating each $X_i, \forall i = 1, \ldots n$ separately by pushing each summation of X_i against the corresponding ψ, eliminating X_i once and then eliminating Y: $\sum_Y \sum_{X_i} \prod_{i=1}^n \psi(X_i, Y) = \sum_Y \prod_{i=1}^n \sum_{X_i} \psi(X_i, Y) = \sum_Y \prod_{i=1}^n \psi'(Y) = \sum_Y \psi'^n(Y) = \psi''()$. Figure 4.11 shows how our approach achieves the same.

4.6 Conclusion

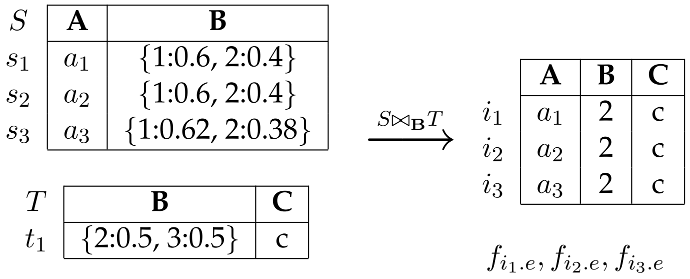
In this chapter, we showed how to exploit shared correlations to speed up probabilistic inference during query evaluation for probabilistic databases. Shared correlations are likely to exist in many probabilistic databases since probabilities and correlations often come from general statistics learnt from (large amounts of) data and rarely vary on a tuple-to-tuple basis. In addition, the query evaluation approach that builds the probabilistic graphical model on which we finally need to run inference itself tends to introduce shared correlations. We introduced a new graph-based data structure and explained how to build it from the probabilistic graphical model. We then showed how the graph can be compressed using an algorithm based on bisimulation. We empirically evaluated our approach and showed that even in the presence of a few shared correlations, we do significantly better than naive inference approaches.

Chapter 5

Approximate Lifted Inference For Probabilistic Databases

In the last chapter, we discussed how to implement a lifted inference algorithm that leverages shared correlations to efficiently perform large-scale probabilistic inference while evaluating queries in probabilistic databases. We discussed how probabilistic databases are likely to contain shared correlations that result in identical factors, and how these identical factors represent a kind of symmetry that lifted inference algorithms attempt to exploit to speed up inference. Although lifted inference often works well in cases when the PGM provides significant symmetry, sometimes such symmetry may not exist. In such cases, and in cases when the application can tolerate errors in the result of inference, we may want to resort to approximate inference to scale up to large datasets.

In this chapter, just as in the last one, we continue with our goal of designing efficient large-scale inference procedures. However, unlike the last chapter, where we concentrated on designing an exact lifted inference algorithm, here our goal is to design approximate procedures. The main question we investigate here is whether it is possible to design approximate lifted inference techniques that allow the user to trade off accuracy of inference for computational efficiency. We answer this question in the affirmative and develop two such techniques. Moreover, we show that these two techniques can be combined for more aggressive exploiting of shared correlations. Also, we show how it is possible to combine our techniques with bounded complexity inference procedures such as the mini-bucket scheme [Dechter and Rish, 2003], so that we do not need to incur the full treewidth of the PGM in question to run inference. Finally, we develop a unified lifted



S	**A**	**B**
s_1	a_1	{1:0.6, 2:0.4}
s_2	a_2	{1:0.6, 2:0.4}
s_3	a_3	{1:0.62, 2:0.38}

T	**B**	**C**
t_1	{2:0.5, 3:0.5}	c

$\xrightarrow{S \bowtie_{\mathbf{B}} T}$

	A	**B**	**C**
i_1	a_1	2	c
i_2	a_2	2	c
i_3	a_3	2	c

$f_{i_1.e}, f_{i_2.e}, f_{i_3.e}$

Figure 5.1: Running example for this chapter. Note that the prior on $s_3.\mathbf{B}$ is slightly different from the priors on $s_1.\mathbf{B}$ and $s_2.\mathbf{B}$.

inference engine that, via the use of a handful of tunable parameters, allows the user full control over what type of lifted inference algorithm s/he desires. The unified lifted inference engine allows the user choice of varying between standard or lifted inference of the exact or approximate variety, along with the user's specification of incurred complexity of inference.

Continuing in the vein of the work described in the last chapter, even though our focus in this one remains that of query evaluation in probabilistic databases, the techniques we develop can be applied to general PGMs generated from any application, even when there are no shared correlations. Also, like the last chapter, we will refrain from assuming that we are given a first-order specification of the PGM (an assumption often made by many lifted inference algorithms [de Salvo Braz et al., 2005; Milch et al., 2008; Pfeffer et al., 1999; Poole, 2003; Singla and Domingos, 2008]). Part of our task is to develop techniques that automatically determine the first-order symmetry in PGMs and exploit it to speed up inference.

In the next section, we introduce a modified version of the running example from the last chapter, which will serve as the running example for this one. In Section 5.2 and Section 5.3, we present two techniques for approximate lifted inference. In Section 5.4, we discuss how to combine our ideas with bounded complexity inference techniques and along with that, present a unified lifted inference engine that combines all the techniques. In Section 5.5, we evaluate our approaches on synthetic and real-world data and show that our techniques can achieve orders of magnitude speedup over standard ground inference procedures and the bisimulation-based exact lifted inference procedure introduced in the last chapter. We conclude with some pointers for future work in Section 5.6.

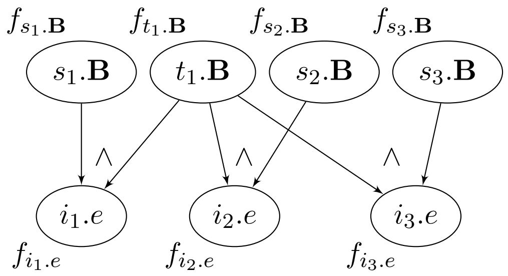
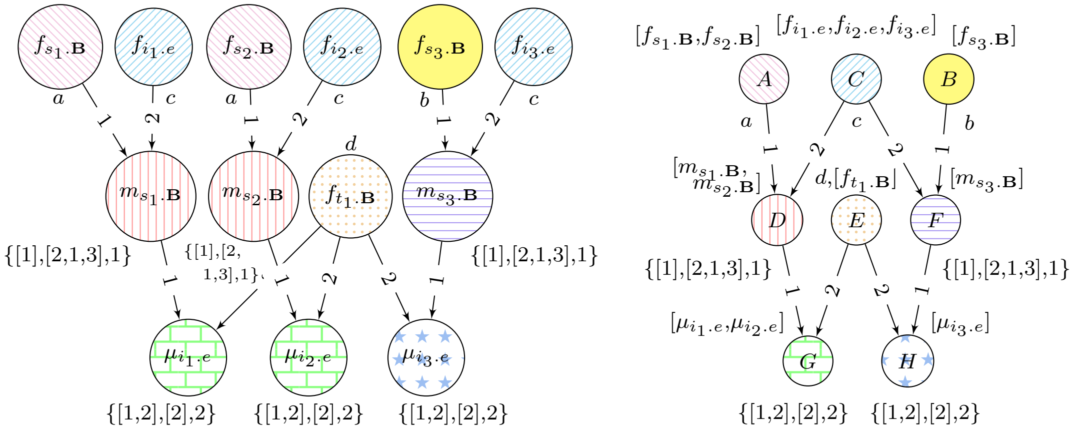


Figure 5.2: PGM produced by the running example described in Figure 5.1.

5.1 Running Example

Similar to the running example introduced in Section 4.1, in Figure 5.1, once again we have a simple 2-relation join query. In this case, S contains 3 tuples with the probability distribution associated with the new tuple, s_3's $\mathbf{B}$ attribute, being slightly different from $s_1.\mathbf{B}$ and $s_2.\mathbf{B}$. As expected, the join produces three result tuples. Our task is now to compute the marginal probabilities associated with the assignments $i_1.e = \texttt{true}$, $i_2.e = \texttt{true}$ and $i_3.e = \texttt{true}$ from the PGM shown in Figure 5.2. Figure 5.3 (a) shows the rv-elim graph obtained using the elimination order $\mathcal{O} = \{s_1.\mathbf{B}, s_2.\mathbf{B}, s_3.\mathbf{B}, t_1.\mathbf{B}\}$, and Figure 5.3 (b) shows the corresponding compressed rv-elim graph on which we can now perform inference faster. Notice how, $\mu_{i_1.e}$ and $\mu_{i_2.e}$ turn out to be identical (just like in the last chapter), but $\mu_{i_3.e}$ is partitioned into a different node in the compressed rv-elim graph. This is because one of the factors that leads to the computation of $\mu_{i_3.e}$, i.e., $f_{s_3.\mathbf{B}}$, is slightly different than the corresponding factors $f_{s_1.\mathbf{B}}$ and $f_{s_2.\mathbf{B}}$. If our application could tolerate an adequate level of approximation in the marginals computed, then we may want to pool $\mu_{i_3.e}$ along with the other two marginal distributions. This would result in a smaller compressed rv-elim graph and could lead to significant savings in computation time.

While the exact lifted inference approach described in the previous chapter can provide significant speedups when the probabilistic model contains moderate to large amounts of symmetry, in many cases we can do much better if we are willing to accept approximations in the marginal probability distributions computed. The main idea here is to explore looser versions of Claim 1 so that we can partition the vertices of the rv-elim graph into bigger blocks and thus arrive at a smaller compressed rv-elim graph. In what



(a) (b)

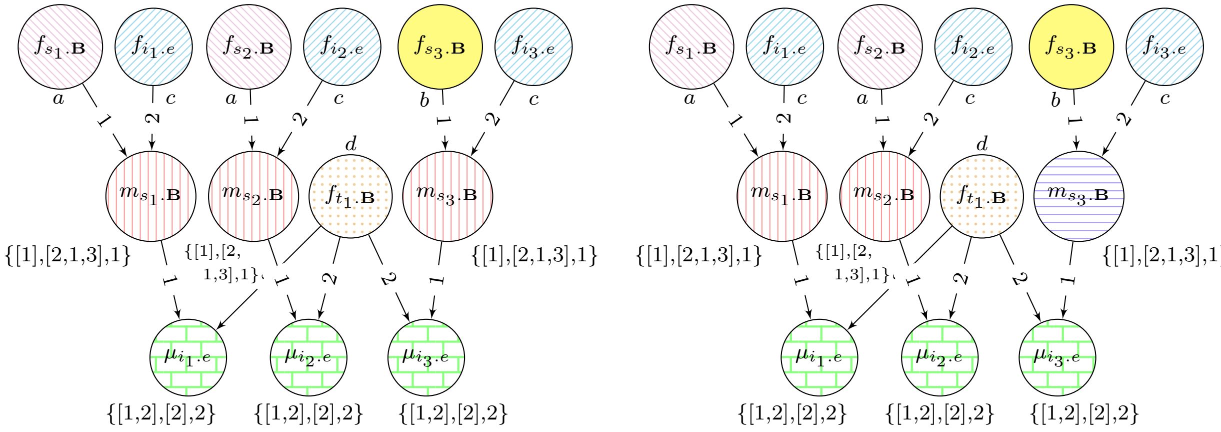
Figure 5.3: (a) RV-Elim graph for the running example (vertices partitioned into 8 blocks, shading indicates partitioning), (b) corresponding compressed rv-elim graph.

follows, we describe two separate and orthogonal generalizations of Claim 1 that can be used to implement approximate lifted inference. After that, we discuss how to combine our techniques with bounded complexity inference algorithms and finally, we discuss how to combine all of our proposed ideas together into one single general purpose approximate lifted inference engine.

5.2 Approximate Lifted Inference with Approximate Bisimulation

To introduce our first technique, we require some notation. Given a vertex, edge labeled graph $G = (V, E, \mathcal{L}_V, \mathcal{L}_E)$ such as an rv-elim graph, let $v_0, \ldots v_n$ denote an n-length *vertex path* such that $\forall i = 0, \ldots n : v_i \in V$ and $\forall i = 0, \ldots, n-1 : \exists j$ s.t. $v_i \xrightarrow{j} v_{i+1} \in E$. Further, we say that *label path* or simply, *path*, $l_0(l'_0)l_1(l'_1) \ldots l_n(l'_n)l_{n+1}$ *matches* vertex path $v_0, \ldots v_{n+1}$ (and vice versa), if $\forall i = 0, \ldots, n+1 : \mathcal{L}_V(v_i) = l_i$ and $\forall i = 0, \ldots n : \mathcal{L}_E(v_i \rightarrow v_{i+1}) = l'_i$.

We will now revisit Claim 1 and try to assign it a path-based interpretation. Using a simple induction (and the fact that edges with the same head have distinct edge labels) it is possible to show that two vertices v_1 and v_2 in



(a)

(b)

Figure 5.4: Results of running approximate bisimulation on the running example (shading indicates partitioning), (a) path-length=0 (partitioning on labels, vertices partitioned into 6 blocks) (b) path-length=1 (vertices partitioned into 7 blocks). At path-length=2, we obtain the results of exact bisimulation (see Figure 5.3 (a)).

