Aggregation and Constraint Processing in Lifted Probabilistic Inference

by

Jacek Jerzy Kisyński

M.Sc., Maria Curie-Skłodowska University in Lublin, 2001

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

March 2010

Abstract

Representations that mix graphical models and first-order logic—called either first-order or relational probabilistic models—were proposed nearly twenty years ago and many more have since emerged. In these models, random variables are parameterized by logical variables.

One way to perform inference in first-order models is to *propositionalize* the model, that is, to explicitly consider every element from the domains of logical variables. This approach might be intractable even for simple first-order models. The idea behind *lifted inference* is to carry out as much inference as possible without propositionalizing.

An exact lifted inference procedure for first-order probabilistic models was developed by Poole [2003] and later extended to a broader range of problems by de Salvo Braz et al. [2007]. The C-FOVE algorithm by Milch et al. [2008] expanded the scope of lifted inference and is currently the state of the art in exact lifted inference.

In this thesis we address two problems related to lifted inference: aggregation in directed first-order probabilistic models and constraint processing during lifted inference.

Recent work on exact lifted inference focused on undirected models. Directed first-order probabilistic models require an aggregation operator when a parent random variable is parameterized by logical variables that are not present in a child random variable. We introduce a new data structure, aggregation parfactors, to describe aggregation in directed first-order models. We show how to extend the C-FOVE algorithm to perform lifted inference in the presence of aggregation parfactors. There are cases where the polynomial time complexity (in the domain size

of logical variables) of the C-FOVE algorithm can be reduced to logarithmic time complexity using aggregation parfactors.

First-order models typically contain constraints on logical variables. Constraints are important for capturing knowledge regarding particular individuals. However, the impact of constraint processing on computational efficiency of lifted inference has been largely overlooked. In this thesis we develop an efficient algorithm for counting the number of solutions to the constraint satisfaction problems encountered during lifted inference. We also compare, both theoretically and empirically, different ways of handling constraints during lifted inference.

Table of Contents

Abstract . . . . . **ii**

Table of Contents . . . . . **iv**

List of Figures . . . . . **viii**

Acknowledgments . . . . . **x**

1 Introduction **1**

1.1 Probabilistic reasoning in complex domains 1

1.2 Thesis overview 4

1.3 Summary of thesis contributions 5

1.4 Thesis organization 5

2 Background **6**

2.1 Introduction 6

2.2 Belief networks 6

2.3 Inference in belief networks 7

2.3.1 Factors 8

2.3.2 Variable elimination for belief networks 9

2.3.2.1 Complexity of variable elimination 11

2.4 First-order probabilistic models 12

2.4.1 Parameterized random variables 13

2.4.1.1 Counting formulas 15

2.4.2 Independent Choice Logic 16

2.5 Lifted probabilistic inference . . . 21
2.5.1 Parametric factors . . . 22
2.5.1.1 Normal-form constraints . . . 24
2.5.2 C-FOVE . . . 25
2.5.2.1 Lifted elimination . . . 26
2.5.2.2 Parfactor multiplication . . . 31
2.5.2.3 Splitting, expanding and propositionalizing . . . 33
2.5.2.4 Counting . . . 36
2.5.2.5 Unification . . . 38
2.5.2.6 The C-FOVE algorithm . . . 46
2.5.2.7 Example computation . . . 48
2.6 Summary . . . 54

3 Aggregation in Lifted Inference . . . 56
3.1 Introduction . . . 56
3.2 Need for aggregation . . . 57
3.3 Modeling aggregation . . . 58
3.3.1 Causal independence . . . 58
3.3.2 Causal independence-based aggregation . . . 60
3.4 Aggregation parfactors . . . 63
3.4.1 Conversion to parfactors . . . 65
3.4.1.1 Conversion using counting formulas . . . 66
3.4.1.2 Conversion for MAX and MIN operators . . . 69
3.4.2 Operations on aggregation parfactors . . . 76
3.4.2.1 Splitting . . . 76
3.4.2.2 Multiplication . . . 85
3.4.2.3 Summing out . . . 87
3.4.3 Generalized aggregation parfactors . . . 92
3.5 Experiments . . . 97
3.5.1 Memory usage . . . 98
3.5.2 Social network experiment . . . 101
3.6 Conclusions . . . 104

4 Solver for #CSP with Inequality Constraints 105
4.1 Introduction 105
4.2 Background 106
4.2.1 Constraint satisfaction problems 108
4.2.2 Variable elimination for #CSP 109
4.2.3 Set partitions 111
4.3 Counting solutions to CSP instances with inequality constraints . . 113
4.3.1 Analysis of the problem 113
4.3.2 The #VE$_{\neq}$ algorithm 117
4.3.2.1 S-constants 118
4.3.2.2 #VE$_{\neq}$ factors 118
4.3.2.3 Multiplication 121
4.3.2.4 Summing out 126
4.3.2.5 The algorithm 129
4.3.3 Example computation 131
4.3.4 Complexity of the algorithm 134
4.3.4.1 Preprocessing 134
4.3.4.2 Inference 137
4.3.5 Empirical evaluation 138
4.4 Conclusions 143

5 Constraint Processing in Lifted Inference 144
5.1 Introduction 144
5.2 Overview of constraint processing in lifted inference 145
5.2.1 Splitting and expanding 147
5.2.2 Multiplication 149
5.2.3 Summing out 150
5.3 Splitting as needed vs. shattering 153
5.4 Normal form parfactors vs. #CSP solver 159
5.4.1 Multiplication 165
5.4.2 Summing out 166
5.4.3 Experiment 167
5.5 Conclusions 168

6 Conclusions 169
6.1 Summary 169
6.2 Future work 170

Bibliography 172

A 1-dimensional Representation of VE Factors 179

B Hierarchical Representation of #VE$_{\neq}$ Factors 182

C From Parfactors to #VE$_{\neq}$ Factors 185

D From #VE$_{\neq}$ Factors to Parfactors 188

E Splitting as Needed 192

List of Figures

1.1 Lifted inference vs propositional inference 3

2.1 A simple belief network 7
2.2 VE algorithm for inference in belief networks 10
2.3 A graph and its induced graph 11
2.4 ICL theory from Example 2.6 18
2.5 ICL theory for multiple lots from Example 2.6 19
2.6 More elegant ICL theory for multiple lots from Example 2.6 . . . 19
2.7 Graphical representation of the ICL theory for multiple lots 20
2.8 AC-3 algorithm for replacing logical variables with constants . . 39
2.9 MGU algorithm for parameterized random variables 41
2.10 Algorithm for checking if an MGU is consistent with constraints . 42
2.11 Algorithm for splitting a parfactor on an MGU 43
2.12 Algorithm for splitting a parfactor on a set of constraints 44

3.1 A first-order model from Example 3.1 57
3.2 A first-order model from Example 3.2 58
3.3 A first-order model with OR-based aggregation 60
3.4 A first-order model with MAX-based aggregation 61
3.5 A first-order model from Example 3.5 62
3.6 Decomposed aggregation 87
3.7 A first-order model from Example 3.14 92
3.8 Results of the experiment 99
3.9 Performance on model (a) (with OR-based aggregation) 99
3.10 Performance on model (b) (with MAX-based aggregation) 100
3.11 Performance on model (c) (with $\mathrm{SUM}_{|3}$-based aggregation) 100

3.12 Performance on model (d) (generalized aggregation parfactors) . . 101
3.13 ICL theory for the smoking-friendship model 102
3.14 Illustration of how ind(X) works 103
3.15 Performance on the smoking-friendship model 104

4.1 #VE algorithm for #CSP from Dechter [2003] 110
4.2 Comparison of ϖ_n and exponential functions 112
4.3 Constraint graph with tree structure 113
4.4 Constraint graph with a cycle 114
4.5 Constraint graph discussed in Example 4.5 115
4.6 Constraint graph discussed in Example 4.5 115
4.7 Relationship between #VE$_{\neq}$ and #VE 117
4.8 A CSP used in Examples 4.11–4.15 121
4.9 #VE$_{\neq}$ algorithm for #CSP with inequality constraints 130
4.10 Initialization procedure for the #VE$_{\neq}$ algorithm 131
4.11 A CSP used in Section 4.3.3 131
4.12 Domains of three variables represented with disjoint subsets . . . 135
4.13 Summary of experiments 140
4.14 Summary of experiments 140
4.15 Results of experiments on CSP instances with $\mathcal{P}(u) = 0.1$ 141
4.16 Results of experiments on CSP instances with $\mathcal{P}(u) = 0.01$ 142

5.1 A first-order model from Example 5.2 147
5.2 Speedup of splitting as needed over shattering for Example 5.8 . . 157
5.3 Splitting as needed vs. shattering speedup for Example 5.9 159
5.4 Algorithm for converting a parfactor to normal form 162
5.5 A simple constraint graph from Example 5.10 163
5.6 Summing out with and without a #CSP solver 168

A.1 Structure of a VE factor 180
A.2 Procedure indexToTuple(f,t) 181

B.1 Domains of three variables represented with disjoint subsets . . . 183
B.2 Structure of a #VE$_{\neq}$ factor 183

Acknowledgments

I am grateful to Dr. David Poole, my research supervisor, for his guidance. I could always rely on his expertise in artificial intelligence, computer science and science in general.

I thank Dr. Nando de Freitas, Dr. William Evans, and Dr. Kevin Murphy, members of my supervisory committee, for their advice and feedback throughout the course of my work.

I would like to thank Dr. Dan Roth (External Examiner), Dr. Martin Puterman (University Examiner), Dr. Kevin Leyton-Brown (University Examiner), and Dr. Vijay Bhargava (Examination Chair) for the time they spent reading and reviewing my thesis.

The Laboratory for Computational Intelligence and the Department of Computer Science at The University of British Columbia provided me with an inspiring work environment. I thank Peter Carbonetto and Mike Chiang for their work on joint projects. In addition to my thesis research I had the opportunity to participate in the AIspace project. My collaboration with Dr. David Poole, Dr. Alan Mackworth, Dr. Giuseppe Carenini, Dr. Cristina Conati, Byron Knoll, and Kyle Porter was a valuable experience.

The UBC Department of Computer Science is also a great place to make friends. My lab-mates Asher, Andrea, Dustin, Eric, Firas, Kasia, Mark, Mike, Rita, Robert, and Scott, and, in particular, my officemates Matt, Hendrik, and Peter were very tolerant of my passion for chitchat.