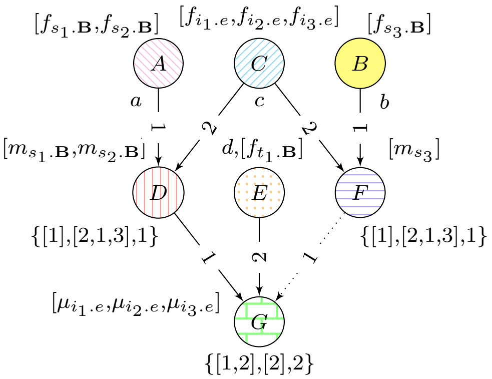
an rv-elim graph are bisimilar *iff their incoming set of paths from the roots are identical*. For instance in Figure 5.3 (a) recall that $m_{s_1.\mathbf{B}} \cong m_{s_2.\mathbf{B}}$, which have the same set of incoming paths from the roots {"$a(1)\{[1],[2,1,3],1\}$", "$c(2)\{[1],[2,1,3],1\}$"}, the matching vertex paths for $m_{s_1.\mathbf{B}}$ are $f_{s_1.\mathbf{B}}, m_{s_1.\mathbf{B}}$ and $f_{i_1.e}, m_{s_1.\mathbf{B}}$, resp., and the matching vertex paths for $m_{s_2.\mathbf{B}}$ are $f_{s_2.\mathbf{B}}, m_{s_2.\mathbf{B}}$ and $f_{i_2.e}, m_{s_2.\mathbf{B}}$, resp. Notice that this path-based interpretation of Claim 1 shows that it is a fairly stringent criteria (albeit necessary for exact inference). For instance, consider a case when two vertices deep in the rv-elim graph have large sets of long incoming paths and both sets are almost identical except for one incoming path to the second vertex which has that one label that does not allow it to match any incoming path to the first vertex; based on Claim 1 these two vertices would be placed in different blocks of the final partition and the compressed rv-elim graph would be correspondingly bloated. This sort of behaviour is, in fact, on display in our running example where $\mu_{i_2.e} \ncong \mu_{i_3.e}$ simply because, of the three incoming paths to $\mu_{i_3.e}$, "$b(1)\{[1],[2,1,3],1\}(1)\{[1,2],[2],2\}$" (matching $f_{s_3.\mathbf{B}}, m_{s_3.\mathbf{B}}, \mu_{i_3.e}$) does not match any of $\mu_{i_2.e}$'s incoming paths.

Instead of comparing sets of all incoming paths to vertices, we propose to relax Claim 1 by comparing sets of only k-length (and less than k-length)

Algorithm 3: Approximate Lifted Inference with Approximate Bisimulation

```
input  : RV-Elim Graph $G = (V, E, \mathcal{L}_V, \mathcal{L}_E)$ and path-length $k$.
output : A disjoint partitioning over $V$.
$d(v) = \begin{cases} 0, \text{if } v \text{ is a root} \\ 1 + \max\{d(v')|v' \to v \in E\} \end{cases}$
$\rho \leftarrow \max\{d(v)|v \in V\}$
$B_{0,l} = \{v|d(v) = 0 \wedge \mathcal{L}_V(v) = l\}$
$B_i = \{v|d(v) = i\} \forall i = 1 \dots \rho$
$C \leftarrow \{B_{0,l}\}_{\forall l} \cup \{B_i\}_{i=1}^{\rho}$
$X \leftarrow C$
for $j = 1 \dots k$ do
    for $i = 1 \dots \rho$ do
        foreach $B \in C$ at depth $i$ do
            order parents by block-ids in $X$
            construct labels $\mathcal{L}_V(v) \forall v \in B$
            construct key $k_v \forall v \in B$ with $\mathcal{L}_V(v)$, parent blocks-ids in $X$
            partition $B$ based on keys $k_v$
            replace $B$ in $C$ with new blocks
        end
    end
    $X \leftarrow C$
end
return $C$
```

incoming paths, where k is a tunable parameter we refer to as the *path-length*. Our compression algorithm permits high compression when the path-length is set to a low value and approaches exact bisimulation as we increase it. Figure 5.4 (a) shows the result of partitioning vertices in our example rv-elim graph with k set to 0, where we simply partition vertices based on their labels. Figure 5.4 (b) is more interesting, where we have set k to 1 and so, compare incoming paths of length 1. Note how, in this case, $m_{s_3.\mathbf{B}}$ has been differentiated from $m_{s_1.\mathbf{B}}$ and $m_{s_2.\mathbf{B}}$ since $m_{s_3.\mathbf{B}}$ has an incoming path “$b(1)\{[1], [2, 1, 3], 1\}''$ (matching $f_{s_3.\mathbf{B}}, m_{s_3.\mathbf{B}}$) of length 1 which does not match any incoming 1-length path of $m_{s_1.\mathbf{B}}$ or $m_{s_2.\mathbf{B}}$. In contrast, $m_{s_1.\mathbf{B}}$, $m_{s_2.\mathbf{B}}$ and $m_{s_3.\mathbf{B}}$ were all placed into the same block in Figure 5.4 (a). Also notice that, in Figure 5.4 (b), $\mu_{i_1.e}$, $\mu_{i_2.e}$ and $\mu_{i_3.e}$ are still partitioned into the same block (leaf vertices tiled with bricks) and this is because the only path that differentiates $\mu_{i_3.e}$ from $\mu_{i_1.e}$ and $\mu_{i_2.e}$ is a path



$[f_{s_1.\mathbf{B}},f_{s_2.\mathbf{B}}]$
$[f_{i_1.e},f_{i_2.e},f_{i_3.e}]$
$[f_{s_3.\mathbf{B}}]$
A
C
B
a
c
b
1
2
2
1
$[m_{s_1.\mathbf{B}},m_{s_2.\mathbf{B}}]$
$d,[f_{t_1.\mathbf{B}}]$
$[m_{s_3}]$
D
E
F
{[1],[2,1,3],1}
{[1],[2,1,3],1}
1
2
1
$[\mu_{i_1.e},\mu_{i_2.e},\mu_{i_3.e}]$
G
{[1,2],[2],2}

Figure 5.5: The compressed graph obtained at path-length=1 with the dotted edge being deleted since its tail has a smaller extent.

of length 2 (vertex path $f_{s_3.\mathbf{B}}, m_{s_3.\mathbf{B}}, \mu_{i_3.e}$) which is beyond the scope of the current path-length setting of 1. This changes however, when we set path-length to 2 and obtain the results of exact bisimulation shown earlier in Figure 5.3 (a).

The partitioning based on comparing incoming k-length paths can be obtained by computing k-bisimilarity [Kanellakis and Smolka, 1983] (for which algorithms are available) since these two properties are equivalent (this can be proved by induction). We formalize the k-bisimilarity property as follows:

Property 1. *Given an rv-elim graph $G = (V, E, \mathcal{L}_V, \mathcal{L}_E)$, $\cong^k$ is defined inductively. For vertices $v_1, v_2 \in V$,*

- $v_1 \cong^0 v_2$ *iff* $\mathcal{L}_V(v_1) = \mathcal{L}_V(v_2)$.
- $v_1 \cong^k v_2$ *($k > 0$) iff* $\mathcal{L}_V(v_1) = \mathcal{L}_V(v_2)$ *and* $\forall u_1 \xrightarrow{i} v_1, \exists u_2 \xrightarrow{i} v_2$ s.t. $u_1 \cong^{k-1} u_2$ *and vice versa.*

The algorithm for obtaining the partition based on $\cong^k$, Algorithm 3, begins by computing the depth of each vertex $d(v)$ and constructing an initial partition based on labels of the roots and the depths of internal vertices. Throughout Algorithm 3, we maintain two partitions, X and C. In the i^{th} iteration, X maintains $\cong^{i-1}$ and is used to update C where we construct $\cong^i$. Note that the inner two loops can be performed in $O(|E| \log D + |V|)$

time (not counting the time spent to construct the vertex labels), where D is the maximum in-degree in the rv-elim graph. Thus, Algorithm 3 runs in $O(k(|E| \log D + |V|))$ time (in contrast to Algorithm 1 which runs in $O(|E| \log D + |V|)$ time). Note that constructing the compressed rv-elim graph corresponding to $\cong^k$ is a bit more complicated now since we are no longer guaranteed that, if two internal vertices fall into the same block of the partition, then the parents will also have been placed into the same block (which holds for Claim 1). Figure 5.5 (the compressed graph obtained at k=1) illustrates this issue where all μ's have been merged into one block but their 1^{st} parents are not, thus G has two 1^{st} parents D and F, which is problematic if we want to use the compressed graph to run inference. Here, we simply get rid of the edge that corresponds to the smaller sized block (the dotted edge $F \rightarrow G$ in Figure 5.5 since F represents a block of size 1 versus D whose block size is 2) to maximize the number of correct marginal probability computations.

5.3 Approximate Lifted Inference with Factor Binning

We now introduce another way of implementing approximated lifted inference using an orthogonal generalization of Claim 1. We begin by associating with Claim 1 a distance-based interpretation. Recall that, Claim 1 bins two factors into the same block of the partition when we can guarantee that their input-output mappings are exactly the same without actually computing them. Stated differently, given any user-defined distance measure that can measure the "distance" between two factors, Claim 1 deems that these factors belong to the same block only if the distance between them is zero. Note that the converse is not true. That is, it is possible for two internal vertices in the rv-elim graph to actually represent factors that comprise of identical input-output mappings but because their parents do not belong to the same blocks or because the parents' arguments do not overlap in the same fashion, Claim 1 cannot bin these into the same block of the partition. We illustrate this with the following example:

$$\sum_{Y}\left(\begin{array}{cc|c} X & Y & f_1 \\ \hline \texttt{t} & \texttt{t} & 0.8 \\ \texttt{t} & \texttt{f} & 0.2 \\ \texttt{f} & \texttt{t} & 0.4 \\ \texttt{f} & \texttt{f} & 0.6 \end{array} \times \begin{array}{c|c} Y & f_2 \\ \hline \texttt{t} & 0.5 \\ \texttt{f} & 0.5 \end{array}\right) = \begin{array}{c|c} X & m_Y \\ \hline \texttt{t} & 0.5 \\ \texttt{f} & 0.5 \end{array}$$

$$\sum_{Y'}\left(\begin{array}{cc|c} X' & Y' & f'_1 \\ \hline \texttt{t} & \texttt{t} & 0.2 \\ \texttt{t} & \texttt{f} & 0.8 \\ \texttt{f} & \texttt{t} & 0.6 \\ \texttt{f} & \texttt{f} & 0.4 \end{array} \times \begin{array}{c|c} Y' & f'_2 \\ \hline \texttt{t} & 0.5 \\ \texttt{f} & 0.5 \end{array}\right) = \begin{array}{c|c} X' & m_{Y'} \\ \hline \texttt{t} & 0.5 \\ \texttt{f} & 0.5 \end{array}$$

where `t` and `f` denote `true` and `false` resp. Notice how factors f_1 and f'_1 have different input-output mappings ($f_1(\texttt{t},\texttt{t}) = 0.8 \neq 0.2 = f'_1(\texttt{t},\texttt{t})$) and hence cannot be binned into the same block which means that it is not possible to determine that the resulting factors m_Y and $m_{Y'}$ comprise of the same input-output mappings solely using Claim 1. This, in turn, means that any intermediate factors derived from these two factors during the inference process will always be binned separately, thus leading to a bloated compressed rv-elim graph.

Such symmetries can not be captured without actually looking into the factors and computing the distance between them (any distance measure such as KL-divergence or root mean squared distance would do). For this purpose, we ask the user for a separate parameter ϵ, that specifies an upper bound on the distance between two factors for them to be considered shared. Note that, unlike the previous algorithm, we can not compute distance between two intermediate factors without computing the factors.

To determine such a distance-based partitioning of the factors, we will need to solve the *factor binning* problem (FB):

$$\begin{array}{rl} \text{Given:} & \text{set of factors } \mathcal{F} = \{f_1, \ldots f_n\} \\ & \text{threshold } \epsilon, \text{ distance function dist}(\cdot,\cdot) \\ \text{Return:} & \text{argmin}_{\mathbf{F}\subseteq\mathcal{F}}|\mathbf{F}| \\ \text{such that} & \forall f_i \in \mathcal{F} \setminus \mathbf{F} \; \exists f \in \mathbf{F} \text{ s.t. dist}(f_i, f) \leq \epsilon \end{array}$$

We will shortly show that the factor binning problem is equivalent to the *dominating set* problem (DS):

$$\begin{array}{rl}
\text{Given:} & \text{graph } \mathbf{G} \text{ with vertex set } \mathbf{V} \text{ and edge set } \mathbf{E} \\
 & \text{denote by } N_v \text{ neighborhood of vertex } v \\
\text{Return:} & \text{argmin}_{\mathbf{D} \subseteq \mathbf{V}} |\mathbf{D}| \\
\text{such that} & \forall v_i \in \mathbf{V} \setminus \mathbf{D} \; \exists v \in \mathbf{D} \text{ s.t. } v \in N_{v_i}
\end{array}$$

Theorem 1. *FB is equivalent to DS.*

Proof. The proof is in two parts; we first show that any instance of FB can be reduced to DS and vice versa. To show the first part, we specify the reduction to DS. Given an instance of FB, define the corresponding DS by setting:

$$DS_{FB} : \mathbf{V} = \mathcal{F}, N_{f_i} = \{f_i\} \cup \{f | \text{dist}(f_i, f) \leq \epsilon\}$$

Note that any solution to DS_{FB} is a solution to FB. We show this by contradiction. Suppose solution $\mathbf{D}$ to DS_{FB} is not a solution to FB, in other words, $\exists f_i \in \mathcal{F} \setminus \mathbf{D}$ s.t. dist$(f_i, f) > \epsilon$, $\forall f \in \mathbf{D}$. This implies $N_{f_i} \cap \mathbf{D} = \emptyset$ which means that $\mathbf{D}$ is not a solution to DS_{FB} and thus we have a contradiction. Similarly, any solution to FB is a solution to DS_{FB}. Again, assume that solution $\mathbf{F}$ to FB is not a solution to DS_{FB}. Thus, $\exists f_i \in \mathcal{F} \setminus \mathbf{F}$ s.t. $N_{f_i} \cap \mathbf{F} = \emptyset$. This implies dist$(f_i, f) > \epsilon, \forall f \in \mathbf{F}$ which means $\mathbf{F}$ is not a solution to FB and we have a contradiction. Given that solution spaces of FB and DS_{FB} are same, and that the objective functions are also same, we have shown that FB can be solved by solving DS_{FB}.

The reduction in the other direction is also easy. Given an instance of DS, define the corresponding FB_{DS} by setting:

$$\begin{array}{ll}
FB_{DS}: & \mathcal{F} = \mathbf{V}, \epsilon = 0 \\
 & \text{dist}(v_i, v_j) = \begin{cases} 0 & \text{if } (v_i, v_j) \in \mathbf{E} \\ 1 & \text{otherwise} \end{cases}
\end{array}$$

Let $\mathbf{D}$ denote a solution to DS that is not a solution to FB_{DS}:

$$\begin{array}{ll}
\Rightarrow & \exists v_i \in \mathbf{V} \setminus \mathbf{D} \text{ s.t. dist}(v_i, v) > \epsilon = 0, \; \forall v \in \mathbf{D} \\
\Rightarrow & \exists v_i \in \mathbf{V} \setminus \mathbf{D} \text{ s.t. dist}(v_i, v) = 1, \; \forall v \in \mathbf{D} \\
\Rightarrow & \exists v_i \in \mathbf{V} \setminus \mathbf{D} \text{ s.t. } (v_i, v) \notin \mathbf{E}, \; \forall v \in \mathbf{D} \\
\Rightarrow & \exists v_i \in \mathbf{V} \setminus \mathbf{D} \text{ s.t. } N_{v_i} \cap \mathbf{D} = \emptyset \\
\Rightarrow & \mathbf{D} \text{ is not a soln. to } DS \text{ and we have a contradiction}
\end{array}$$

Algorithm 4: Approximate Lifted Inference with Factor Binning

```
input  : RV-Elim Graph $G = (V, E, \mathcal{L}_V, \mathcal{L}_E)$, a distance function and $\epsilon$.
output : A disjoint partitioning over $V$.
$d(v) = \begin{cases} 0, \text{if } v \text{ is a root} \\ 1 + \max\{d(v')|v' \rightarrow v \in E\} \end{cases}$
$\rho \leftarrow \max\{d(v)|v \in V\}$
$B_{0,l} = \{v|v \text{ is a root } \wedge \mathcal{L}_V(v) = l\}$
instantiate one factor per block $B_{0,l}$
$\mathbf{B}_0^{ds} \leftarrow$ compute dominating set and construct new set of blocks by merging $\{B_{0,l}\}$
$C = \mathbf{B}_0^{ds}$
$B_i = \{v|d(v) = i\}, \forall i = 1 \dots \rho$
for $i = 1 \dots \rho$ do
    foreach $v \in B_i$ do
        order parents by block-ids
        construct label $\mathcal{L}_V(v)$
        construct key $k_v$ with $\mathcal{L}_V(v)$ and parents' blocks-ids
    end
    $B_{i,k} = \{v \in B_i|k_v = k\}$
    instantiate one factor per new block $B_{i,k}$
    $\mathbf{B}_i^{ds} \leftarrow$ compute dominating set and construct new set of blocks by merging $\{B_{i,k}\}$
    $C \leftarrow C \cup \mathbf{B}_i^{ds}$
end
return $C$
```

Also, trying it the other way round. Let $\mathbf{F}$ denote a solution to FB_{DS} that is not a solution to DS:

$$\begin{aligned} &\Rightarrow \quad \exists v_i \in \mathbf{V} \setminus \mathbf{F} \text{ s.t. } N_{v_i} \cap \mathbf{F} = \emptyset \\ &\Rightarrow \quad \exists v_i \in \mathbf{V} \setminus \mathbf{F} \text{ s.t. dist}(v_i, v) = 1 > 0 = \epsilon,\ \forall v \in \mathbf{F} \\ &\Rightarrow \quad \mathbf{F} \text{ is not a soln. to } FB_{DS} \text{ and we have a contradiction} \end{aligned}$$

□

DS is NP-Complete [Garey and Johnson, 1979]. Further, Feige [1998] showed that DS is not approximable to a factor of $(1 - o(1))ln(|\mathbf{V}|)$ unless NP has "slightly super-polynomial time" algorithms (or $NP \subset DTIME(n^{\log(log(|\mathbf{V}|))})$). One way to solve DS is to utilize the fact that it is a special case of *set cover* and use the obvious greedy heuristic (described below) for set cover. This gives us an $ln(|\mathbf{V}|)$-approximation algorithm [Vazirani, 2001]. Thus, for our experiments, we use the same greedy approach to solve FB. FB is also equivalent to the *ρ-dominating set* problem Bar-Ilan et al. [1993], which, in turn, is the converse of the classic *k-center* problem [Kariv and Hakimi, 1979] where we are given a graph from which we need to choose a subset of k vertices so that their distance from the other vertices is minimized.

Even though the above discussion suggests FB is hard to solve, the situation is not so dire. When the distance function satisfies special properties, better algorithms may be available but this will require us to "tweak" the definition of FB. For instance, when dealing with euclidean spaces, there are algorithms that can solve the *minimum geometric disk cover* problem (GDC) near-optimally. GDC is posed as follows: consider a set of points in some high-dimensional plane. Our task is to return the smallest set of points (centers) from the plane such that each input point is within r distance of a center. Hochbaum and Maass [1985] describe approximation algorithms with bounds on their (non-)optimality for this problem. Note that the bounds depend on the dimensionality of the space, the algorithms work better in low dimensional spaces. Now consider the following reduction of a modified version of FB to GDC. We will assume that each input factor has d rows. We will interpret each input factor as a point in a d-dimensional space and the coordinates of the corresponding point are given by the output of the factor. Note that this is a one-to-one mapping from points to factors. Once we solve GDC on these points, we can get back factors corresponding to the returned centers. Note that this is not the same definition of FB as before since the centers need not correspond to any of the input factors. Also, to make this work desirably one may need to normalize the outputs of factors in some sensible way.

The algorithm to obtain the greedy solution for FB is to first construct each subset N_{f_i} (as defined above) and repeatedly pick f_i corresponding to the current largest N_{f_i} to include into our solution. Every time we pick f_i, we update all N_{f_j}'s by deleting from them all factors that are within ϵ distance of f_i. Another question we need to consider is whether to bin factors based on distance once and then run approximate lifted inference or whether to bin the intermediate factors based on distance also. For our experiments, we also binned the intermediate factors, since this allows us to compress the rv-elim graph more agrressively. Algorithm 4 shows the complete algorithm to run approximate lifted inference using FB.

5.4 Unified Lifted Inference

The approximation techniques we have introduced so far do not alleviate the worst-case complexity of the inference procedure. In other words, these techniques would not help if the ground inference procedure is associated with high treewidth (common with structured probabilistic graphical models). Next we show how to incorporate the mini-bucket scheme [Dechter and Rish, 2003], a bounded complexity approximate (ground) inference algorithm, this allows us to keep a tight control over the complexity of inference incurred. Next, we discuss how to combine all the proposed ideas in this chapter to construct one single unified lifted inference engine.

5.4.1 Bounded Complexity Lifted Inference

The mini-buckets scheme is a modification of the variable elimination algorithm [Zhang and Poole, 1994] where at each step instead of eliminating a random variable by multiplying *all* factors it appears as argument in, one devises a set of *mini-buckets* each containing a (disjoint) subset of factors that contains that variable as argument and then eliminates the variable separately from each mini-bucket. More precisely, given a set of factors, one first constructs a canonical partition such that all subsumed factors are placed into the same bucket of the partition. A factor f is said to be subsumed by factor f' if any argument of f is also an argument of f'. After constructing the canonical partition, the user has two choices:

- construct mini-buckets by restricting the total number of arguments i (a user-defined parameter) in each mini-bucket. Since inference complexity is directly affected by the size of the largest factor encountered, this is one way to control the amount of computation incurred.
- specify how many buckets m of the canonical partition to merge to form a mini-bucket. Again, this (indirectly) controls the size of the largest factor generated and keeps the complexity bounded.

Dechter and Rish [2003] show how such a modification of the variable elimination algorithm provides an upper bound over the numbers produced in the resulting factors.

It is easy to combine our approaches with the mini-bucket scheme. Instead of building the rv-elim graph by introducing internal vertices corresponding to intermediate factors produced by multiplying all factors involving a certain random variable as argument, we simply introduce vertices corresponding to factors produced by the mini-bucket scheme. Since our approaches work on any rv-elim graph, this requires no change to the

Parameter Name	Description
UB (bisimulation)	compresses rv-elim graphs if `true`
PL (path length)	approximate bisimulation parameter, use exact bisimulation when set to ∞
ϵ	factor binning parameter, uses factor binning if $\epsilon > 0$
UMB (mini-bucket)	allows using mini-buckets if `true`
ACR (arg. count restriction)	if `true` then restricts based on number of arguments in mini-buckets
MBR (mini bucket restriction)	if ACR=`true` then this is i (the max number of args per mini-bucket), else it is interpreted as m (the number of canonical partition buckets merged to form a mini-bucket).

Table 5.1: Parameters for our unified lifted inference engine.

approaches presented earlier, while keeping the complexity of inference bounded.

5.4.2 Unified Lifted Inference Engine

By interleaving the various steps, it is possible to combine all the ideas we have presented in this section into one unified approximate lifted inference engine. Our combined inference engine takes a set of eight parameters which define the combinations of techniques we would like to invoke (see Table 5.1). The experiments presented in the next section use this generic inference engine.

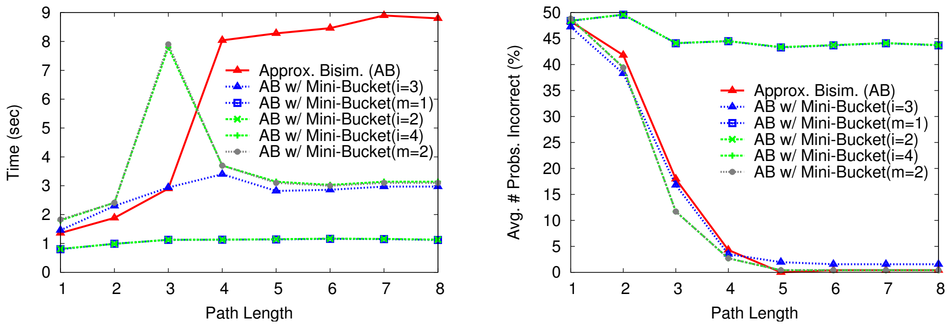
5.5 Experimental Evaluation

Through our experiments, we want to emphasize that even though our focus is query evaluation in probabilistic databases, our techniques work for any PGM. We conducted experiments on synthetic and real data to determine how lifted inference with approximate bisimulation and factor binning perform on their own. We also report experiments with our unified lifted inference engine where we used both approaches in tandem. Each number we report is an average over 3 runs, our implementation is in JAVA, and our experiments were performed on a machine with a 3GHz

Xeon processor and 3GB RAM. We compare our results with two baseline algorithms: A ground inference procedure which is basically variable elimination [Zhang and Poole, 1994] modified so that we obtain all marginals in a single pass, and the exact lifted inference procedure introduced in Chapter 4. We report two metrics for each experiment: run times incurred by the various algorithms in seconds (*Time*) and error measured by computing the average number of marginal probabilities which were not within 10^{-8} of their correct values (*Avg. #Probs. Incorrect*).

5.5.1 Synthetic Bayesian Network Generator

We set up a synthetic Bayesian network (BN) generator to test various aspects of our algorithms. The generator produces BNs where the random variables are organized in layers and random variables from the i^{th} layer randomly choose parents from the $i - 1^{th}$ layer. For our experiments, we generated BNs with 3 layers: the first layer contained 1000 random variables, the second 500 and the third 250. We introduced priors randomly for each variable in the first layer, every 25^{th} prior was identical. The random variables in the last layer are our query variables for which we computed marginal probabilities. All random variables had domain of size 30. To generate factors defining the dependency between random variables from the i^{th} and $i - 1^{th}$ layers, for each variable in the i^{th}, we randomly chose 2 parents from the previous layer. Two children can choose the same parents, so we generated non-tree structured BNs. All factors with children from the i^{th} layer are identical. This closely follows many structured probabilistic graphical models we have come across, where the priors usually closely resemble each other but may not be identical; whereas the factors defining dependencies between various random variables come from generic rules and are thus identical. We used a parameter to control how many times a random variable can be picked as a parent. This helps vary the complexity of the inference problem. We also used a parameter to add random noise after the factors are generated. We tried other parameter settings as well and the trends were as expected. For instance, increasing domain size increases the speedups obtained since with larger domains, we increase the time spent summing over random variables and multiplying factors while running ground inference – our lifted inference procedures are designed to save on this assuming the symmetry among factors is kept constant. Similarly, increasing the number of random variables with constant symmetry also increases speedups obtained.



(a) (b)

Figure 5.6: Experimental results for lifted inference with approximate bisimulation, (a) and (b) report time and error with varying path-length, respectively. Variable elimination took 25.95 sec and exact lifted inference took 8.2 sec.