Vancouver is a paradise for a skiing and swimming enthusiast. Vast slopes of Whistler and Blackcomb mountains offer a seven-months-long skiing season. Together with my riding buddies Ken, Clint, Reid, Jonathan, and Lowell I took full

advantage of it. The Empire Pool at The UBC Aquatic Centre is open seven months per year for outdoor swimming. I took full advantage of it as well. My friendship with Gośka, Kris, and Sam started there. The duration of my Ph.D. studies can easily be explained by the fact that on occasion research had to compete with skiing and swimming.

Ever since I arrived in Vancouver I was lucky to have great landlords and roommates. Andy and Dave care more about well-being of their tenants than about collecting the rent. Tom, Mike, Wini, Manos, Adam and Maher were cool roommates and we had a good time living together.

I would not be able to enter a Ph.D. program without the education I received in Poland. My high school math teacher, Waldemar "Byko" Łobodziński, not only taught me mathematics, but also respect for knowledge and contempt for ignorance. Dr. Tadeusz Kuczumow's courses in calculus and measure theory were of world-class quality. Dr. Jerzy Mycka, my M.Sc. thesis supervisor, introduced me to computability theory as well as to the art of logic and functional programming.

I am also indebted to my family. My aunt Anne and uncle Andrzej encouraged me to pursue studies abroad and provided support once I arrived in Canada. My wife Sylwia has made a lot sacrifices to help me with my studies. People to whom I owe the most are my parents, Magdalena and Jan Kisyńscy. I would like to thank them for everything they have done for me.

Chapter 1

Introduction

I could never bear to be buried with people to whom I had not been introduced. — Norman Parkinson

1.1 Probabilistic reasoning in complex domains

Artificial intelligence studies the design and synthesis of agents that act intelligently and are rational [Poole and Mackworth, 2010; Russell and Norvig, 2009]. Any agent acting in the real world faces uncertainty. Uncertainty arises because of limited information available to an agent: an agent's observations might be incomplete, an agent's sensor might be noisy, and the effects of an agent's own actions might be unknown to an agent.

The design of representation and reasoning systems for agents is a core area of artificial intelligence. These systems are used to model and make predictions about the world and to support decision making. Unavoidably, they must handle uncertainty.

Probability theory provides a foundation for representation and reasoning systems that can reason under uncertainty. Given a model of the world, a usual probabilistic inference task is to make predictions about the value of a random variable given evidence bout the value of other random variables.

While reasoning about the real world, an agent has to deal with domains that involve a large number of individuals. It might be interested only in a few of them,

but it cannot ignore the influence of other objects and individuals. Poole [2003] gives the following example of a domain involving a large number of individuals.

We are given a description of a person who committed a crime in a town. We also happen to know that a guy named Marian roughly matches the description. What is the probability that Marian is guilty? The probability depends on how well Marian matches the description. It also depends on the rest of the population of the town. If the town is a small village, Marian is likely to be guilty. If the town is a large city, there are potentially many people living in the city who match the description and Marian is likely to be innocent. To represent this problem and compute the probability of Marian being guilty we need to reason about Marian, but we also have to represent and reason about a potentially large number of individuals about whom we do not have any specific information.

Probabilistic graphical models, such as belief networks or Markov networks [Koller and Friedman, 2009], are a popular tool for representing dependencies between random variables. However, such standard representations are propositional (zeroth-order), and therefore are not well suited for describing relations between individuals or quantifying over sets of individuals. In order to reason about multiple individuals, we typically make each property of each individual into a separate random variable. Even the relatively simple example domain described above could not be easily represented using probabilistic graphical models.

First-order logic has the capacity for representing relations and quantification of logical variables, but it does not treat uncertainty. It has only a very primitive mechanism for handling uncertainty, namely disjunction and existential quantification.

Representations that mix graphical models and first-order logic—called either first-order or relational probabilistic models—were proposed nearly twenty years ago [Breese, 1992; Horsch and Poole, 1990] and many more have since emerged [De Raedt et al., 2008; Getoor and Taskar, 2007]. In these models, random variables are parameterized by logical variables that are typed with populations of individuals. This allows a model to be represented before modeled individuals are known, or even before their numbers are known. It also allows an agent to compactly represent the same information about multiple individuals, and to exploit this compactness facilitate efficient inference.

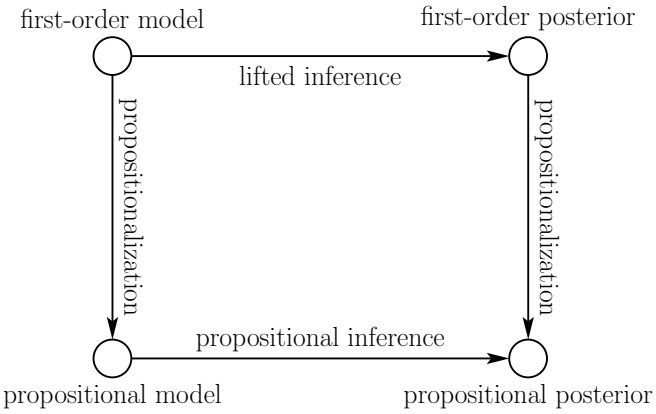


Figure 1.1: Lifted inference vs propositional inference for first-order probabilistic models

A popular exact inference technique in first-order probabilistic models is based on dynamical propositionalization of the portion of the model that is relevant to the query, followed by probabilistic inference performed at the propositional level. Unfortunately, even for simple relational probabilistic models, inference at the propositional level—that is, inference that explicitly considers every individual—is very often intractable.

The idea of *lifted probabilistic inference* is to carry out as much inference as possible without propositionalizing. The correctness of this approach is judged by having the same result as if we had first propositionalized the model and then carried out standard probabilistic inference (see Figure 1.1). An exact lifted probabilistic inference procedure for first-order probabilistic directed models was proposed in Poole [2003]. It was later extended to a broader range of problems by de Salvo Braz et al. [2005, 2006, 2007]. Further work by Milch et al. [2008] expanded the scope of lifted probabilistic inference and resulted in the C-FOVE algorithm, which is currently the state of the art in exact lifted probabilistic inference.

In this thesis we address two problems related to exact lifted probabilistic inference. The first one is the efficient aggregation during lifted probabilistic inference in directed relational probabilistic models. The second one is constraint process-

ing during lifted probabilistic inference in both directed and undirected relational probabilistic models.

1.2 Thesis overview

While early work on lifted probabilistic inference by Poole [2003] considered directed models, later work by de Salvo Braz et al. [2007] and Milch et al. [2008] focused on undirected models. Although their results can also be used for directed models, one aspect that arises exclusively in directed models is the need for aggregation that occurs when a parent random variable is parameterized by logical variables that are not present in a child random variable. Currently available lifted inference algorithms do not represent aggregation in relational probabilistic models using data-structures that are independent of sizes of populations. In this thesis we introduce a new data structure, *aggregation parfactor* and describe how to use it to represent aggregation in first-order probabilistic models. We also show how to perform lifted probabilistic inference in presence of aggregation parfactors, by integrating it into the C-FOVE algorithm. Results of our theoretical and empirical evaluations show that inference with aggregation parfactors can lead to gains in efficiency.

First-order models typically contain constraints on logical variables. Constraints are important for capturing knowledge regarding particular individuals. Constraint processing during lifted probabilistic inference includes counting the number of solutions to constraint satisfaction problems induced during the inference (this counting problem is written as #CSP). In this thesis we present an algorithm for solving #CSPs encountered during lifted probabilistic inference and through empirical evaluation show that it significantly improves the efficiency of the inference.

Previous works on lifted probabilistic inference adopted various approaches to constraint processing. The impact of these different strategies on computational efficiency of lifted inference has been largely overlooked. In this thesis we analyze constraint processing during lifted probabilistic inference, both theoretically and empirically. Our results stress the importance of informed constraint processing in lifted inference and motivate our work on a specialized #CSP solver.

Although this thesis focuses on exact lifted probabilistic inference, our results in the area of constraint processing for lifted probabilistic inference apply to research on approximate lifted probabilistic inference, for example to work by Singla and Domingos [2008].

1.3 Summary of thesis contributions

The contributions of this thesis are as follows:

- efficient aggregation algorithms for lifted probabilistic inference
- specialized algorithm for #CSPs that greatly improves the efficiency of lifted probabilistic inference
- analysis of constraint processing in lifted probabilistic inference.

1.4 Thesis organization

The rest of this thesis is organized as follows. In Chapter 2 we provide the necessary background for the topics covered in this thesis; in particular, we describe existing exact lifted probabilistic inference techniques. The contributions of this thesis begin in Chapter 3, where we present algorithms for lifted aggregation in directed first-order probabilistic models. In Chapter 4 we develop an efficient algorithm for solving #CSPs encountered during lifted probabilistic inference. Next, in Chapter 5, we analyze various approaches to constraints processing in lifted probabilistic inference. Finally, in Chapter 6, we summarize the contributions of the thesis and discuss possible future work.

Some of the parts of this work have been published as technical papers in AI conferences [Kisyński and Poole, 2009a,b].

Chapter 2

Background

No matter how hard a man may labor, some woman is always in the background of his mind. — Gertrude Franklin Atherton

2.1 Introduction

In this chapter we introduce notation and concepts used throughout the thesis. First, we describe belief networks (Section 2.2) and the variable elimination algorithm (Section 2.3), which is used to perform inference in belief networks. Next, we give an overview of first-order probabilistic models (Section 2.4). We describe in more detail an example of first-order probabilistic modeling, Independent Choice Logic (Section 2.4.2). Finally, we discuss currently available exact lifted probabilistic inference methods (Section 2.5).

2.2 Belief networks

One of the areas of artificial intelligence is the design of representation and reasoning formalisms. Belief networks (also known as Bayesian networks) were proposed by Pearl [1988] and have since became a popular representation for independence among random variables. They are a member of class of models known as *probabilistic graphical models*.

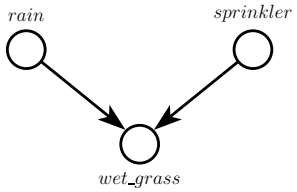


Figure 2.1: A simple belief network.

In the definition below and in the rest of this chapter we use lower case letters for random variables and upper case letters for logical variables. It is not a standard notation, but it helps to distinguish logical variables from random variables.