5.5.2 Lifted Inference with Approximate Bisimulation

Our first set of experiments tests our algorithm for lifted inference with approximate bisimulation. The results are reported in Figure 5.6 (a) and Figure 5.6 (b). The plots show that as we increase path-length (x-axis in these plots), the time for inference (Figure 5.6 (a)) slowly increases, but error decreases (Figure 5.6 (b)). The solid line with triangles depict the results of running lifted inference with approximate bisimulation without mini-buckets, and with path-length set to 3, we see that the error stands around 18%; the inference procedure took about 3 seconds to run, which is almost a 3 times speedup over exact lifted inference (which took 8.2 seconds) and almost a 9 times speedup over ground inference (which took 25.95 sec). All the other lines in the plots correspond to lifted inference with approximate bisimulation run with various mini-bucket schemes. Among these, the mini-bucket scheme with mini-buckets restricted by argument count at $i = 3$ seems to be a promising setting (dotted line with triangles), since it runs faster than lifted inference with approximate bisimulation, but does not incur significantly higher error. Another interesting point that shows up in these plots is that with mini-buckets with $i = 4$ or $m = 2$ at path-length set to 3, the time taken to run inference goes up noticeably. This shows that at very low path-lengths, using mini-buckets could actually lead to loss of symmetry in the rv-elim graph.

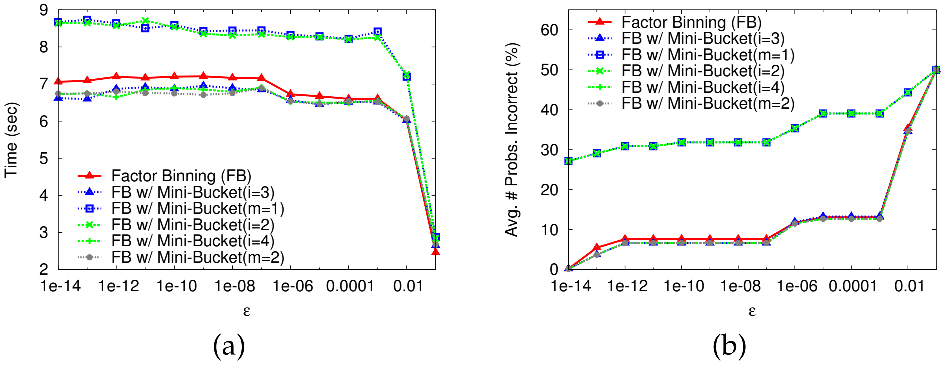
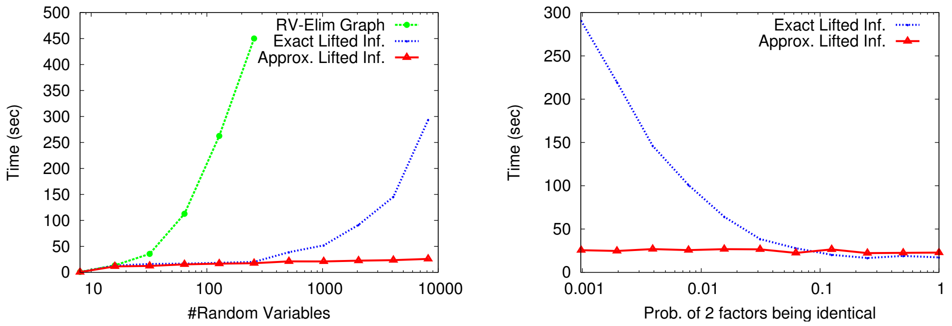


Figure 5.7: Experimental results for lifted inference with factor binning, (a) and (b) report time and error with varying path-length, respectively. Variable elimination took 33.12 seconds and exact lifted inference took 25.24 seconds.

5.5.3 Approximate Lifted Inference with Factor Binning

Our second set of experiments tests our factor binning approach. The results are shown in Figure 5.7 (a) and Figure 5.7 (b). For these experiments, we used root mean squared distance to compare two factors. More precisely, given two factors f_1 and f_2 with a common joint domain D, $\text{dist}(f_1, f_2) = \sqrt{\frac{1}{|D|}\sum_{\mathbf{x}\in D}(f_1(\mathbf{x}) - f_2(\mathbf{x}))^2}$. The plots show that as we increase ϵ (on the x-axis), the times for inference go down (Figure 5.7 (a)), and the error goes up (Figure 5.7 (b)). On these experiments, ground inference took about 33 seconds and exact lifted inference took 25.24 seconds, which means factor binning without mini-buckets (solid line with triangles) achieves a speedup of about 3.5 times over exact lifted inference, and a speedup of almost 5 times over ground inference. Among the various mini-bucket schemes, once again $i = 3$ (dotted line with triangles) seems to be the best setting; it gives small but noticeable reductions in run-times at almost no cost to accuracy. Notice that mini-buckets with small settings of either m or i tends to perform very poorly, neither giving good accuracies nor providing good run-times, and this is likely due to the sheer number of factors with which we are dealing. At such small settings, the mini-bucket scheme produces a lot of factors and computing the dominating set (which has a quadratic time complexity) becomes too expensive.



(a) (b)

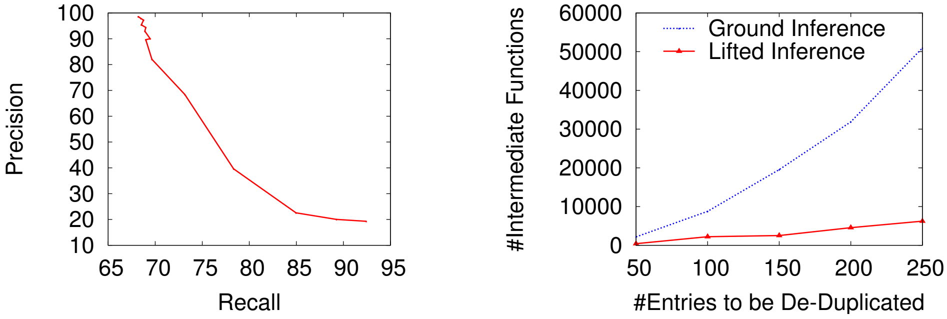
Figure 5.8: Experimental results for unified lifted inference engine. (a) depicts how variable elimination and exact lifted inference compare with respect to time with varying size of PGMs. (b) shows how unified lifted inference does not require that factors be exactly identical.

5.5.4 Unified Lifted Inference Engine

In our last set of experiments, we used both approximate bisimulation (path-length=3) and factor binning ($\epsilon = 0.01$) with mini-buckets (restricted by argument count $i = 3$). Here we report run-times for probabilistic models with varying number of random variables. The results are reported in Figure 5.8 (a) and Figure 5.8 (b). As should be clear from Figure 5.8 (a), with increasing size of the probabilistic model, all three inference procedures, ground inference, exact lifted inference and approximate lifted inference, show an increase in run-time but there is an order of magnitude difference in times between ground inference and exact lifted inference (which partitions identical factors together) and another order of magnitude speedup over exact lifted inference for approximate lifted inference (which also bins nearly identical factors together) while keeping the accuracy within bounds. Thus approximate lifted inference is more than two orders of magnitude faster than ground inference. The accuracies for approximate lifted inference for these experiments varied between 65-95%. For these complex networks, we could not run ground inference on models with more than 256 random variables due to memory limitations. Figure 5.8 (b) makes it clear how the run-time between exact lifted inference and approximate lifted inference varies. Here we set all priors in our probabilistic model similar to each other but varied the probability of two factors being identical to each other. The plot shows that as this probability increases,

Dataset	Inf. Alg.	Time (s)	Arith. Ops.	Rem. Ops.	Acc.
Cora	Ground Inf.	163.5	163	0.5	77.8
	Lifted Inf.	60.6	59.9	0.7	73
CiteSeer	Ground Inf.	101.0	100.8	0.2	68.7
	Lifted Inf.	65.0	63.9	1.1	66.8

(a)



(b) (c)

Figure 5.9: (a) Times for Cora and CiteSeer. (b) Precision-Recall curve for Cora-ER and (c) number of factors generated.

exact lifted inference captures the symmetry and does better, whereas approximate lifted inference keeps run-times low throughout.

5.5.5 Experiments on Real-World Data

We experimented with a number of real world datasets. We first report results on the Cora [McCallum et al., 2000] and CiteSeer [Giles et al., 1998a] datasets. The Cora dataset contains 2708 machine learning papers with 5429 citations; each paper is labeled from one of seven topics. The CiteSeer dataset consists of 3316 publications with 4591 citations; each paper is labeled with one of 6 topics. The task is to predict the correct topic label of the papers. We divided each dataset into three roughly equal splits and performed three-fold cross valiation. For each experiment, we train on two splits and test on the third, randomly choosing 10% of the papers' class labels to be our query nodes. Each number we report is an average across all splits. For these experiments, we produce Markov networks using the citations in the datasets as dependencies among the topic labels and then perform *collective classification* [Sen et al., 2008b]. Note that, the treewidth of the Markov networks thus produced can be unbounded, so we compare against ground inference with mini-buckets restricted to 6 arguments. Also, while testing on the third split, we include as evidence topic labels of the papers belonging to the training set linked to from the test set. We tried various parameter settings with our approximate lifted inference engine and report the best results. As Table 5.9 (a) shows, we obtained a 2.7 times speedup for Cora and 1.55 times speedup for CiteSeer with our approximate lifted inference engine over ground inference. The loss in accuracy was 4.8% for Cora and 1.9% for CiteSeer. These results were obtained with path length $= 2$, $\epsilon = 0.01$, and using mini-buckets restricted to 6 arguments. We also show how much time was spent by each inference scheme to multiply factors and sum over random variables (arithmetic operations or "Arith. Ops." in Table 5.9 (a)) and the remaining operations (or "Rem. Ops." in Table 5.9 (a)). As should be clear from Table 5.9 (a), the various operations required to implement lifted inference (bisimulation algorithms and dominating set computations) do not really add much overhead; we spend about $0.7 - 0.5 = 0.2$ seconds for Cora and $1.1 - 0.2 = 0.9$ seconds for CiteSeer (column "Rem. Ops." in Table 5.9 (a)).

We also experimented with the Cora dataset for entity resolution (Cora-ER) [Cora Entity Resolution Dataset]. For this experiment, we used a Markov logic network with 46 distinct rules. Unfortunately, we could not get any noticeable speedup for this dataset. This dataset consists solely of random variables with domain size 2 (match/non-match). As a result, all the factors produced are extremely small in size (size of a factor is determined by the number of rows in it) which implies that the time spent performing arithmetic operations (multiplying factors and eliminating random variables) is not the bottleneck during inference. The techniques proposed in this paper are mainly directed towards reducing the time spent to perform arithmetic operations. However, we do present the precision-recall curve we obtained for Cora-ER (Figure 5.9 (b), increasing argument count restriction for the mini-buckets scheme reduces precision but increases recall) and we also counted the number of intermediate factors computed by ground and lifted inference for various samplings of the dataset consisting of 50-250 bibliographic citations to be deduplicated. Figure 5.9 (c) shows that lifted inference produces far fewer intermediate factors during inference than ground inference; recall that ground inference produces an intermediate factor everytime a random variable is eliminated but lifted inference saves on this computation by computing one factor for each *block* in the final partitioning. This, in turn, indicates that the dataset possesses symmetry which could lead to speedups if the domain sizes of the random variables and factors were large. Note that Figure 5.9 (c) also gives an idea of the reduced memory consumption for lifted inference.

5.6 Conclusion and Future Work

In this chapter, we described light-weight, generally applicable approximate algorithms for lifted inference based on the graph theoretic concept of bisimulation. Essentially, our techniques are wrap-arounds for variable elimination [Zhang and Poole, 1994], and can be used whenever variable elimination is applicable, even though our focus in this thesis happens to be probabilistic databases. Besides being able to compute single node marginal probabilities, the techniques we propose here can also be used to perform other kinds of inference, including computing joint conditional probabilities and MAP assignments (by switching from the sum-product operator to max-product). One interesting avenue of future work is to look for other bounded complexity inference algorithms (besides mini-buckets) that can be combined with the techniques introduced in this chapter. Other avenues of future work are determining the optimal values of the various parameters (path-length and ϵ) automatically, and building the compressed rv-elim graph directly from the first-order description of the probabilistic model.

Chapter 6

Read-Once Functions and Probabilistic Databases

Until now, we have discussed a general representation for probabilistic databases (Chapter 3) and efficient large-scale query processing (Chapter 4 and Chapter 5). In this chapter, we take a closer look at the query evaluation problem itself. How hard is it to evaluate a single query, or, more specifically, to compute the marginal probability of a single result tuple and what can we do to do this efficiently? Recall that, for probabilistic databases based on possible world semantics, query evaluation is #P-Complete. Most of the probabilistic databases proposed in prior literature use one of two approaches to circumvent this issue: they either resort to approximate results using approximate inference ([Jampani et al., 2008; Re et al., 2007]) or they restrict their attention to a smaller class of tractable queries for which efficient evaluation is possible (*hierarchical queries* or *safe plans* [Dalvi and Suciu, 2007, 2004]).