Definition 2.1. A *belief network* over a set of random variables $\{x_1, x_2, \ldots, x_n\}$ consist of a acyclic directed graph over random variables $x_1, x_2, \ldots, x_n$, with one node of each random variable, and a set of conditional probability distributions $\{\mathcal{P}(x_i | \, parent(x_i)) \, | i = 1, 2, \ldots, n\}$, where $parent(x_i)$ denotes the nodes in the associated graph that have directed edges going into x_i. A belief network represents the joint probability over random variables $x_1, x_2, \ldots, x_n$:

$$\mathcal{P}(x_1, x_2, \ldots, x_i) = \prod_{i=1}^{n} \mathcal{P}(x_i | \, parent(x_i)) \, .$$

Example 2.1. Figure 2.1 presents a simple belief network over three random variables *rain*, *sprinkler*, and *wet_grass*. Assume they all have domain $\{false, true\}$. Associated with the belief network are three probability distributions: $\mathcal{P}(rain)$, $\mathcal{P}(sprinkler)$ and $\mathcal{P}(wet\_grass | rain, sprinkler)$. The distributions could, for instance, encode that rain increases probability of grass being wet and so does running sprinkler (see Example 2.2).

2.3 Inference in belief networks

Given a belief network, a common inference problem is to compute the posterior distribution over a set of random variables given some evidence. We will describe how to solve this problem with the variable elimination (VE) algorithm [Zhang and Poole, 1994]. The aim of VE is to obtain the global solution in an efficient manner

through local computations. The variable elimination algorithm uses *factors* to represent the input problem instance, the results of intermediate local computations and the final solution.

2.3.1 Factors

Let $dom(x)$ denote the domain of random variable x.

A *factor* on random variables $x_1, x_2, \ldots, x_n$ is a representation of a function from $dom(x_1) \times dom(x_2) \times \cdots \times dom(x_n)$ into the real numbers.

Let $\mathbf{v}$ denote an *assignment of values* to some set of random variables; $\mathbf{v}$ is a function that takes a random variable and returns its value.

Let $\mathcal{F}$ be a factor on a set of random variables $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$. Let $\mathbf{v}$ be an assignment of values to random variables in S. We extend $\mathbf{v}$ to factors and denote by $\mathbf{v}(\mathcal{F})$ the value of the factor $\mathcal{F}$ given $\mathbf{v}$, that is

$$\mathbf{v}(\mathcal{F}) = \mathcal{F}(\mathbf{v}(x_1), \mathbf{v}(x_2), \ldots, \mathbf{v}(x_n)).$$

If $\mathbf{v}$ does assign values only to some of the random variables in $\mathcal{S}$, then $\mathbf{v}(\mathcal{F})$ denotes a factor on other random variables.

Example 2.2. Consider a belief network from Example 2.1 and Figure 2.1. The three probability distributions could be represented by the following three factors:

$\mathcal{P}(rain = false)$	$\mathcal{P}(rain = true)$
0.8	0.2

,

$\mathcal{P}(sprinkler = false)$	$\mathcal{P}(sprinkler = true)$
0.6	0.4

,

$rain$	$sprinkler$	$\mathcal{P}(wet\_grass = false)$	$\mathcal{P}(wet\_grass = true)$
$false$	$false$	1.0	0.0
$false$	$true$	0.2	0.8
$true$	$false$	0.1	0.9
$true$	$true$	0.01	0.99

.

Operations on factors include computing a product of factors and summing out random variables from a factor.

Suppose $\mathcal{F}_1$ is a factor on random variables $x_1,\ldots,x_i,y_1,\ldots,y_j$, and $\mathcal{F}_2$ is a factor on random variables $y_1,\ldots,y_j,z_1,\ldots,z_l$, where sets $\{x_1,\ldots,x_i\}$, $\{y_1,\ldots,y_j\}$ and $\{z_1,\ldots,z_l\}$ are pairwise disjoint. Given an assignment of values to random variables $\mathbf{v}$, the *product* of $\mathcal{F}_1$ and $\mathcal{F}_2$ is a factor $\mathcal{F}_1 \odot \mathcal{F}_2$ on the union of the random variables, namely $x_1,\ldots,x_i,y_1,\ldots,y_j,z_1,\ldots,z_l$, defined by:

$$(\mathcal{F}_1 \odot \mathcal{F}_2)(x_1,\ldots,x_i,y_1,\ldots,y_j,z_1,\ldots,z_l) = \\ \mathcal{F}_1(x_1,\ldots,x_i,y_1,\ldots,y_j)\cdot\mathcal{F}_2(y_1,\ldots,y_j,z_1,\ldots,z_l). \tag{2.1}$$

Suppose $\mathcal{F}$ is a factor on random variables $x_1,\ldots,x_i,\ldots,x_j$. The *summing out* of random variable x_i from $\mathcal{F}$, denoted as $\sum_{x_i}\mathcal{F}$ is the factor on random variables $x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_j$ such that

$$\left(\sum_{x_i}\mathcal{F}\right)(x_1,\ldots,x_{i-1},x_{i+1},\ldots,x_j) = \sum_{x\in dom(x_i)}\mathcal{F}(x_1,\ldots,x_{i-1},x_i=x,x_{i+1},\ldots,x_j).$$

Given a total ordering of the random variables and total ordering of their domains, factors can be implemented as 1-dimensional arrays (see Appendix A).

2.3.2 Variable elimination for belief networks

Variable elimination (VE) is a general technique that can solve many problems, such as, CSP [Dechter, 1999] or belief network inference [Zhang and Poole, 1994]. The first VE algorithm was a non-serial dynamic programming [Bertelè and Brioschi, 1972]. In this section we present a variant of VE for inference in belief networks.

Consider a belief network over a set of random variables $V = \{x_1, x_2, \ldots, x_n\}$. We represent conditional probability distributions from the network as factors. For each random variable $x_i \in V$, we represent the conditional probability distribution $\mathcal{P}(x_i|\, parent(x_i))$ as a factor $\mathcal{F}_{x_i}$ on a set of random variables $\{x_i\}\cup parent(x_i)$.

Assume that we are given:

- a set of observations of values on a set of random variables $O \subset V$, which we can think of as an assignment $\mathbf{v}_O$ of values to random variables O;
- a query random variable $q \in V \setminus O$.

```
[00] procedure VE_BN(V, F, v_O, q, H)
[01]   input: set of random variables V,
[02]          set of factors F,
[03]          assignment of values to random variables O ⊂ V, v_O,
[04]          query random variable q ∈ V \ O,
[05]          elimination ordering heuristic H;
[06]   output: factor F representing the posterior distribution on q;
[07]   set E := V \ (O ∪ {q});
[08]   while there is a factor in F involving a random variable from E do
[09]     select random variable y ∈ E according to H;
[10]     set F := eliminate(y, F);
[11]     set E := E \ {y};
[12]   end
[13]   set F := ⊙_{F_i ∈ F} F_i;
[14]   compute normalizing constant c := Σ_q F;
[15]   return F / c;
[16] end

[17] procedure eliminate(y, F)
[18]   input: random variable to be eliminated y,
[19]          set of factors F;
[20]   output: set of factors F with y summed out;
[21]   partition F into set of factors on y, F_y and set F_{-y} := F \ F_y;
[22]   return F_{-y} ∪ {Σ_y ⊙_{F_i ∈ F_y} F_i};
[23] end
```

Figure 2.2: VE algorithm for inference in belief networks.

Let $V \setminus (O \cup \{q\}) = \{x_{s_1}, x_{s_2}, \ldots, x_{s_m}\}$. We want to compute the posterior distribution on q:

$$\sum_{x_{s_1}} \sum_{x_{s_2}} \cdots \sum_{x_{sm}} \mathbf{v}_O(\mathcal{F}_{x_1}) \odot \mathbf{v}_O(\mathcal{F}_{x_2}) \odot \cdots \odot \mathbf{v}_O(\mathcal{F}_{x_n}).$$

Computing the product $\mathbf{v}_O(\mathcal{F}_{x_1}) \odot \mathbf{v}_O(\mathcal{F}_{x_2}) \odot \cdots \odot \mathbf{v}_O(\mathcal{F}_{x_n})$ is usually not tractable. Instead, we take advantage of the (possible) sparseness of the associated graph. First, we order sums according to an *elimination ordering* ρ:

$$\sum_{x_{\rho(m)}} \cdots \sum_{x_{\rho(2)}} \sum_{x_{\rho(1)}} \mathbf{v}_O(\mathcal{F}_{x_1}) \odot \mathbf{v}_O(\mathcal{F}_{x_2}) \odot \cdots \odot \mathbf{v}_O(\mathcal{F}_{x_n}).$$

Next, we use the distribution law, distribute factors that are not functions of $x_{\rho(1)}$ outside of the sum $\sum_{x_{\rho(1)}}$, multiply remaining factors and sum out $x_{\rho(1)}$ from their

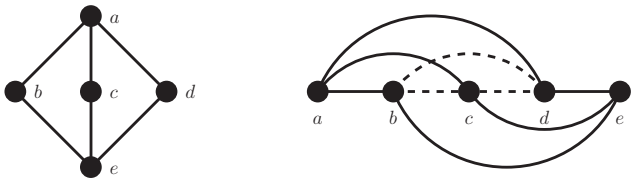


Figure 2.3: A graph (*left*) and its induced graph generated by ordering $\rho = \langle a,b,c,d,e \rangle$ (*right*). Edges added during construction of the induced graph are represented by *dotted lines*. We have $w(\rho) = w^*(\rho) = 3$.

product. We repeat the process for sums $\sum_{x_{\rho(2)}}$ to $\sum_{x_{\rho(m)}}$. The above procedure is the essence of the VE algorithm. Table 2.2 presents the pseudo-code involved.

The next section discusses the impact of an elimination ordering on computational cost of the VE algorithm.

2.3.2.1 Complexity of variable elimination

First, we need to introduce some concepts from the graph theory.

Given an ordered or unordered graph G and an ordering of its nodes ρ, an *ordered graph* is a pair (G,ρ). The *width of a node* is the number of nodes sharing an edge with the node that precede it in the ordering ρ. The *width of the ordering*, $w(\rho)$, is the maximum width over all nodes, and the *width of the graph* is the minimum width over all orderings of the graph.