In this chapter, we take a closer look at the latter approach. *Hierarchical* queries Dalvi and Suciu [2007], are a purely query-centric way of determining whether a query posed on a tuple-independent probabilistic database can be solved efficiently or not. More precisely, if the query q satisfies a certain criteria (defined in [Dalvi and Suciu, 2007] and reviewed in the next section) then it can be evaluated in PTIME. Since the criteria only looks at the query and does not involve the database, it stands to reason that if the query is hierarchical then it can be evaluated efficiently for *any* tuple-independent probabilistic database. As the reader may have already guessed, this is quite a pessimistic way of determining solvability of queries. Usually, the user is more interested in evaluating her/his query on the database at hand and not for all possible databases; in other words, the final evaluation problem is a result of the combination of the query and the database, not just the query alone and we need to leverage both query and data, if we are to evaluate queries in the most efficiently possible way.

In this chapter, we view the problem of evaluating queries on tuple-level uncertainty probabilistic databases at a different level of abstraction. It is straightforward to show that in such databases, the PGM corresponding to the query evaluation problem can also be represented using result tuple specific-boolean formulas, and the query evaluation problem reduces to computing the marginal probability for the boolean formulas holding true. Prior research performed by the graph theory community has shown that if the boolean formula can be factorized into a form where every boolean variable (a tuple-level existence variable, in our case) appears not more than once, then one can compute the marginal probability for the formula extremely efficiently. Boolean formulas that have such a factorization are known as *read-once* functions (Golumbic et al. [2006]; Hayes [1975]). It is also possible to show that hierarchical queries only produce result tuples with read-once functions, thus providing a connection to efficient query evaluation in probabilistic databases. In this chapter, we propose to turn the previous approach to efficient query evaluation on its head. Instead of adopting a query-centric approach, we evaluate the user-submitted query and propose algorithms that generate for each result tuple, its factorized form (if it exists), so that we can compute the required marginal probabilities efficiently. With this approach, not only do we allow efficient computation of hierarchical queries, but also for non-hierarchical queries that produce result tuples with read-once functions on the given database.

Another issue with hierarchical queries is that their definition involves the specific operators used in the query. As reviewed in Chapter 2, most of the work on hierarchical queries (Dalvi and Suciu [2007, 2004]) almost exclusively deals with equality joins[*]. Presumably, extending the approach to deal with other operators requires effort and dealing with queries composed of different operators is even more cumbersome. On the contrary, our approach of treating result tuples as boolean formulas allows us to restrict our attention to only two operators, $\wedge$ (`and`) and $\vee$ (`or`). We do not care what kind of join operator (equality or inequality or anything else) gave rise to the boolean formula associated with the result tuple. Thus, our techniques are likely to be wider in range than earlier work on efficient query evaluation.

[*] Olteanu and Huang [2009, 2008] are notable exceptions.

Here we restrict ourselves to the simpler case of probabilistic databases with tuple-level uncertainty only. The more difficult case of databases with attribute and tuple uncertainty will be left open, although some proposals in the machine learning community [Darwiche, 2002] may help in this regard (see Chapter 2 for more discussion). In the next section, we review tuple-level uncertainty probabilistic databases and discuss how to generate boolean formulas for result tuples while evaluating queries. In Section **??**, we discuss read-once functions. In Section **??**, we discuss hierarchical queries and show how they always generate read-once functions. In Section 6.2, we propose our approach to solving queries by generating read-once functions in probabilistic databases. In Section 6.3, we consider the special case of conjunctive queries without self-joins allowing for non-equality predicates. We conclude the chapter with Section 6.5, after a discussion in Section **??**.

6.1 Preliminaries

Most of the notation remains the same, but since we are dealing with a simpler model of a probabilistic database some changes/simplifications are in order. As before, let R denote a relation defined over a set of attributes $attr(R) = \{A_1, A_2 \dots A_{|attr(R)|}\}$ and each tuple $t \in R$ is a mapping from attributes to values from some pre-defined domain. We associate a unique (boolean-valued) random variable with t denoted by x_t and a probability of existence of p_t. Often, if it is clear from the context, we will abuse notation and refer to the tuple's random variable by the tuple itself. Possible worlds semantics remains the same, a (probabilistic) database $\mathcal{D} = \{R_1, \dots R_m\}$ is a set of relations and represents a distribution over many possible worlds, each obtained by choosing a (sub)set of tuples in each relation R_i to be present. If a tuple t is present, we say x_t is assigned the value `true` or `t` and `false` or `f`, otherwise. Each possible world w is associated with a probability:

$$Pr(w) = \prod_{i=1}^{m} \prod_{\substack{t \in R_i \\ x_t = \mathtt{t}}} p_t \prod_{\substack{t \in R_i \\ x_t = \mathtt{f}}} (1 - p_t)$$

Given a query q to be evaluated against database $\mathcal{D}$, the result of the query is defined to be the union of results returned by each possible world along with the marginal probabilities of each result tuple. Since we are dealing exclusively with uncertain tuples, one way to compute the marginal probability of a result tuple t produced by (relational algebra) query q is to

$$\boxed{\begin{array}{l} f(t \in R) = x_t \\ f(\sigma_c(t)) = \texttt{if } c(t) \texttt{ then } f(t) \texttt{ else f} \\ f(\prod(t_1, \dots t_k)) = \bigvee_{i=1}^{k} f(t_i) \\ f(t \times t') = f(t) \wedge f(t') \end{array}}$$

Figure 6.1: Extended definitions for σ_c, $\times$, $\prod$ where c denotes a selection predicate. t denotes true and f denotes false.

L:

	X
x_1	x_1
x_2	x_2
x_3	x_3

J:

	X	**Y**
z_1	x_1	y_1
z_2	x_1	y_2
z_3	x_2	y_3
z_4	x_3	y_3

R:

	Y
y_1	y_1
y_2	y_2
y_3	y_3

$$q() :- L(\mathbf{X}), J(\mathbf{X}, \mathbf{Y}), R(\mathbf{Y})$$

$$\begin{aligned} r = \; & x_1 z_1 y_1 + x_1 z_2 y_2 \\ & + x_2 z_3 y_3 + x_3 z_4 y_3 \end{aligned}$$

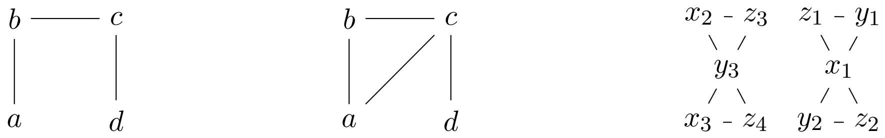
Figure 6.2: A query q, its singleton result r and the corresponding boolean formula.

extend each (relational algebra) operator in q so that it builds a boolean formula for each (intermediate) tuple generated during query evaluation. We refer to the boolean formula for t by $f(t)$. Figure 6.1 provides these extended definitions for operators σ, $\times$ and $\prod$. The marginal probability of the result tuple can then be obtained by computing the probability of the corresponding boolean formula holding true. Figure 6.2 shows a three-relation join query which produces a singleton result tuple r and the corresponding result tuple's boolean formula.

6.2 Read-Once Functions: Unateness, P_4-Free and Normality

Even though computing marginal probabilities of a result tuple's boolean formula holding true is #P-Complete in general [Dalvi and Suciu, 2004], special classes of boolean formulas allow tractable computations. *Read-once* functions are one such class of formulas:

Definition 8 (Read-Once Function [Hayes, 1975]**).** *A boolean formula ϕ is said to be read-once if there exists a factorization such that each variable appears*



(i) (ii) (iii)

Figure 6.3: (i) $\phi = ab+bc+cd$, co-occurrence graph is a P_4 and ϕ is not read-once. (ii) $\phi = c(ab+d)$, co-occurrence graph is P_4-free and ϕ is read-once. (iii) shows the co-occurrence graph for r from Figure 6.2.

not more than once.

Further, the read-once factorized form of the boolean formula is known as its read-once expression. For instance, r in Figure 6.2 is a read-once function with the read-once expression $x_1(z_1y_1 + z_2y_2) + y_3(x_2z_3 + x_3z_4)$. Prior work [Golumbic et al., 2006] has identified properties that a formula should satisfy for it to be read-once:

Theorem 2 ([Golumbic et al., 2006]). *A boolean formula is read-once iff it is unate, P_4-free and normal.*

We describe each property in turn.

A boolean formula ϕ is said to be *unate* (or monotone) [Golumbic et al., 2006] if every variable either appears in its positive or negated form throughout. Thus, ab and $\bar{a}b + \bar{a}c$ are unate but $\bar{a}b + ac$ is not.

For any boolean formula ϕ, the *co-occurrence graph* G_ϕ is formed by representing every variable in ϕ using a vertex and introducing an undirected edge between variables x_i and x_j if they appear together in some clause when ϕ is expressed in disjunctive normal form (dnf). Let X denote a subset of vertices, then the subgraph of G induced by X is the subgraph formed by restrcting edges of G to edges with end points in X. The graph P_4 denotes a chordless path with 4 vertices and 3 edges (see Figure 6.3 (i)). ϕ is P_4-free if no induced subgraph of G_ϕ forms a P_4. Figure 6.3 (i) and (ii) show two formulas one of which is not read-once because it contains a P_4, Figure 6.3 (iii) shows the co-occurrence graph for the result tuple r from Figure 6.2. Notice that in Figure 6.3 (ii), even though a, b, c and d do form a path of length 3, they do not form a P_4 because a and c have an edge between them that provides a shorting.

A formula ϕ is said to be *normal* (or clique-maximal) if every clique in its co-occurrence graph is contained in some clause in its dnf form [Golumbic et al., 2006]. For instance, even though the two formulas $\phi_1 = abc$ and

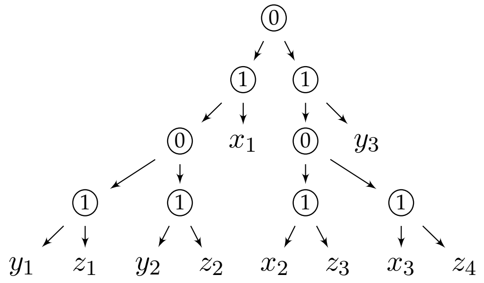


Figure 6.4: Co-tree of result tuple r from Figure 6.2.

$\phi_2 = ab + bc + ca$ both have the same co-occurrence graph (the triangle), ϕ_1 is normal (and read-once) whereas ϕ_2 is not.

Traditionally, *co-trees* [Corneil et al., 1981] have been used to concisely represent read-once expressions of read-once functions. Co-trees are trees where leaves correspond to boolean variables while internal node ① represents $\wedge$ and ⓪ represents $\vee$. A given read-once expression can be represented by many co-trees but there exists a canonical co-tree, where ① and ⓪ alternate on every path. Given the co-tree for a read-once result tuple, the probability can be computed using a simple, bottom-up procedure:

$$Pr(v) = \begin{cases} \prod_{c \in ch(v)} Pr(c) & \text{if } v \text{ is } ① \\ 1 - \prod_{c \in ch(v)} (1 - Pr(c)) & \text{if } v \text{ is } ⓪ \\ p_t & \text{if } v = x_t \end{cases}$$

where $ch(v)$ denotes children of v. The marginal probability of the result tuple can then be retrieved from the root of the co-tree. Figure 6.4 shows the co-tree for the result tuple from Figure 6.2.

6.3 Hierarchical Queries and Read-Once Functions

Earlier work on query evaluation in probabilistic databases has identified tractable queries for which probability computation is efficient and this set of queries is referred to as *hierarchical queries* [Dalvi and Suciu, 2007]. We next illustrate the close connection between hierarchical queries and read-once functions. For the ensuing discussion, we will assume that all queries are projected onto the empty set of attributes. Queries projected onto a non-empty set of attributes can be handled by replacing these attributes with constants. Let q denote a query in datalog notation. Let A denote an attribute in q and $sg(A)$ denote the set of relations a is mentioned in. In other words, $sg(A)$ denotes the set of relations or subgoals of A. For the example in Figure 6.2, $sg(\mathbf{X}) = \{L, J\}$ and $sg(\mathbf{Y}) = \{J, R\}$.

Definition 9 (Hierarchical Query [Dalvi and Suciu, 2007]). *A (conjunctive) query q is a hierarchical if for any two attributes A and B either $sg(A) \subseteq sg(B)$, $sg(A) \supseteq sg(B)$ or $sg(A) \cap sg(B) = \emptyset$.*

For instance, the query q in Figure 6.2 is not hierarchical ($sg(\mathbf{X}) \cap sg(\mathbf{Y}) = \{J\} \neq \emptyset, sg(\mathbf{X}) \nsubseteq sg(\mathbf{Y}), sg(\mathbf{Y}) \nsubseteq sg(\mathbf{X})$) but $q'() :- S(\mathbf{X}, \mathbf{Y}), T(\mathbf{Y})$ is. Further, an attribute A is said to be *maximal*, if $\forall B, sg(B) \cap sg(A) \neq \emptyset \Rightarrow sg(B) \subseteq sg(A)$. Note that, using the notion of maximality it is possible to divide the attributes in any hierarchical query q into disjoint sets $\mathbf{A}_1 \cup \ldots \cup \mathbf{A}_k$ such that:

- subgoals of attributes across the sets are disjoint: $sg(A) \cap sg(B) = \emptyset, \; \forall A \in \mathbf{A}_i, \forall B \in \mathbf{A}_j, i \neq j$
- there is a maximal attribute A in each set $\mathbf{A}_i$: $\exists A \in \mathbf{A}_i$ s.t. $sg(A_m) \subseteq sg(A) \; \forall A_m \in \mathbf{A}_i \; \forall i = 1, \ldots k$.