For the ordered graph (G,ρ), the *induced graph* (G^*,ρ) is constructed in the following way: nodes are traversed in the opposite order to ρ. When a node is visited, we connect all of the nodes sharing an edge with the node that precede it in the ordering ρ. The *induced width of the ordered graph* (G,ρ), $w^*(\rho)$, is the width of the induced ordered graph (G^*,ρ) (Figure 2.3).

Assume that domains of all random variables have the same size $|dom(x_1)| = \cdots = |dom(x_n)| = d$ and that random variables were processed according to ordering ρ. The time and space complexity of the variable elimination algorithm are determined by the size of the biggest factor created during inference, which depends on the induced width $w^*(\rho)$ of the belief network graph

$$\mathcal{O}(nd^{w^*(\rho)}).$$

The problem of finding an optimal elimination ordering, that is an ordering ρ that minimizes $w^*(\rho)$, is NP-hard [Arnborg et al., 1987], so usually an elimination ordering heuristic is used [Kjærulff, 1990].

2.4 First-order probabilistic models

Belief networks and other probabilistic graphical models are widely used for representing dependencies between random variables. However, they are propositional representations. That is, in order to represent information about multiple individuals, each property of each individual is represented as a separate node. Probabilistic graphical models are not well suited for describing relations between individuals or quantifying over sets of individuals. First-order logic has the capacity for representing relations and quantification of logical variables, but it does not handle uncertainty well. Representations that mix graphical models and first-order logic (*first-order probabilistic models*) were proposed nearly twenty years ago [Breese, 1992; Horsch and Poole, 1990]. A common building block of these models is a *parameterized random variable*, a random variable parameterized by logical variables. Logical variables are typed with populations of individuals.

Among the appeals of the first-order probabilistic models is that it is possible to fully specify a model, that is, its structure and the accompanying probability distributions, before knowing the individuals in the modeled domain. This means that, even though we might not know the populations or even their sizes, we still should be able to specify the model. In order to make this possible, the length of a specification of a first-order probabilistic model must be independent of the sizes of the populations in the model.

This thesis is not tied to any particular first-order probabilistic language. We reason at the level of data structures and assume that various first-order languages (or their subsets) will compile to these data structures. As we said above, first-order probabilistic languages share a concept of a parameterized random variable. We introduce it formally in Section 2.4.1. We also describe Independent Choice Logic (Section 2.4.2), an example of a first-order probabilistic language.

2.4.1 Parameterized random variables

A *population* is a set of *individuals*. A population corresponds to a domain in logic.

A logical variable will be written starting with an uppercase letter. A logical variable is typed with a population. Given a logical variable X, we denote its population by $\mathcal{D}(X)$. Given a set of constraints $\mathcal{C}$ on X, we denote the set of individuals from $\mathcal{D}(X)$ that satisfy the constraints in $\mathcal{C}$ by $\mathcal{D}(X) : \mathcal{C}$.

A *term* is a logical variable or a constant denoting an individual from a population.

Definition 2.2. A *parameterized random variable* is of the form $f(t_1,\ldots,t_k)$, where f is a symbol, called a *functor*, and t_i are terms.

Each functor has a set of values called the *range* of the functor. We denote the range of the functor f by $range(f)$. Functors with range $\{false, true\}$ are *predicate* symbols, other are called *function* symbols.

We denote the set of logical variables that appear in the parameterized random variable $f(t_1,\ldots,t_k)$ by $param(f(t_1,\ldots,t_k))$.

A *substitution* to a set of distinct logical variables $\{X_1,\ldots,X_l\}$ is of the form $\{X_1/t_{i_1},\ldots,X_l/t_{i_l}\}$, where each t_{i_j} is a term. A *ground substitution* is a substitution, where each t_{i_j} is a constant.

The *application of a substitution* $\theta = \{X_1/t_{i_1}\ldots,X_l/t_{i_l}\}$ to a parameterized random variable $f(t_1,\ldots,t_k)$, written $f(t_1,\ldots,t_k)[\theta]$, is a parameterized random variable that is the original parameterized random variable $f(t_1,\ldots,t_k)$ with every occurrence of X_j in $f(t_1,\ldots,t_k)$ replaced by the corresponding t_{i_j}. The parameterized random variable $f(t_1,\ldots,t_k)[\theta]$ is called an *instance* of $f(t_1,\ldots,t_k)$. An instance $f(t_1,\ldots,t_k)[\theta]$ that does not contain logical variables is called a *ground instance* of $f(t_1,\ldots,t_k)$.

Let $f(t_1,\ldots,t_k)$ be a parameterized random variable and θ be a ground substitution to all logical variables in $param(f(t_1,\ldots,t_k))$. An application of θ to $f(t_1,\ldots,t_k)$, $f(t_1,\ldots,t_k)[\theta]$ results in a ground instance of $f(t_1,\ldots,t_k)$.

A ground instance of a parameterized random variable is a random variable. A parameterized random variable $f(t_1,\ldots,t_k)$ represents a set of random variables, one random variable for each ground substitution to all logical variables

in $param(f(t_1,\ldots,t_k))$. We denote this set by $ground(f(t_1,\ldots,t_k))$. The domain of each random variables represented by $f(t_1,\ldots,t_k)$ is the range of f.

Given a parameterized random variable $f(t_1,\ldots,t_k)$, a set of constraints $\mathcal{C}$ on $param(f(t_1,\ldots,t_k))$, and its ground instance $f(t_1,\ldots,t_k)[\theta]$, we say that ground instance $f(t_1,\ldots,t_k)[\theta]$ *satisfies the constraints* in $\mathcal{C}$ if substitution θ satisfies the constraints in $\mathcal{C}$. We denote the set of ground instances of $f(t_1,\ldots,t_k)$ that satisfy the constraints in $\mathcal{C}$ by $ground(f(t_1,\ldots,t_k)):\mathcal{C}$.

We extend our notation of an assignment of values to random variables to parameterized random variables. When we say that $\mathbf{v}$ assigns a value to a parameterized random variable $f(t_1,\ldots,t_k)$, we mean that it assigns this value to each random variable in the set $ground(f(t_1,\ldots,t_k))$.

Example 2.3. Let *Lot* be a logical variable typed with a population of all lots in a town $\{lot_1,\ldots,lot_n\}$. We have $|\mathcal{D}(Lot)| = n$. Let $wet\_grass(Lot)$ be a parameterized random variable, where *wet_grass* is a functor with range $\{false,true\}$. Thus, $range(wet\_grass) = \{false,true\}$ and $param(wet\_grass(Lot)) = \{Lot\}$. The parameterized random variable $wet\_grass(Lot)$ represents a set of n random variables, each with domain $\{false,true\}$, one random variable for each substitution $\{Lot/lot_1\},\ldots,\{Lot/lot_n\}$. Let $\mathbf{v}$ be an assignment of values to random variables such that $\mathbf{v}(wet\_grass(Lot)) = true$. It means that each of the random variables in the set $ground(wet\_grass(Lot))$, is assigned the value $true$ by $\mathbf{v}$.

Substitution θ is a *unifier* of two parameterized random variables $f(t_{i_1},\ldots,t_{i_k})$ and $f(t_{j_1},\ldots,t_{j_k})$ if $f(t_{i_1},\ldots,t_{i_k})[\theta] = f(t_{j_1},\ldots,t_{j_k})[\theta]$. We then say that the two parameterized random variables *unify*.

Substitution θ is a *most general unifier* (MGU) of two parameterized random variables if

- θ is a unifier of the two parameterized random variables, and
- if there exists another unifier θ' of the two parameterized random variables, then $f(\ldots)[\theta']$ must be an instance of $f(\ldots)[\theta]$ for all parameterized random variables $f(\ldots)$.

If two parameterized random variables have a unifier, they have at least one MGU. Sets of random variables represented by two parameterized random variables that unify have a non-empty intersection.

Example 2.4. Parameterized random variables $wet\_grass(Lot)$ and $wet\_grass(lot_1)$ have a unifier $\{Lot/lot_1\}$. Let *Parcel* be a logical variable such that $\mathcal{D}(Parcel) = \mathcal{D}(Lot)$. Parameterized random variables $wet\_grass(Lot)$ and $wet\_grass(Parcel)$ have a unifier $\{Lot/Parcel\}$. Let *adjacent* be a binary functor, $adjacent(Lot, lot_1)$ and $adjacent(lot_3, Parcel)$ have a unifier $\{Lot/lot_3, Parcel/lot_1\}$. Parameterized random variables $wet\_grass(lot_1)$ and $wet\_grass(lot_2)$ do not unify. Parameterized random variables $adjacent(Lot, Lot)$ and $adjacent(lot_1, lot_3)$ also do not unify. Finally, parameterized random variables $f(t_{i_1}, \ldots, t_{i_k})$ and $h(t_{j_1}, \ldots, t_{j_l})$ do not unify as they have different functors.

2.4.1.1 Counting formulas

So far, we can specify a value assignment for named individuals, or for all individuals from some population. Sometimes it is useful to be able to say that a certain number of individuals have some assignment, without specifying which individuals. We can achieve this with *counting formulas* [Milch et al., 2008]. Counting formulas were inspired by work on *cardinality potentials* [Gupta et al., 2007] and *counting elimination* [de Salvo Braz et al., 2007].

Definition 2.3. A *counting formula* is of the form $\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]$, where A is a logical variable that is *bound* by the # sign, $\mathcal{C}$ is a set of inequality constraints involving A and $f(\ldots, A, \ldots)$ is a parameterized random variable. The value of $\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]$, given an assignment of values to random variables $\mathbf{v}$, is the *histogram* function $h^{\mathbf{v}} : range(f) \to \mathbb{N}$ defined by

$$\mathbf{v}(\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]) = h^{\mathbf{v}}(x) = |\{a \in (\mathcal{D}(A) : \mathcal{C}) : \mathbf{v}(f(\ldots, a, \ldots)) = x\}|,$$

where x is in the range of f.

Thus, the set of values of the above counting formula is the set of histograms having a bucket for each element x in the range of f with entries adding up to $|\mathcal{D}(A) : \mathcal{C}|$. The number of such histograms is $\binom{|\mathcal{D}(A):\mathcal{C}|+|range(f)|-1}{|range(f)|-1}$, which for small values of $|range(f)|$ is $\mathcal{O}(|\mathcal{D}(A) : \mathcal{C}|^{|range(f)|-1})$. Therefore, a tabular representation of a function on a counting formula $\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]$ requires an

amount of space that is linear in $|\mathcal{D}(A) : \mathcal{C}|$ for functor f with a binary range and increases dramatically with range size.

Example 2.5. Consider a counting formula $\#_{Lot}[wet\_grass(Lot)]$. It has a range $\{(\#_{false} = n, \#_{true} = 0), (\#_{false} = n-1, \#_{true} = 1), \ldots, (\#_{false} = 0, \#_{true} = n)\}$. Assume that assignment of values to random variables $\mathbf{v}$ assigns value $false$ to random variables $wet\_grass(lot_4), wet\_grass(lot_5)$ and $wet\_grass(lot_9)$ and value $true$ to all other ground instances of $wet\_grass(Lot)$. We have $\mathbf{v}(\#_{Lot}[wet\_grass(Lot)]) = (\#_{false} = 3, \#_{true} = n-3)$.

Counting formulas are a form of parameterized random variables. Consider a counting formula $\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]$. If the set $param(f(\ldots,A,\ldots)) \setminus \{A\}$ is not empty, the counting formula $\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]$ represents a set of random variables, one random variable for each ground substitution to all logical variables in $param(f(\ldots,A,\ldots)) \setminus \{A\}$, such that the substitution satisfies the constraints in $\mathcal{C}_A$. The domain of each of these random variables is the range of the counting formula, that is, the domains is the set of histograms having a bucket for each element x in the range of f with entries adding up to $|\mathcal{D}(A) : \mathcal{C}_A|$.

Very often we need to analyze a set of random variables underlying a counting formula $\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]$, that is random variables $ground(f(\ldots,A,\ldots))$. For simplicity, by $ground(\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)])$ we denote the set $ground(f(\ldots,A,\ldots))$, rather than random variables described in the previous paragraph.

Unless otherwise stated, by parameterized random variables we understand both forms: the standard and counting formulas.

First-order probabilistic models describe probabilistic dependencies between parameterized random variables. A *grounding* of a first-order probabilistic model is a propositional probabilistic model obtained by replacing each parameterized random variable with the random variables it represents and replicating appropriate probability distributions. In the next section we describe an example first-order probabilistic modeling language.

2.4.2 Independent Choice Logic

The Independent Choice Logic (ICL) [Poole, 1993, 1997, 2000] is a simple and powerful first-order probabilistic language. It subsumes both belief networks and

logic programs Lloyd [1987]. We have chosen ICL among many other formalisms for its easy syntax and semantics and because it defines directed first-order probabilistic models which are the focus of a big part of this thesis. We start with defining concepts related to logic programs and proceed to define the ICL formally.

A *term* is either a constant, a logical variable, or is of the form $f(t_1, t_2, \ldots, t_k)$ where f is a function symbol and $t_1, t_2, \ldots, t_k$ are terms.

An *atom* is of the form $p(t_1, t_2, \ldots, t_k)$, where p is a predicate symbol and $t_1, t_2, \ldots, t_k$ are terms.

A *clause* is either an atom or is of the form $h \leftarrow a_1 \wedge a_2 \wedge \cdots \wedge a_k$, where h is an atom and a_i is an atom or the negation ($\neg$) of an atom, for $i = 1, 2, \ldots, k$. We call h the *head* of the clause and $a_1 \wedge a_2 \wedge \cdots \wedge a_k$ the *body* of the clause.

A *logic program* is a set of clauses. Informally, *acyclic logic programs* [Apt and Bezem, 1991] are a restricted class of logic programs for which all recursions for variable-free queries eventually halt.

The ICL can be understood as an acyclic logic program with stochastic inputs. The stochastic part is defined by a *choice space*.

An *atomic choice* is an atom that does not unify with the head of any clause. An *alternative* is a set of atomic choices that do not unify with each other. A *choice space* is a set of alternatives such that the atomic choices in different alternatives do not unify.

Definition 2.4. An *ICL theory* consist of:

F: facts, an acyclic logic program;
C: a choice space;
$\mathcal{P}_0$**:** a probability distribution over the alternatives in **C** such that

$$\forall \mathbf{A} \in \mathbf{C} \sum_{\alpha \in \mathbf{A}} \mathcal{P}_0(\alpha) = 1.$$

The meaning of an ICL theory is defined in terms of possible worlds.

A *total choice* for choice space **C** is a selection of exactly one atomic choice from each grounding of each alternative in **C**. Each total choice corresponds to a possible world. The logic program **F** together with the atoms chosen by the total choice define what is true in a possible world.

[00] **C** = {{rain, ¬rain},
[01] {sprinkler, ¬sprinkler},
[02] {wet_r_s, ¬wet_r_s},
[03] {wet_r_ns, ¬wet_r_ns},
[04] {wet_nr_s, ¬wet_nr_s},
[05] {wet_nr_ns, ¬wet_nr_ns}}

[06] **F** = {wet_grass ← rain ∧ sprinkler ∧ wet_r_s,
[07] wet_grass ← rain ∧ ¬sprinkler ∧ wet_r_ns,
[08] wet_grass ← ¬rain ∧ sprinkler ∧ wet_nr_s,
[09] wet_grass ← ¬rain ∧¬sprinkler ∧ wet_nr_ns}

[10] $\mathcal{P}_0(\text{rain}) = 0.2 \quad \mathcal{P}_0(\neg\text{rain}) = 0.8$
[11] $\mathcal{P}_0(\text{sprinkler}) = 0.4 \quad \mathcal{P}_0(\neg\text{sprinkler}) = 0.6$
[12] $\mathcal{P}_0(\text{wet\_r\_s}) = 0.99 \quad \mathcal{P}_0(\neg\text{wet\_r\_s}) = 0.01$
[13] $\mathcal{P}_0(\text{wet\_r\_ns}) = 0.9 \quad \mathcal{P}_0(\neg\text{wet\_r\_ns}) = 0.1$
[14] $\mathcal{P}_0(\text{wet\_nr\_s}) = 0.8 \quad \mathcal{P}_0(\neg\text{wet\_nr\_s}) = 0.2$
[15] $\mathcal{P}_0(\text{wet\_nr\_ns}) = 0.0 \quad \mathcal{P}_0(\neg\text{wet\_nr\_ns}) = 1.0$

Figure 2.4: ICL theory from Example 2.6.

When domains of logical variables are finite, the probability of a possible world is the product of the probabilities of the atomic choices in the corresponding total choice[1]. The probability of a proposition is the sum of probabilities of possible worlds in which the proposition is true.

More details on ICL semantics are provided in [Poole, 2000], see also an overview in [Poole, 2008].

In the example below we show how a belief network from Examples 2.1 and 2.2 can be represented as an ICL theory.

Example 2.6. Consider an ICL theory in Figure 2.4. It represents the same probability distribution as a belief network described in Examples 2.1 and 2.2. The probability distribution has 6 parameters. The ICL theory has a choice space consisting of 6 alternatives with 2 atomic choices each (lines [00] – [05]). In each possible world it rains or not (line [00], a sprinkler is on or not (line [01]), grass is wet or dry when it is raining and the sprinkler is on (line [02]) and so on. The probability distribution over the alternatives is specified in lines [07] – [18]. For ex-

[1] In the general case, which is beyond the scope of this thesis, we need measure over sets of possible worlds.

[00] **C** = {{rain, ¬rain},
[01] {sprinkler(Lot), ¬sprinkler(Lot)},
[02] {wet_r_s(Lot), ¬wet_r_s(Lot)},
[03] {wet_r_ns(Lot), ¬wet_r_ns(Lot)},
[04] {wet_nr_s(Lot), ¬wet_nr_s(Lot)},
[05] {wet_nr_ns(Lot), ¬wet_nr_ns(Lot)}}

[06] **F** = {wet_grass(Lot) ← rain ∧ sprinkler(Lot) ∧ wet_r_s(Lot),
[07] wet_grass(Lot) ← rain ∧ ¬sprinkler(Lot) ∧ wet_r_ns(Lot),
[08] wet_grass(Lot) ← ¬rain ∧ sprinkler(Lot) ∧ wet_nr_s(Lot),
[09] wet_grass(Lot) ← ¬rain ∧¬sprinkler(Lot) ∧ wet_nr_ns(Lot)}

[10] $\mathcal{P}_0(\text{rain}) = 0.2 \quad \mathcal{P}_0(\neg\text{rain}) = 0.8$
[11] $\mathcal{P}_0(\text{sprinkler(Lot)}) = 0.4 \quad \mathcal{P}_0(\neg\text{sprinkler(Lot)}) = 0.6$
[12] $\mathcal{P}_0(\text{wet\_r\_s(Lot)}) = 0.99 \quad \mathcal{P}_0(\neg\text{wet\_r\_s(Lot)}) = 0.01$
[13] $\mathcal{P}_0(\text{wet\_r\_ns(Lot)}) = 0.9 \quad \mathcal{P}_0(\neg\text{wet\_r\_ns(Lot)}) = 0.1$
[14] $\mathcal{P}_0(\text{wet\_nr\_s(Lot)}) = 0.8 \quad \mathcal{P}_0(\neg\text{wet\_nr\_s(Lot)}) = 0.2$
[15] $\mathcal{P}_0(\text{wet\_nr\_ns(Lot)}) = 0.0 \quad \mathcal{P}_0(\neg\text{wet\_nr\_ns(Lot)}) = 1.0$

Figure 2.5: ICL theory for multiple lots from Example 2.6.