Dalvi and Suciu [2007] showed that hierarchical queries always give rise to result tuples with read-once functions. Here, we express the same proof for the simple case of queries without self-joins in our notation:

Proposition 6.3.1. *Hierarchical queries always produce result tuples with read-once expressions.*

Proof. Assume q is hierarchical. By induction on the number of attributes in q, we can show that the result tuple is read-once. The base case is when q has only one attribute A which is maximal and whose set of subgoals contains all the relations in q. The boolean formula for the result tuple produced by q can be expressed as $\bigvee_{c \in U_A} q[A/c]$, where $q[A/c]$ denotes the query obtained with A set to constant c and U_A denotes the domain of A. Note that $q[A/c]$ indexes into a different set of variables for different c's, thus variables appearing in $q[A/c_i]$ and $q[A/c_j]$ for $i \neq j$ are distinct. Also, within $q[A/c]$, we essentially have a cartesian product among tuples from different relations satisfying $A = c$ (if $|sg(A)| > 1$), which is clearly read-once. Thus, q produces a read-once result tuple. For the inductive case, let us assume that all sub-queries of q with at least one less attribute produces read-once result tuples. Given that we can divide the attributes in q into disjoint sets $\mathbf{A}_1 \cup \ldots \cup \mathbf{A}_m$ such that each set has a distinct maximal attribute and subgoals for attributes across sets are disjoint; let q_i denote the part of q restricted to subgoals of A_i (the maximal attribute in $\mathbf{A}_i$). Then, we can express the result tuple's boolean formula as $\bigwedge_i \bigvee_{c \in U_{A_i}} q_i[A_i/c]$. No two variables in $\bigvee_{c \in U_{A_i}} q_i[A_i/c]$ and $\bigvee_{c \in U_{A_j}} q_j[A_j/c]$ for $i \neq j$ can be identical since the relations are distinct. Also, $q_i[A_i/c]$ is read-once by our inductive hypothesis since it contains at least one less attribute than q, and $q_i[A_i/c]$ and $q_i[A_i/c']$ for $c \neq c'$ do not share variables since A_i is maximal and they index into different sets of tuples thus implying $\bigvee_{c \in U_{A_i}} q_i[A_i/c]$ is read-once. These two observations put together imply $\bigwedge_i \bigvee_{c \in U_{A_i}} q_i[A_i/c]$ is read-once, hence q produces a read-once result tuple. □

Since read-once functions form the basis of tractability of hierarchical queries and we have already seen how hierarchical queries guarantee read-once result tuples, a natural question to ask is whether the converse is true? That is, if we have a read-once result tuple is it necessary for the query that produced it to be hierarchical? If the answer is yes then that would imply that by equipping our query engine with techniques to deal with hierarchical queries, introduced in Dalvi and Suciu [2007, 2004], we have done all we can to deal with tractable cases. The answer, however, is no, as should be clear from our running example. In Figure 6.2, we showed a query that is not hierarchical, however, the result tuple it produced has a read-once expression for which probability computation is easy. In this chapter, we would like to develop techniques that helps us evaluate such cases efficiently.

6.4 Read-Once Expressions for Probabilistic Databases

We now concentrate our efforts on devising a query evaluation engine that efficiently evaluates read-once result tuples without restricting itself to hierarchical queries. We first describe a simple query evaluator that works for all result tuples without making any assumptions about the query. We then discuss the complexity of our proposed engine. After that we concentrate on a subset of relational algebra queries for which we attempt to devise a faster approach.

One viable approach to evaluating queries is to generate boolean formulas for result tuples (using the extended operators in Figure 6.1) and then determine whether it is a read-once function. If the result tuple is read-once, then we compute its probability from its read-once expression's co-tree, else we resort to a general-purpose inference engine. Checking for read-once result tuples is possible in polynomial time. We briefly discuss the algorithms that have been proposed previously in the literature to check the three properties that determine whether a formula is read-once.

In what follows, let ϕ denote a result tuple's boolean formula, $|\phi|$ the length of its dnf form (with different occurrences of the same variable counted multiple times) and $Vars(\phi)$ the distinct variables in it. To check for unateness, a linear scan of the formula is sufficient which requires $O(|\phi|)$ time. To check for P_4's in ϕ's co-occurrence graph G_ϕ, there have been a handful of algorithms proposed in the literature [Bretscher et al., 2008; Corneil et al., 1985; Habib and Paul, 2005][†]. The common aspects of all of these algorithms is that all of them require G_ϕ to be provided as input and they run in time linear in size of G_ϕ ($O(|Vars(\phi)| + |E|)$, where $|E|$ is the number of edges in G_ϕ). G_ϕ can be obtained easily from the formula's dnf form. For instance, Corneil et al. [1985] takes the co-occurrence graph and picks each variable from the graph along with its neighbours and incrementally builds the co-tree which depicts the read-once function. If the algorithm returns a co-tree successfully then the co-occurrence graph did not contain any P_4; if there is a P_4 then the algorithm stops and provides the P_4. Thus, the good thing about this algorithm is that not only does it check for the absence of P_4's, it also returns the read-once expression as a co-tree which we can subsequently use for probability computation. Figure **??** shows a run of Corneil et al.'s approach to build the co-tree for the result tuple in Figure 6.2. Bretscher et al. [2008]; Habib and Paul [2005] have a slightly different approach, they first build an ordering on the variables using the co-occurrence graph (Habib and Paul [2005] uses vertex partitioning techniques, while Bretscher et al. [2008] uses LexBFS techniques) and then subsequently use the ordering to build the co-tree. Checking for normality is also possible in polynomial time, but is more expensive than checking for unateness or P_4-freeness. Golumbic et al. [2006] describes a way to check for normality in $O(|Vars(\phi)||\phi|)$ time using the co-tree obtained from the previous step of checking for P_4's.

6.5 Read-Once Functions and Conjunctive Queries without Self-Joins

Recall that the most expensive step while generating read-once functions is the step that checks for normality. In this section, we specifically look

[†] Note that P_4-free graphs are also referred to as Cographs. Thus, some of the algorithms that test for the absence of P_4's also go by the name of "Cograph recognition algorithms".

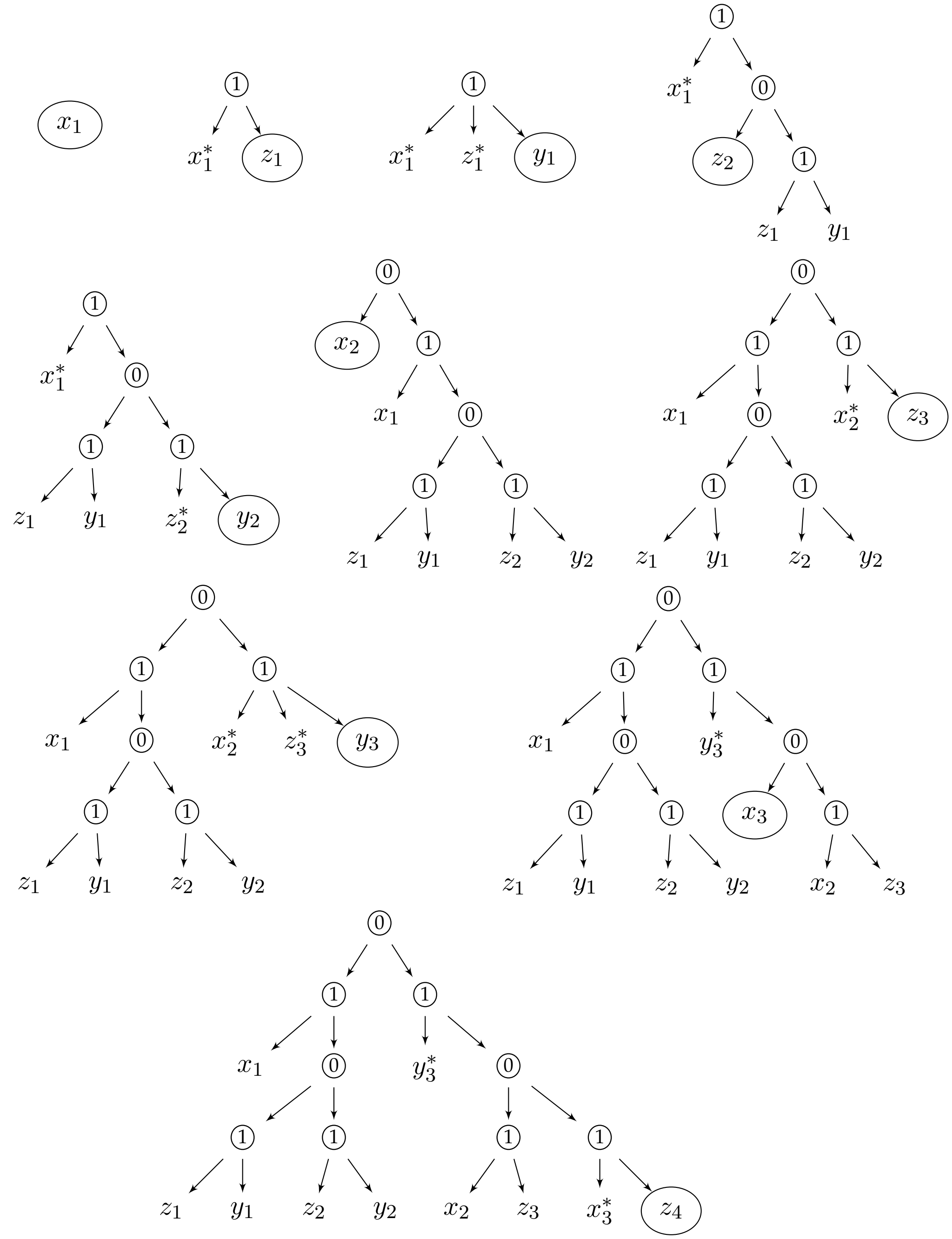


Figure 6.5: Run of Corneil et al. [1985]'s algorithm to generate the co-tree for result tuple r in Figure 6.2. In each iteration, an ellipse depicts the variable being added and the asterisks denote its neighbors already present in the tree.

at the case of generating read-once functions for result tuples produced by conjunctive queries without self-joins, also known as select(distinct)-project-join or SPJ queries. Essentially, we show that for conjunctive queries without self-joins, the normality check at the end is not required. Conjunctive queries form a large fragment of relational algebra (or SQL) and other works have also concentrated their efforts on this subclass of queries [Dalvi and Suciu, 2004; Olteanu and Huang, 2008]. We first define our notion of conjunctive queries.

Let A denote an attribute. An *atomic formula* is a predicate of the form $A\ op\ B$ where B is either an attribute or a constant conforming to the type of A and op is any binary operator conforming to the same type such as $=, >, <, \neq$ etc. A *conjunctive query* q without self-joins is a relational algebra query that involves the three operators σ, $\bowtie$ and $\prod$, where the joins are among distinct relations $R_1, \ldots R_k$. We refer to the relations in q by $Rels(q)$. We allow join and selection predicates of the form $c_1 \wedge c_2 \ldots \wedge c_n$, where each c_i is an atomic formula. Note that if the final set of projected attributes in q is empty then we refer to it as a *boolean* conjunctive query. Also note that we allow operators besides equality in our selection and join predicates which makes our definition of conjunctive queries more general than what is usually considered.

As earlier, we will denote by r the input result tuple (whose marginal probability we would like to compute), by ϕ its boolean formula (which we would like to factorize) and by G_ϕ its co-occurrence graph. We may also abuse notation and refer to the result tuple by its formula ϕ whenever it is clear from the context. A clause $C = x_1 x_2 \ldots x_n$ is a conjunction of multiple boolean variables. A monotone clause is one where all variables appear in their positive form, no negations. We will often refer to a clause as a set of variables. Further, since the variables in ϕ come from tuple-existence random variables, we will denote the relation of variable x by $Rel(x)$. We will also frequently refer to ϕ's dnf form by ϕ_{dnf}. Since we consider conjunctive queries without self-joins, ϕ_{dnf} has a very uniform structure:

Definition 10 (k-monotone dnf). *Given conjunctive query q without self-joins and any result tuple ϕ produced by it, ϕ_{dnf} is a k-monotone dnf where every clause is monotone (or unate), contains exactly one variable from each relation in $Rels(q)$ and is of size k where $k = |Rels(q)|$*[‡].

Given that conjunctive queries do not allow negations, it follows that

[‡] These observations have been made in prior work [Re and Suciu, 2008].

the result tuples we will be dealing with are automatically unate (variables appear only in their positive form). We next show that P_4-free result tuples produced by conjunctive queries without self-joins are guaranteed to be normal. This implies that when we are generating read-once functions for such result tuples, the only operation we need to do is check for P_4's and generate the co-tree corresponding to its read-once expression. The other steps for generating read-once functions are not required and this should help make our approach much more efficient. We next make an observation about result tuples produced by conjunctive queries and then prove a lemma which will allow us to prove our main result.