[00] **C** = {{rain(false), rain(true)},
[01] {sprinkler(Lot,false), sprinkler(Lot,true)},
[02] {wet_rain_sprinkler(Lot,false,false,false), wet_rain_sprinkler(Lot,true,false,false)},
[03] {wet_rain_sprinkler(Lot,false,false,true), wet_rain_sprinkler(Lot,true,false,true)},
[04] {wet_rain_sprinkler(Lot,false,true,false), wet_rain_sprinkler(Lot,true,true,false)},
[05] {wet_rain_sprinkler(Lot,false,true,true), wet_rain_sprinkler(Lot,true,true,true)}}

[06] **F** = {wet_grass(Lot,X) ← rain(Y) ∧ sprinkler(Lot,Z) ∧ wet_rain_sprinkler(Lot,X,Y,Z)}

[07] $\mathcal{P}_0(\text{rain(false)}) = 0.8$
[08] $\mathcal{P}_0(\text{rain(true)}) = 0.2$
[09] $\mathcal{P}_0(\text{sprinkler(Lot,false)}) = 0.6$
[10] $\mathcal{P}_0(\text{sprinkler(Lot,true)}) = 0.4$
[11] $\mathcal{P}_0(\text{wet\_rain\_sprinkler(Lot,false,false,false)}) = 1.0$
[12] $\mathcal{P}_0(\text{wet\_rain\_sprinkler(Lot,true,false,false)}) = 0.0$
[13] $\mathcal{P}_0(\text{wet\_rain\_sprinkler(Lot,false,false,true)}) = 0.2$
[14] $\mathcal{P}_0(\text{wet\_rain\_sprinkler(Lot,true,false,true)}) = 0.8$
[15] $\mathcal{P}_0(\text{wet\_rain\_sprinkler(Lot,false,true,false)}) = 0.1$
[16] $\mathcal{P}_0(\text{wet\_rain\_sprinkler(Lot,true,true,false)}) = 0.9$
[17] $\mathcal{P}_0(\text{wet\_rain\_sprinkler(Lot,false,true,true)}) = 0.01$
[18] $\mathcal{P}_0(\text{wet\_rain\_sprinkler(Lot,true,true,true)}) = 0.99$

Figure 2.6: More elegant ICL theory for multiple lots from Example 2.6.

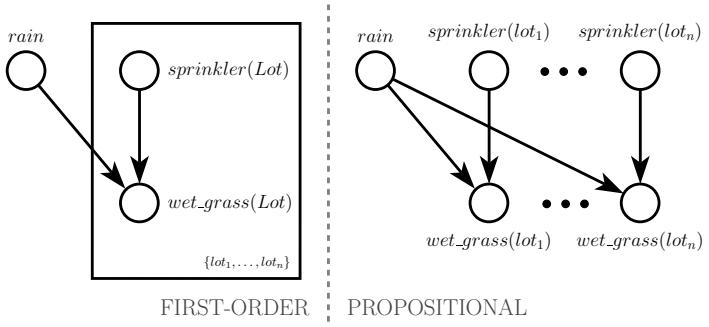


Figure 2.7: Graphical representation of the ICL theory for multiple lots from Example 2.6.

ample, $\mathcal{P}_0(\text{wet\_r\_s}) = 0.99 = \mathcal{P}(wet\_grass = true | rain = true, sprinkler = true)$. Finally, clauses in lines [06]–[09] specify the conditional probability for *wet_grass* given *rain* and *sprinkler*. Atoms *rain*, *wet_grass*, and *sprinkler* from the ICL theory correspond to nodes of the belief network.

The above theory can be easily generalized to multiple lots as we show in Figure 2.5. Figure 2.7 illustrates the resulting directed first-order probabilistic model using *plates*. The notion of *plates* [Buntine, 1994] is similar to the idea of parameterized random variables; we use plates notation in our figures throughout this thesis. Consider the left part of Figure 2.7. A subgraph involving two parameterized random variables $sprinkler(Lot)$ and $wet\_grass(Lot)$ is enclosed within a box. The box is referred to as a plate. It implies that the subgraph is duplicated as many times as there are individuals in the population associated with the plate. The individuals are enumerated in the bottom-right corner of the plate. Arcs coming into the plate and leaving the plate are duplicated as well. Atoms from the ICL theory presented in Figure 2.5 are parameterized random variables. Atoms *rain*, $wet\_grass(Lot)$, and $sprinkler(Lot)$ from the ICL theory correspond to nodes of the first-order probabilistic model shown in Figure 2.7.

Facts can be specified in a more elegant way if we parameterize atoms over truth values of the corresponding parameterized random variables from the first-order model. Figure 2.6 shows a version of the ICL theory from Figure 2.5 that does this. Variable X parameterizes truth values of a parameterized random variable $wet\_grass(Lot)$, variable Y parameterizes truth values of a parameterized ran-

dom variable $rain()$, and variable Z parameterizes truth values of a parameterized random variable $sprinkler(Lot)$. For example, an atom $sprinkler(Lot, true)$ is true when the parameterized random variable $sprinkler(Lot)$ is true and an atom $sprinkler(Lot, false)$ is true when when the parameterized random variable $sprinkler(Lot)$ is false.

2.5 Lifted probabilistic inference

Although many first-order probabilistic languages have emerged, for many years the most common exact inference technique has been based on dynamical propositionalization of the portion of the first order model that is relevant to the query, followed by probabilistic inference performed at the propositional level [Breese, 1992]. This approach is known as *knowledge-based model construction* (KBMC). Unfortunately, even a very simple first-order model can result in a very large propositional model and the inference at propositional level can be formidably expensive.

The big computational cost of the KBMC approach motivated work on exploiting redundant computation [Koller and Pfeffer, 1997; Pfeffer and Koller, 2000] during inference in first-order probabilistic models and work on compiling first-order models to secondary structures (for example arithmetic circuits [Chavira et al., 2006]) to facilitate efficient inference.

The idea of *lifted probabilistic inference* is to carry out as much inference as possible without propositionalizing a first-order probabilistic model or its part. An exact lifted probabilistic inference procedure (in some literature called *inversion elimination*) for first-order probabilistic models was presented in Poole [2003]. Under certain conditions, the inversion elimination procedure can sum out multiple random variables from a model without considering each variable separately. de Salvo Braz et al. [2005, 2006, 2007] worked on increasing the scope of lifted probabilistic inference and introduced a *counting elimination* procedure. Further work by Milch et al. [2008] added counting formulas (see Section 2.4.1.1) and further expanded the scope of lifted probabilistic inference. Their work resulted in the C-FOVE algorithm, which is currently the state of the art in exact lifted probabilistic inference.

While Poole considered directed models, the later work by de Salvo Braz et al. and Milch et al. focused on undirected models, although their results can be used for directed models. Directed models have the advantage of allowing pruning of the part of a model that is irrelevant to the query. Also, conditional probability distributions in directed models can be interpreted and learned locally, which is important for models that are specified by people or need to be understood by people.

In Section 2.5.1 we describe data structures used during lifted inference and in Section 2.5.2 we provide an overview of the C-FOVE algorithm.

2.5.1 Parametric factors

To perform inference in first-order probabilistic models we need a data structure that would fulfill a role analogical to the role of factors (Section 2.3.1) during inference in belief networks. The above is a motivation behind parfactors [Poole, 2003]. They are used to represent conditional probability distributions in directed first-order models and potentials in undirected first-order models as well as intermediate computation results during lifted inference in first-order models.

Definition 2.5. A *parametric factor* or *parfactor* is a triple $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ where:

- $\mathcal{C}$ is a set of inequality constraints on logical variables (between a logical variable and a constant or between two logical variables);
- $\mathcal{V}$ is a set of parameterized random variables, such that for any two parameterized random variables $f(\dots)$, $f'(\dots)$ from $\mathcal{V}$ we have

$$ground(f(\dots)) : \mathcal{C} \cap ground(f'(\mathcal{C})) : \mathcal{C} = \emptyset; \tag{2.2}$$

- $\mathcal{F}$ is a factor from the Cartesian product of ranges of parameterized random variables in $\mathcal{V}$ to the reals.

A parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ represents a set of factors, one for each ground substitution $\mathcal{G}$ to all free logical variables in $\mathcal{V}$ that satisfies the constraints in $\mathcal{C}$. Each such factor $\mathcal{F}_\mathcal{G}$ is a factor on the set of random variables obtained by applying a substitution $\mathcal{G}$. Given an assignment $\mathbf{v}$ to random variables represented by $\mathcal{V}$, $\mathbf{v}(\mathcal{F}_\mathcal{G}) = \mathbf{v}(\mathcal{F})$. $|\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle|$ denotes the number of factors represented by a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$.

Condition (2.2) ensures that each ground substitution $\mathcal{G}$ to all free logical variables in $\mathcal{V}$ that satisfies the constraints in $\mathcal{C}$ results in a set of random variables of the same size.

In this thesis we additionally assume that two logical variables that participate in the same inequality constraint from $\mathcal{C}$ are typed with the same population. Extending lifted inference to handle hierarchies of types is an interesting idea for future work, but it is orthogonal to the focus of this thesis.

Example 2.7. The ICL theory for multiple lots from Example 2.6 (shown in Figure 2.7) can be represented by the following parfactors:

$$\langle \emptyset, \{rain()\}, \begin{array}{c|c} \mathcal{P}(rain() = false) & \mathcal{P}(rain() = true) \\ \hline 0.8 & 0.2 \end{array} \rangle, \quad [01]$$

$$\langle \emptyset, \{sprinkler(Lot)\}, \begin{array}{c|c} \mathcal{P}(sprinkler(Lot) = false) & \mathcal{P}(sprinkler(Lot) = true) \\ \hline 0.6 & 0.4 \end{array} \rangle, \quad [02]$$

$$\langle \emptyset, \{rain(), sprinkler(Lot), wet\_grass(Lot)\}, \quad [03]$$

$rain()$	$sprinkler(Lot)$	$\mathcal{P}(wet\_grass(Lot) = false)$	$\mathcal{P}(wet\_grass(Lot) = true)$
$false$	$false$	1.0	0.0
$false$	$true$	0.2	0.8
$true$	$false$	0.1	0.9
$true$	$true$	0.01	0.99

$\rangle$.

Parfactor [01] represents a set containing one factor. Parfactors [02] and [03] represent sets of factors of size n.

Constraints within parfactors allow us to store knowledge about particular individuals. It might be prior knowledge as well as coming from observations.

Example 2.8. Assume that we know that sprinkler on lot lot_i is very likely to be on. Then the second parfactor from Example 2.7 ([02]) could be replaced by the following two parfactors:

$$\langle \emptyset, \{sprinkler(lot_i)\}, \begin{array}{c|c} \mathcal{P}(sprinkler(lot_i) = false) & \mathcal{P}(sprinkler(lot_i) = true) \\ \hline 0.1 & 0.9 \end{array} \rangle,$$

$$\langle \{Lot \neq lot_i\}, \{sprinkler(Lot)\}, \begin{array}{c|c} \mathcal{P}(sprinkler(Lot) = false) & \mathcal{P}(sprinkler(Lot) = true) \\ \hline 0.6 & 0.4 \end{array} \rangle.$$

Next example illustrates the trade-offs between the sizes of parfactors on standard parameterized random variables, parfactors on counting formulas and parfactors on not parameterized random variables and their expressive power.

Example 2.9. Let $wet\_grass(Lot)$ be a parameterized random variable from Example 2.3 and $\#_{Lot}[wet\_grass(Lot)]$ be a counting formula from Example 2.5. Recall that Lot is a logical variable typed with a population $\{lot_1, lot_2, \ldots, lot_n\}$ and wet_grass is a functor with range $\{false, true\}$. Consider the following three parfactors:

$$\langle \emptyset, \{wet\_grass(Lot)\}, \mathcal{F}_1 \rangle; \quad [1]$$

$$\langle \emptyset, \{\#_{Lot}[wet\_grass(Lot)]\}, \mathcal{F}_2 \rangle; \quad [2]$$

$$\langle \emptyset, \{wet\_grass(lot_1), wet\_grass(lot_2), \ldots, wet\_grass(lot_n)\}, \mathcal{F}_3 \rangle . \quad [3]$$

Factor $\mathcal{F}_1$ represents a function from $\{false, true\}$ to the reals and has size 2, thus its size is independent of $|\mathcal{D}(Lot)| = n$. Parfactor [1] can represent a set of n identical real-valued discrete functions on each random variable from $ground(wet\_grass(Lot))$.

The next factor, $\mathcal{F}_2$, represents a function from the set $\{(\#_{false} = n, \#_{true} = 0), \ldots, (\#_{false} = 0, \#_{true} = n)\}$ to the set of real numbers and has size $n+1$. Parfactor [2] can represent these real-valued discrete functions on random variables from the set $ground(wet\_grass(Lot))$ that only depend the number of random variables from $ground(wet\_grass(Lot))$ that take a particular value not on which of them take it.

Finally, factor $\mathcal{F}_3$ represents a function from $\times_{i=1}^{n} \{false, true\}$ to the reals and has size 2^n. Parfactor [3] can represent arbitrary real-valued discrete functions on random variables $ground(wet\_grass(Lot))$.

2.5.1.1 Normal-form constraints

Let X be a logical variable in $\mathcal{V}$ from a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$. In general, the size of the set $\mathcal{D}(X) : \mathcal{C}$ depends on other logical variables in $\mathcal{V}$.

Example 2.10. Consider a parfactor $\langle \{X \neq x_1, X \neq Y\}, \{g(X), h(X,Y)\}, \mathcal{F} \rangle$, where $\mathcal{D}(X) = \mathcal{D}(Y) = \{x_1, \ldots, x_n\}$. The size of the set $\mathcal{D}(X) : \{X \neq x_1, X \neq Y\}$ depends on the logical variable Y. It is equal $n-1$ when $Y = x_1$ and $n-2$ when $Y \neq x_1$.

The above property is undesirable for parfactors involving counting formulas because the set of possible values of a counting formula $\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]$ can

take depends on the size of the set $\mathcal{D}(A) : \mathcal{C}$. Milch et al. [2008] introduced a special class of sets of inequality constraints.

Let $\mathcal{C}$ be a set of inequality constraints on logical variables and X be a logical variable. We denote by $\mathcal{E}_X^{\mathcal{C}}$ the *excluded set* for X, that is, the set of terms t such that $(X \neq t) \in \mathcal{C}$. Set $\mathcal{C}$ is in *normal form* if for each inequality $(X \neq Y) \in \mathcal{C}$, where X and Y are logical variables, $\mathcal{E}_X^{\mathcal{C}} \setminus \{Y\} = \mathcal{E}_Y^{\mathcal{C}} \setminus \{X\}$. A parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F}_{\mathcal{F}} \rangle$ is in normal form if set $\mathcal{C}$ is in normal form.

Consider a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F}_{\mathcal{F}} \rangle$, where $\mathcal{C}$ is in normal form. For all logical variables X in $\mathcal{V}$, $|\mathcal{D}(X) : \mathcal{C}| = |\mathcal{D}(X)| - |\mathcal{E}_X^{\mathcal{C}}|$.

Example 2.11. Consider the parfactor from Example 2.10. Let $\mathcal{C}$ denote a set of constraints from this parfactor. The set $\mathcal{C}$ contains only one inequality between logical variables, namely $X \neq Y$. We have $\mathcal{E}_X^{\mathcal{C}} = \{x_1, Y\}$ and $\mathcal{E}_Y^{\mathcal{C}} = \{X\}$. As $\mathcal{E}_X^{\mathcal{C}} \setminus \{Y\} \neq \mathcal{E}_Y^{\mathcal{C}} \setminus \{X\}$, the parfactor is not in normal form.

Consider a parfactor $\langle \{X \neq Y,\, X \neq x_1,\, Y \neq x_1\}, \{g(X),\, h(X,Y)\}, \mathcal{F} \rangle$, where $\mathcal{D}(X) = \mathcal{D}(Y) = \{x_1, \ldots, x_n\}$. Let $\mathcal{C}'$ denote a set of constraints from this parfactor. As $\mathcal{E}_X^{\mathcal{C}'} \setminus \{Y\} = \mathcal{E}_Y^{\mathcal{C}'} \setminus \{X\}$, the parfactor is in normal form and $|\mathcal{D}(X) : \mathcal{C}'| = n - 2$ and $|\mathcal{D}(Y) : \mathcal{C}'| = n - 2$.

Following Milch et al. [2008] we require that for a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F}_{\mathcal{F}} \rangle$ involving counting formulas, the union of $\mathcal{C}$ and the constraints in all the counting formulas in $\mathcal{V}$ is in normal form. Other parfactors do not need to be in normal form. The trade-off between requiring normal-form and allowing unrestricted sets of inequality constraints is discussed in Section 5.4.

2.5.2 C-FOVE

In this section we provide an overview of the C-FOVE algorithm [Milch et al., 2008]. The algorithm builds on previous work by Poole [2003] and de Salvo Braz et al. [2007]. C-FOVE is not tied to any first-order probabilistic language. It can be used to perform inference in any model for which joint probability distribution can be represented as a product of factors.

Let Φ be a set of parfactors. Let $\mathcal{J}(\Phi)$ denote a factor equal to the product of all factors represented by elements of Φ. Let $\mathbf{U}$ be the set of all random variables represented by parameterized random variables present in parfactors in Φ. Let $\mathbf{Q}$

be a subset of $\mathbf{U}$. The *marginal of* $\mathcal{J}(\Phi)$ *on* $\mathbf{Q}$, denoted $\mathcal{J}_{\mathbf{Q}}(\Phi)$, is defined as $\mathcal{J}_{\mathbf{Q}}(\Phi) = \sum_{\mathbf{U}\setminus\mathbf{Q}} \mathcal{J}(\Phi)$.

Given Φ and $\mathbf{Q}$, the C-FOVE algorithm computes the marginal $\mathcal{J}_{\mathbf{Q}}(\Phi)$ by summing out random variables from $\mathbf{U} \setminus \mathbf{Q}$, where possible in a lifted manner. Evidence can be handled by adding to Φ additional parfactors on observed random variables. The C-FOVE algorithm assumes that all parfactors are in normal form.

As lifted summing out is only possible under certain conditions, the C-FOVE algorithm uses elimination enabling operations, such as applying substitutions to parfactors and multiplication. We start our description of C-FOVE with lifted elimination, then describe elimination enabling operations and finally show how they can be combined to perform lifted probabilistic inference.

Milch et al. [2008] define parfactors as quadruples $\langle \mathcal{C}, \mathcal{L}, \mathcal{V}, \mathcal{F} \rangle$. $\mathcal{C}$, $\mathcal{V}$, $\mathcal{F}$ have the same meaning as in Section 2.5.1 and $\mathcal{L}$ is a set of logical variables. Thus, the only difference with our notation is that they explicitly list logical variables present in a parfactor. This lets $\mathcal{L}$ be a superset of the set of logical variables that parameterize elements of $\mathcal{V}$.

2.5.2.1 Lifted elimination

The aim of lifted elimination is to sum out a parameterized random variable from a parfactor with much less computation then it would be required if we first converted the parfactor to a set of factors and summed out the ground instances of the parameterized random variable from these factors. We first describe a lifted summation of a parameterized random variable that is not a counting formula and next present a lifted elimination of a counting formula.

Let $\mathcal{F}$ be a factor and r be a real number. By $\mathcal{F}^r$ we denote a factor such that given an assignment $\mathbf{v}$ of values to random variables, $\mathbf{v}(\mathcal{F}^r) = \mathbf{v}(\mathcal{F})^r$. In $\mathcal{F}^r$, the values of factor $\mathcal{F}$ are brought to power r.

Let $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ be a parfactor and r be a real number. By $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle^r$ we denote a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F}^r \rangle$.

The following two propositions follow directly from Propositions 1, 2 and 4 from Milch et al. [2008]. The first proposition considers summing out a set of

random variables represented by a parameterized random variable (not a counting formula) from a product of factors represented by a set of parfactors.

Proposition 2.1. Let $\Phi = \{g_1, g_2, \ldots, g_m\}$ be a set of parfactors. Let $g_i = \langle \mathcal{C}_i, \mathcal{V}_i, \mathcal{F}_i \rangle$ be a normal-form parfactor from Φ and $f(\ldots)$ be a parameterized random variable (not a counting formula) from $\mathcal{V}_i$. Suppose that:

(S1) For all parfactors $g_j = \langle \mathcal{C}_j, \mathcal{V}_j, \mathcal{F}_j \rangle \in \Phi$, $i \neq j$ and all $h(\ldots) \in \mathcal{V}_j$ we have $ground(f(\ldots)) : \mathcal{C}_i \cap ground(h(\ldots)) : \mathcal{C}_j = \emptyset$. That is, no other parfactor in Φ includes parameterized random variables that represent random variables represented by $f(\ldots)$.

(S2) $param(f(\ldots)) \supseteq \bigcup_{f'(\ldots) \in (\mathcal{V} \setminus \{f(\ldots)\})} param(f'(\ldots))$. That is, the set of logical variables in $f(\ldots)$ is a superset of the union of logical variables in other parameterized random variables from $\mathcal{V}$.