Property 2 (Conjunctive Query Clique Structure)**.** *Given a result tuple r produced by a conjunctive query q without self-joins along with its formula ϕ, the set of variables $C = \{x_1, \ldots x_{|Rels(q)|}\}$ represents a clause in ϕ_{dnf} iff C is a clique in G_ϕ.*

Proof. Note that if C is a clause in ϕ_{dnf} then it has to be a clique in G_ϕ by construction. The other way is also easy. An edge between two variables a, b in G_ϕ implies that the corresponding tuples satisfy all join and selection predicates in q, and agree with r on all of the final projected attributes (if a or b have any of those). Thus, a $|Rels(q)|$-sized clique in G_ϕ implies that all member variables' tuples satisfy all predicates associated with the query and agree with r's values, and should produce an intermediate join tuple and thus should appear as a clause in ϕ_{dnf}. □

Lemma 6.5.1. *Let ϕ denote a result tuple produced by a conjunctive query q without self-joins. Let $Rels(q) = \{R_1, \ldots R_{|Rels(q)|}\}$, and $a \in R_1$ and $b \in R_2$ denote two variables in ϕ. Let C_1, C_2, C_3 denote three clauses in ϕ_{dnf} such that $a \in C_1 \not\ni b$, $a \notin C_2 \ni b$ and $a, b \in C_3$. If ϕ is P_4-free then $\exists$ clause C_4 in ϕ_{dnf} that contains both a, b and $w_3, \ldots w_{|Rels(q)|}$ such that either $w_i \in C_1$ or $w_i \in C_2, \forall i = 3, \ldots |Rels(q)|$.*

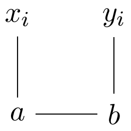
Proof. Let us begin by completing clauses C_1, C_2:

- $C_1 = \{a, b', x_3, \ldots x_n, z_{n+1}, \ldots z_{|Rels(q)|}\}$, $b' \neq b$
- $C_2 = \{a', b, y_3, \ldots y_n, z_{n+1}, \ldots z_{|Rels(q)|}\}$, $a' \neq a$

where $a, a' \in R_1$, $b, b' \in R_2$, $x_i, y_i \in R_i, x_i \neq y_i, \forall i = 3, \ldots n$ and $z_i \in R_i, \forall i = n+1, \ldots |Rels(q)|$. The x's and y's denote the variables in which C_1 and C_2 differ, besides a and b. z's denote the variables they share in common. Further note that n can be either 2 or $|Rels(q)|$.

First note that if neither edge $x_i - b$ nor $y_i - a$ exists, then G_ϕ has a P_4:

$$\begin{aligned} x_i \not- y_i &\quad \because Rel(x_i) = Rel(y_i), \text{ no self joins} \\ x_i - a &\quad \because \{x_i, a\} \subset C_1 \\ y_i - b &\quad \because \{y_i, b\} \subset C_2 \\ a - b &\quad \because \{a, b\} \subset C_3 \end{aligned}$$

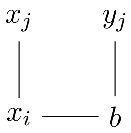


Now consider the following selection procedure that picks variables from $\{x_3, \ldots x_n\}$ and $\{y_3, \ldots y_n\}$:

$$\boxed{\text{if edge } x_i - b \text{ exists in } G_\phi \text{ then pick } x_i\text{, else pick } y_i}$$

Note that, once we have picked a set of x's and y's, we will have picked a variable from each relation $R_i, i = 3, \ldots n$. Also, note that, if among the chosen variables there exists a pair of $x_i \in R_i$ and $y_j \in R_j$ $(i \neq j)$ such that $x_i \not- y_j$ in G_ϕ, then we have a P_4:

$$\begin{aligned} x_j \not- y_j &\quad \because Rel(x_j) = Rel(y_j) \\ x_j - x_i &\quad \because \{x_i, x_j\} \subset C_1 \\ x_j \not- b &\quad \because \text{otherwise we would pick } x_j\text{, not } y_j \\ x_i - b &\quad \because \text{otherwise we would not pick } x_i \\ y_j - b &\quad \because \{b, y_j\} \subset C_2 \end{aligned}$$



Thus, if ϕ is P_4-free, then the chosen x's and y's, along with $a, b, z_{n+1}, \ldots z_{|Rels(q)|}$ form a $|Rels(q)|$-sized clique in G_ϕ, and by Property 2 that means this set of variables forms the clause C_4 we need. □

Proposition 6.5.1. *Let ϕ be a k-monotone dnf produced by some conjunctive query q without self-joins. If ϕ is P_4-free then ϕ is normal.*

Proof. Assume the contrary, i.e., let ϕ be P_4-free but not normal. This means that ϕ should have a distributed 3-clique, in other words, ϕ has at least three clauses C_1, C_2, C_3 such that $a, b \in C_1, c \notin C_1$; $b, c \in C_2, a \notin C_2$; $c, a \in C_3, b \notin C_3$ but no clause C such that $a, b, c \in C$. However, by Lemma 6.3.1, since ϕ is P_4-free there should be another clause C' that contains a, b and variables exclusively from C_2 and C_3. This means C' also contains c and hence we have a contradiction. □

X:	$\mathbf{A_1}$	$\mathbf{B_1}$
x_1	3	4
x_2	1	2

Y:	$\mathbf{A_2}$
y_1	1
y_2	3

Z:	$\mathbf{B_2}$
z_1	2
z_2	4

$$\begin{aligned} q() \coloneq\ & X(\mathbf{A_1}, \mathbf{B_1}), Y(\mathbf{A_2}), Z(\mathbf{B_2}), \\ & A_1 = A_2 \vee B_1 = B_2 \\ r =\ & x_1y_1z_2 + x_1y_2z_1 \\ & + x_1y_2z_2 + x_2y_1z_1 \\ & + x_2y_1z_2 + x_2y_2z_1 \end{aligned}$$

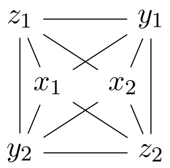


Figure 6.6: A disjunctive query where Proposition 6.3.1 does not hold.

6.6 Discussion

Having considered the case for conjunctive queries, the next obvious question is whether we can do something similar for queries with disjunctions. Some disjunctions can be allowed in the queries we considered in this chapter without breaking any of our results. For instance, if the cumulative join predicate in the query is such that it can be partitioned into a conjunctions of smaller formulas $c_1 \wedge c_2 \ldots c_m$ such that each formula c_i involves attributes from only two relations then allowing disjunctions inside each c_i still allows us to use our efficient read-once function building approach. However, if a disjunction appears between any of the c_i's, then seemingly innocuous queries produce cases where our results do not hold. Figure 6.12 shows one such case, where we have a three relation join boolean query and the join predicate involves a disjunction among attributes from three separate relations $A_1 = A_2 \vee B_1 = B_2$. The result tuple ϕ's co-occurrence graph turns out to be the complete graph minus the edges connecting tuples from the same relation ($x_1 \not- x_2$, $y_1 \not- y_2$, $z_1 \not- z_2$), which means it is P_4-free. But there are clauses which are not present in ϕ_{dnf}, $x_1y_1z_1$ and $x_2y_2z_2$, implying ϕ_{dnf} is not normal which means that Proposition 6.3.1 does not hold.

Even though our discussion throughout the chapter mainly involved tuple-independent probabilistic databases, the techniques we proposed are likely to be useful for databases with correlated tuples also. In this case, our techniques can be used to convert the part of the graphical model generated during query evaluation into a tree. It is easy to show that the combined treewidth of the complete probabilistic graphical model thus produced (including the probabilistic graphical model among the base tuples and the part constructed during query evaluation) is not larger than the treewidth of the graphical model that would have otherwise been produced.

6.7 Conclusion

In summary, we considered the problem of efficiently evaluating queries over tuple-level uncertainty probabilistic databases. For such databases, every result tuple is associated with a boolean formula and the problem reduces to computing the marginal probabilities of the result tuples returned by the query. Previously proposed approaches to this problem have either resorted to the use of expensive (exact/approximate) inference algorithms or concentrated on a subset of the query language that allows efficient evaluation. In this chapter, we build on the latter approach by going beyond just looking at the query to decide whether it is PTIME-solvable or not. Inference problems arising out of query evaluation on probabilistic databases are a combination of both the query and the database. If the result tuple's formula can be factored into a tree-structured form, then computing its marginal probability is in PTIME. We proposed efficient algorithms that return such a factorization if it exists.

Chapter 7

Conclusion

In this dissertation, we presented (a few of) the nuts and bolts that may one day form part of a system that can manage uncertain data. Here, we briefly summarize the main contributions made and list a few or the broader, more compelling, possible avenues for future work.

7.1 Summary of Contributions

Here is a brief listing of the major contributions made in this dissertation:

- We began by showing how the concept of probabilistic graphical models from the machine learning literature, can be utilized in probabilistic databases as a means of modeling uncertainty associated with data. We showed that probabilistic databases based on probabilistic graphical models have precise and intuitive semantics in terms of possible worlds, that every query posed on such a database has precisely defined answers.

- We showed how queries can be evaluated under such a setting by first generating an augmented probabilistic graphical model and then running probabilistic inference on it. We illustrated the generality of our approach: for any query a PGM can be generated on which we simply need to run inference to obtain the desired results. This also allowed us to utilize any inference algorithm (exact or approximate) developed previously to build our query evaluation engine.

- We then proceeded to generalizing our representation for modeling uncertainty. Instead of using standard graphical models (such as

Bayesian networks and Markov networks), we motivated the use of first-order probabilistic graphical models (such as probabilistic relational models and Markov logic networks). First-order graphical models represent one of the more popular approaches to modeling uncertainty not only due to their compactness and ease of maintenance, but also because they are easier to estimate statistically.

- First-order graphical models provide symmetry in the form of shared correlations. We designed a inference procedure that exploits shared correlations to perform large-scale inference efficiently for evaluating queries in probabilistic databases. Not only that, our inference procedure is general enough so that it can be applied to any probabilistic graphical model. It also subsumes inversion elimination, a popular lifted inference procedure developed in the machine learning community.

- We generalized our lifted inference scheme to be able to perform faster, approximate lifted inference. We introduced two different techniques to do this. Moreover, both techniques can be combined, and along with bounded complexity inference techniques, they form the core of a unified lifted inference scheme that lets the user specify her/his desired level of lifting, approximation and complexity of inference through the use of a handful of tunable parameters.

- Finally, we designed a novel query evaluation scheme where we first attempt to reorder the probabilistic graphical model produced during query evaluation so that we get an optimal, tree-structured graphical model (if one exists) on which we can efficiently run inference. This has the potential to reduce an intractable query evaluation problem to a tractable one.

7.2 Avenues for Future Work

Due to the almost ubiquitous need to model uncertainty for large scale data, we believe, probabilistic databases are going to be an overwhelming driving force behind database and machine learning research in the near future. Besides the aspects of user interfacing and query languages that require our immediate attention, we list below some of the main research areas that, we think, are of specific interest.

Information Integration and Information Extraction Two of the main applications that can immediately benefit from the application of probabilistic databases are information extraction and information integration. These two areas have been of interest to researchers for a long period of time, however, neither is close to being solved. Most machine learning approaches to solving these problems face issues when scaling to large data sets and most solutions proposed by the database community tend to ignore the rich correlations that can help achieve good quality solutions. By looking at these problems from the point of view of probabilistic databases, perhaps for the first time, we can deal with these issues in uniform and principled manner to achieve practical solutions that can immediately benefit many applications.

Efficient Algorithms for Lifted Inference Our original work on lifted inference just scratches the surface of this very exciting field. Recall that, the techniques introduced in Chapter 4 subsume inversion elimination, and in Chapter 5 we proposed algorithms for approximate lifted inference. In future, we would like to explore whether other kinds of lifted inference can be included into our general framework. Foremost on this list would be extending our approach to include *counting elimination* [de Salvo Braz et al., 2005] and *counting formulas* [Milch et al., 2008], which are techniques that may, in some cases, lead to exponential speedups during inference, if implemented properly.

Unifying Uncertainty Model Description and Query Evaluation Most probabilistic databases allow the user to express queries in a high-level logic-based language (usually SQL, barring a few exceptions), but do not allow declarative specification of the uncertainty model. Machine learning researchers, on the other hand, regularly use first-order logic to describe the uncertainty models but rarely allow the use of a high-level declarative language for querying purposes. We would like to explore the interplay between these two aspects in the context of use in probabilistic databases since we believe both are essential for a system to be usable. To the best of our knowledge, there is no work to date that has systematically explored the expressiveness of the various languages used to describe uncertainty models, and we would also like to explore if such high level model descriptions can be exploited to make query processing more efficient.

Approximate Inference Algorithms based on Generalizations of Read-Once Functions Following our work described in Chapter 6, where we showed how to generate a tree-structured PGM given a query, the next obvious question to follow would be: What if a tree-structured PGM does not exist? In such cases, it may be possible to construct a PGM that is "close" to being tree-structured which may help run inference fast with reasonably accurate query results. There is ample work in the graph theory community on generalizations of P_4-free graphs (such as P_4-tidy graphs [Giakoumakis et al., 1997]) that may lead us to novel approximate query evaluation algorithms which have not been seen before in the database or machine learning communities.

7.3 Conclusion

One of the over-arching themes underlying this dissertation has been to explore the synergy between related fields of research. Probabilistic databases is a topic that lies at the intersection of database research, machine learning and graph theory. Even though the challenges in working under such a setting are obvious, one needs to have expertise on not one but each of the related fields to be able to produce original, useful research, the rewards are also plentiful. Among the various pieces of work that form parts of this dissertation, perhaps the most rewarding are the ones that find use beyond just that of probabilistic databases. For instance, our work on lifted inference (Chapter 4 and Chapter 5) are of obvious interest to the machine learning community, our work on altering the structure of the PGMs to produce read-once functions may find use in the statistical relational learning community where people have recently begun to investigate the use of declarative querying (e.g., the ProbLog system [De Raedt et al., 2007]). This seems to be true for most work done in the context of probabilistic databases, and that, we believe, is what makes it worthwhile working in such an multi-disciplinary environment. We hope that further research with the canvas of probabilistic databases as the background will lead to more synergy among related research communities and will eventually lead to a system that can efficiently handle large-scale uncertain data.

Bibliography

P. Andritsos, A. Fuxman, and R. J. Miller. Clean answers over dirty databases. In *International Conference on Data Engineering (ICDE)*, 2006. 4

L. Antova, C. Koch, and D. Olteanu. 10^{10^6} worlds and beyond: Efficient representation and processing of incomplete information. In *International Conference on Data Engineering (ICDE)*, 2007. 2.1

S. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. *BIT*, 25(1):2–23, 1985. 1, 3.3.3, 4.3

J. Bar-Ilan, G. Kortsarz, and D. Peleg. How to allocate network centers. *Journal of Algorithms*, 1993. 5.3

D. Barbara, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 1992. 2.1

O. Benjelloun, A. Das Sarma, A. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *Proceedings of Very Large Data Bases (PVLDB)*, Seoul, Korea, 2006. 2.1, 2.4, 4, 4.5, 4.5

P. Bosc and O. Pivert. About projection-selection-join queries addressed to possibilistic relational databases. *IEEE Transactions on Fuzzy Systems*, 2005. 2.1

A. Bretscher, D. Corneil, M. Habib, and C. Paul. A simple linear time lexbfs cograph recognition algorithm. *SIAM Journal on Discrete Mathematics*, 2008. 6.2.1

B. Buckles and F. Petry. A fuzzy model for relational databases. *International Journal of Fuzzy Sets and Systems*, 1982. 2.1

R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluating prob. queries over imprecise data. In *International Conference on Management of Data (SIGMOD)*, 2003. 1, 4.5

S. Choenni, H. E. Blok, and E. Leertouwer. Handling uncertainty and ignorance in databases: A rule to combine dependent data. In *Database Systems for Advanced Applications*, 2006. 2.1

G. F. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 1990. 3.3.3

Cora Entity Resolution Dataset. http://www.cs.umass.edu/˜mccallum/data/cora-refs.tar.gz. 5.5.5

D. Corneil, H. Lerchs, and L. Burlingham. Complement reducible graphs. *Discrete Applied Mathematics*, 1981. 6.1.1

D. Corneil, Y. Perl, and L. Stewart. A linear recognition algorithm for cographs. *SIAM Journal of Computing*, 1985. 6.2.1, 6.4

R. Cowell, A. Dawid, S. Lauritzen, and D. Spiegelhater. *Probabilistic Networks and Expert Systems*. Springer, 1999. 1.2, 2.2, 3.1

N. Dalvi and D. Suciu. Management of probabilistic data: foundations and challenges. In *Principles of Database Systems (PODS)*, 2007. 2.4, 6, 6.1.2, 9, 6.1.2, 6.1.2

N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proceedings of Very Large Data Bases (PVLDB)*, 2004. 1.2, 1, 2.1, 2.4, 3.2.1, 3.3, 3.4.1, 3.4.2, 4, 6, 6.1.1, 6.1.2, 6.3

A. Darwiche. A logical approach to factoring belief networks. In *International Conference on Principles and Knowledge Representation and Reasoning*, 2002. 2.4, 6

A. Das Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *International Conference on Data Engineering (ICDE)*, Atlanta, Georgia, 2006. 2.1, 4.5

A. Das Sarma, M. Theobald, and J. Widom. Exploiting lineage for confidence computation in uncertain and probabilistic databases. In *International Conference on Data Engineering (ICDE)*, 2008. 2.4

L. De Raedt, A. Kimmig, and H. Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2007. 2.2, 7.3

R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2005. 2.3, 4.5, 5, 7.2

R. de Salvo Braz, E. Amir, and D. Roth. MPE and partial inversion in lifted probabilistic variable elimination. In *AAAI Conference on Artificial Intelligence*, 2006. 2.3

R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Uncertainty in Artificial Intelligence (UAI)*, 1996. 3.3.1, 3.3.3, 4.1

R. Dechter and I. Rish. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM*, 2003. 1.2, 5, 5.4, 5.4.1

A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proceedings of Very Large Data Bases (PVLDB)*, 2004. 1, 1.2, 4.5

A. Deshpande, L. Getoor, and P. Sen. Graphical models for uncertain data. In C. Aggarwal, editor, *Managing and Mining Uncertain Data*. Springer, 2008. 2.1

A. Dovier, C. Piazza, and A. Policriti. A fast bisimulation algorithm. In *International Conference on Computer Aided Verification*, Paris, France, 2001. 4.2.3, 4.2.4, 4.3

U. Feige. A threshold of $ln(n)$ for approximating set cover. *Journal of the ACM*, 1998. 5.3

B. Frey. Extending factor graphs so as to unify directed and undirected graphical models. In *Uncertainty in Artificial Intelligence (UAI)*, 2003. 2.2

N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 1999. 2.1, 2.2

N. Fuhr and T. Rolleke. A probabilistic NF2 relational algebra for integrated information retrieval and database systems. In *World Conference on Integrated Design and Process Technology*, 1996. 2.1

N. Fuhr and T. Rolleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 15(1):32–66, 1997. 1.2, 2.1, 2.4

M. Garey and D. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness*. W. H. Freeman, 1979. 5.3

L. Getoor and B. Taskar, editors. *Introduction to Statistical Relational Learning*. MIT Press, 2007. 2.2

L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *International Conference on Management of Data (SIGMOD)*, 2001. 4.5

L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of link structure. *Journal of Machine Learning Research*, 3:679–707, 2002. 4.4.2

V. Giakoumakis, F. Roussel, and H. Thuillier. On P4-tidy graphs. *Discrete Mathematics and Theoretical Computer Science*, 1997. 7.2

C. Giles, K. Bollacker, and S. Lawrence. Citeseer: An automatic indexing system. *ACM Digital Libraries*, 1998a. 5.5.5

C. L. Giles, K. Bollacker, and S. Lawrence. Citeseer: An automatic citation indexing system. In *Conference on Digital Libraries*, 1998b. 3.4.1

M. Golumbic, A. Mintz, and U. Rotics. Factoring and recognition of read-once functions using cographs and normality and the readability of functions associated with partial k-trees. *Discrete Applied Mathematics*, 2006. 2.4, 6, 6.1.1, 6.1.1, 6.1.1, 6.2.1

M. Habib and C. Paul. A simple linear time algorithm for cograph recognition. *Discrete Applied Mathematics*, 2005. 6.2.1

J. Halpern. An analysis of first-order logics for reasoning about probability. *Artificial Intelligence*, 1990. 1.2

J. Hayes. The fanout structure of switching functions. *Journal of the ACM*, 1975. 6, 8

D. Hochbaum and W. Maass. Approximation schemes for covering and packing problems in image processing and VLSI. *Journal of the ACM*, 1985. 5.3

C. Huang and A. Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1994. 3.3.1, 3.3.3, 4.1, 4.2

T. Imielinski and W. Lipski, Jr. Incomplete information in relational databases. *Journal of the ACM*, 1984. 2.1

A. Jaimovich, O. Meshi, and N. Friedman. Template based inference in symmetric relational markov random fields. In *Uncertainty in Artificial Intelligence (UAI)*, 2007. 2.3

R. Jampani, F. Xu, M. Wu, L. Perez, C. Jermaine, and P. Haas. MCDB: A monte carlo approach to managing uncertain data. In *International Conference on Management of Data (SIGMOD)*, 2008. 6

T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. In *IEEE Data Engineering Bulletin*, 2006. 1

B. Kanagal and A. Deshpande. Indexing correlated probabilistic databases. In *International Conference on Management of Data (SIGMOD)*, 2009. 2.4

P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. In *ACM Symposium on Principles of Distributed Computing (PODC)*, Montreal, Canada, 1983. 1.2, 2.3, 5.2

O. Kariv and S. Hakimi. An algorithmic approach to network location problems I: The p-centers. *SIAM Journal on Applied Mathematics*, 1979. 5.3

K. Kersting, B. Ahmadi, and S. Natarajan. Counting belief propagation. In *Uncertainty in Artificial Intelligence (UAI)*, 2009. 2.3

U. Kjaerulff. Triangulation of graphs - algorithms giving small total state space. Technical report, University of Aalborg, Denmark, 1990. 4.3

C. Koch and D. Olteanu. Conditioning probabilistic databases. In *Proceedings of Very Large Data Bases (PVLDB)*, 2008. 2.1, 2.4

F. Kschischang, B. Frey, and H. andrea Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 2001. 2.3

L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *ACM Transactions on Data Base Systems*, 1997. 2.1

S. Lauritzen. *Graphical Models*. Oxford University Press, 1996. 2.2

J. Li and A. Deshpande. Consensus answers for queries over probabilistic databases. In *Principles of Database Systems (PODS)*, 2009. 2.1

A. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval Journal*, 2000. 5.5.5

B. Milch, B. Marthi, S. J. Russell, D. Sontag, D. L. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2005. 2.2

B. Milch, L. Zettlemoyer, K. Kersting, M. Haimes, and L. Kaelbling. Lifted probabilistic inference with counting formulas. In *AAAI Conference on Artificial Intelligence*, 2008. 5, 7.2

D. Olteanu and J. Huang. Secondary-storage confidence computation for conjunctive queries with inequalities. In *International Conference on Management of Data (SIGMOD)*, 2009. 2.4, 3.4.2, ∗

D. Olteanu and J. Huang. Using OBDDs for efficient query evaluation on probabilistic databases. In *Scalable Uncertainty Management*, 2008. 2.4, 3.4.2, ∗, 6.3

R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987. 2.3, 4.2.3, 4.3

J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988. 1.2, 2.2, 2.4, 3.1, ∗

A. Pfeffer, D. Koller, B. Milch, and K. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Uncertainty in Artificial Intelligence (UAI)*, Stockholm, Sweden, 1999. 5

D. Poole. First-order prob. inference. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003. 2.3, 2.4, 4.5, 5

C. Re and D. Suciu. Materialized views in probabilistic databases for information exchange and query optimization. In *Proceedings of Very Large Data Bases (PVLDB)*, Vienna, Austria, 2007. 4, 4.5

C. Re and D. Suciu. Approximate lineage for probabilistic databases. In *Proceedings of Very Large Data Bases (PVLDB)*, Auckland, New Zealand, 2008. ‡

C. Re, N. Dalvi, and D. Suciu. Query evaluation on probabilistic databases. *IEEE Data Engineering Bulletin Special Issue on Probabilistic Data Management*, 2006. 2.1, 4.5

C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *International Conference on Data Engineering (ICDE)*, 2007. 2.4, 6

M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1-2):107–136, 2006. 2.1, 2.2

T. Richardson. A characterization of markov equivalence for directed cyclic graphs. *International Journal of Approximate Reasoning*, 1997. 2.2

I. Rish. *Efficient Reasoning in Graphical Models*. PhD thesis, University of California, Irvine, 1999. 3.3.2

P. Sen and A. Deshpande. Representing and querying correlated tuples in prob. databases. In *International Conference on Data Engineering (ICDE)*, 2007. 1.3, 2.1

P. Sen, A. Deshpande, and L. Getoor. Representing tuple and attribute uncertainty in probabilistic databases. In *ICDM Workshop on Data Mining of Uncertain Data*, 2007. 1.3, 2.1

P. Sen, A. Deshpande, and L. Getoor. Exploiting shared correlations in probabilistic databases. *Proceedings of Very Large Data Bases (PVLDB)*, 1(1): 809–820, 2008a. 1.3

P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad. Collective classification in network data. *AI Magazine*, 29(3), 2008b. 5.5.5

P. Sen, A. Deshpande, and L. Getoor. Bisimulation-based approximate lifted inference. In *Uncertainty in Artificial Intelligence (UAI)*, 2009a. 1.3

P. Sen, A. Deshpande, and L. Getoor. PrDB: Managing and exploiting shared correlations in probabilistic databases. *Special issue on Uncertain and Probabilistic Databases, VLDB Journal*, 2009b. 1.3, 2.1, *

S. Singh, C. Mayfield, S. Prabhakar, S. Hambrusch, and R. Shah. Indexing uncertain categorical data. In *International Conference on Data Engineering (ICDE)*, 2007. 2.4

P. Singla and P. Domingos. Lifted first-order belief propagation. In *AAAI Conference on Artificial Intelligence*, 2008. 2.3, 5

B. Taskar, P. Abbeel, and D. Koller. Discriminative probabilistic models for relational data. In *Uncertainty in Artificial Intelligence (UAI)*, Edmonton, Canada, 2002. 2.2

TPC-H Benchmark. http://www.tpc.org/tpch/. 3.4

E. Ukkonen. Approximate string matching with q-grams and maximal matches. In *Theoretical Computer Science*, 1992. 3.4.1

V. Vazirani. *Approximation Algorithms*. Springer, 2001. 5.3

D. Wang, E. Michelakis, M. Garofalakis, and J. Hellerstein. BayesStore: Managing large, uncertain data repositories with prob. graphical models. In *Proceedings of Very Large Data Bases (PVLDB)*, 2008. 2.3, 2.4

J. Yedidia, W. Freeman, and Y. Weiss. Generalized belief propagation. In *Neural Information Processing Systems Conference (NIPS)*, 2000. 2.3

N. Zhang and D. Poole. A simple approach to bayesian network computations. In *Canadian Conference on Artificial Intelligence*, Banff, Canada, 1994. 2.3, 2.4, 3.3.1, 3.3.3, 4.1, 4.2, 4.4, 5.4.1, 5.5, 5.6

N. Zhang and D. Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Intelligence Research*, 1996. 3.3.2