Let $g'_i = \left\langle \mathcal{C}_i, \mathcal{V}_i \setminus \{f(\ldots)\}, \sum_{f(\ldots)} \mathcal{F} \right\rangle$ and $r = |g_i|/|g'_i|$, where . Then

$$\sum_{ground(f(\ldots)):\mathcal{C}_i} \mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_i\} \cup \{(g'_i)^r\}) . \tag{2.3}$$

Condition (S1) makes sure that we are completely eliminating random variables $ground(f(\ldots)) : \mathcal{C}_i$ from the set of parfactors Φ. Condition (S2) guarantees that a result of lifted summation is equivalent to performing a series of summations at a propositional level. If after summing out $f(\ldots)$ some logical variables disappear from the parfactor g_i, then g'_i represents less factors than g_i. To compensate for this, the values of the factor in g'_i are brought to the power $r = |g_i|/|g'_i|$.

Example 2.12. Consider the following set of parfactors:

$$\Phi = \{\big\langle \{A \neq B\}, \{f(A,B), h(B)\},$$

$f(A,B)$	$h(B)$	value
$false$	$false$	α_1
$false$	$true$	α_2
$true$	$false$	α_3
$true$	$true$	α_4

$\big\rangle,$ [1]

$$\big\langle \emptyset, \{e(C), h(x_1)\},$$

$e(C)$	$h(x_1)$	value
$green$	$false$	β_1
$green$	$true$	β_2
$orange$	$false$	β_3
$orange$	$true$	β_4
red	$false$	β_5
red	$true$	β_6

$\big\rangle\},$ [2]

where $range(f) = range(h) = \{false, true\}$, $range(e) = \{green,\ orange,\ red\}$, $\mathcal{D}(A) = \mathcal{D}(B) = \{x_1, x_2, \ldots, x_n\}$, $\mathcal{D}(C) = \{y_1, y_2, \ldots, y_m\}$, and $\alpha_1, \ldots, \alpha_4, \beta_1, \ldots, \beta_6$ are real numbers. Assume we want to compute $\sum_{ground(f(A,B)):\{A \neq B\}} \mathcal{J}(\Phi)$. Conditions (S1) and (S2) of Proposition 2.1 are satisfied. Let us define a new parfactor $[1']$:

$$\Big\langle \emptyset, \{h(B)\}, \sum_{f(A,B)}$$

$f(A,B)$	$h(B)$	value
$false$	$false$	α_1
$false$	$true$	α_2
$true$	$false$	α_3
$true$	$true$	α_4

$\Big\rangle = \big\langle \emptyset, \{h(B)\},$

$h(B)$	value
$false$	$\alpha_1 + \alpha_3$
$true$	$\alpha_2 + \alpha_4$

$\big\rangle.$ [1']

The logical variable A is present in parfactor [1] and it is absent from parfactor [1']. Parfactor [1] represents $n(n-1)$ factors. Parfactor [1'] represents n factors, that is, it represents $n-1$ times fewer factors than [1]. We have

$$[1']^{(n-1)} = \big\langle \{A \neq B\}, \{f(A,B), h(B)\},$$

$h(B)$	value
$false$	$(\alpha_1 + \alpha_3)^{(n-1)}$
$true$	$(\alpha_2 + \alpha_4)^{(n-1)}$

$\big\rangle.$

From Proposition 2.1:

$$\sum_{ground(f(A,B)):\{A \neq B\}} \mathcal{J}(\Phi) = \mathcal{J}(\{[1']^{(n-1)}, [2]\}).$$

An analogous proposition to Proposition 2.1 holds for counting formulas. It considers summing out a set of random variables represented by a counting formula from a product of factors represented by a set of parfactors.

Proposition 2.2. Let $\Phi = \{g_1, g_2, \ldots, g_m\}$ be a set of parfactors. Let $g_i = \langle \mathcal{C}_i, \mathcal{V}_i, \mathcal{F}_i \rangle$ be a normal-form parfactor from Φ and $\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]$ be a counting formula from $\mathcal{V}_i$. Suppose that:

(SC1) For all $g_j = \langle \mathcal{C}_j, \mathcal{V}_j, \mathcal{F}_j \rangle \in \Phi$, $i \neq j$ and all $h(\ldots) \in \mathcal{V}_j$ we have $ground(\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]) : \mathcal{C}_i \cap ground(h(\ldots)) : \mathcal{C}_j = \emptyset$;

(SC2) $param(\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]) \supseteq \bigcup_{f'(\ldots)\in(\mathcal{V}\setminus\{\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]\})} param(f'(\ldots))$. That is, the set of logical variables of $\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]$ is a superset of the union of logical variables of other parameterized random variables from set $\mathcal{V}$.

Let $g_i' = \langle \mathcal{C}_i, \mathcal{V}_i', \mathcal{F}_i' \rangle$, where $\mathcal{V}_i' = \mathcal{V}_i \setminus \{\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]\}$ and $\mathcal{F}_i'$ is a factor from the Cartesian product of ranges of parameterized random variables in $\mathcal{V}_i'$ to the reals. Let $\mathbf{v}$ be a value assignment to all random variables but $ground(f(\ldots,A,\ldots)) : \mathcal{C}_i \cup \mathcal{C}_A$. Factor $\mathcal{F}_i'$ is defined as follows:

$$\mathbf{v}(\mathcal{F}_i') = \sum_{h()\in range(\#_{A:\mathcal{C}_A}[f()])} \frac{|\mathcal{D}(A) : \mathcal{C}_A|!}{\prod_{x\in range(f(\ldots,A,\ldots))} h(x)!} \mathbf{v}(\mathcal{F}_i)(h()), \tag{2.4}$$

Let $r = |g_i|/|g_i'|$. Then

$$\sum_{ground(f(\ldots)):\mathcal{C}_i} \mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_i\} \cup \{(g_i')^r\}). \tag{2.5}$$

The only difference between Proposition 2.2 and Proposition 2.1 is the expression

$$\frac{|\mathcal{D}(A) : \mathcal{C}_A|!}{\prod_{x\in range(f(\ldots,A,\ldots))} h(x)!}.$$

The expression is equal to the number of assignments $\mathbf{v}'$ of values to random variables in a set $ground(\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)])$ such that $\mathbf{v}'(\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]) = h()$.

Example 2.13. Consider the following set of parfactors:

$$\Phi = \{\big\langle \emptyset, \{\#_{A:\{A\neq B\}}[f(A)], h(B)\},$$

$\#_{A:\{A\neq B\}}[f(A)]$	$h(B)$	value
$(\#_{false} = n-1, \#_{true} = 0)$	$false$	α_1
$(\#_{false} = n-1, \#_{true} = 0)$	$true$	α_2
$(\#_{false} = n-2, \#_{true} = 1)$	$false$	α_3
$(\#_{false} = n-2, \#_{true} = 1)$	$true$	α_4
$\vdots$	$\vdots$	$\vdots$
$(\#_{false} = 0, \#_{true} = n-1)$	$false$	α_{2n-1}
$(\#_{false} = o, \#_{true} = n-1)$	$true$	α_{2n}

$\big\rangle$, [1]

$$\big\langle \emptyset, \{e(C), h(x_1)\},$$

$e(C)$	$h(x_1)$	value
$green$	$false$	β_1
$green$	$true$	β_2
$orange$	$false$	β_3
$orange$	$true$	β_4
red	$false$	β_5
red	$true$	β_6

$\big\rangle\}$, [2]

where $range(f) = range(h) = \{false, true\}$, $range(e) = \{green,\ orange,\ red\}$, $\mathcal{D}(A) = \mathcal{D}(B) = \{x_1, x_2, \ldots, x_n\}$, $\mathcal{D}(C) = \{y_1, y_2, \ldots, y_m\}$, and $\alpha_1, \ldots, \alpha_{2n}, \beta_1, \ldots, \beta_6$ are real numbers. Assume we want to compute $\sum_{ground(\#_{A:\{A\neq B\}}[f(A)])} \mathcal{J}(\Phi)$. Conditions (SC1) and (SC2) of Proposition 2.2 are satisfied. Let us define a new parfactor $[1']$:

$$\Big\langle \emptyset, \{h(B)\}, \textstyle\sum_{\#_{A:\{A\neq B\}}[f(A)]}$$

$\#_{A:\{A\neq B\}}[f(A)]$	$h(B)$	value
$(\#_{false} = n-1, \#_{true} = 0)$	$false$	α_1
$(\#_{false} = n-1, \#_{true} = 0)$	$true$	α_2
$(\#_{false} = n-2, \#_{true} = 1)$	$false$	α_3
$(\#_{false} = n-2, \#_{true} = 1)$	$true$	α_4
$\vdots$	$\vdots$	$\vdots$
$(\#_{false} = 0, \#_{true} = n-1)$	$false$	α_{2n-1}
$(\#_{false} = 0, \#_{true} = n-1)$	$true$	α_{2n}

$\Big\rangle$, $=$

$$\big\langle \emptyset, \{h(B)\},$$

$h(B)$	value
$false$	$\sum_{i=1}^{n} \binom{n-1}{n-i} \alpha_{2i-1}$
$true$	$\sum_{i=1}^{n} \binom{n-1}{n-i} \alpha_{2i}$

$\big\rangle$. [1']

In the above calculation the expression $\frac{|\mathcal{D}(A):\mathcal{C}_A|!}{\prod_{x\in range(f(\ldots,A,\ldots))} h(x)!}$ from Proposition 2.2 for histogram $(\#_{false} = n-i, \#_{true} = i-1)$ is equal to $\frac{(n-1)!}{(n-i)!(i-1)!}$, which is equal to $\binom{n-1}{n-i}$.

Parfactor $[1]$ represents n factors and so does parfactor $[1']$. Therefore we do not need to bring elements of $[1']$ to any power. From Proposition 2.2:

$$\sum_{ground(\#_{A:\{A\neq B\}}[f(A)])} \mathcal{J}(\Phi) = \mathcal{J}(\{[1'],[2]\}).$$

2.5.2.2 Parfactor multiplication

Parfactor multiplication allows us to combine parfactors which involve parameterized random variables that represent the same set of random variables. This allows us to satisfy condition (S1) of Proposition 2.1 and condition (SC1) of Proposition 2.2. Parfactor multiplication can be performed in a lifted manner. This means that, although parfactors participating in a multiplication as well as their product represent multiple factors, only one factor multiplication is performed. The following proposition is a consequence of Propositions 5 and 4 from Milch et al. [2008] and earlier work by de Salvo Braz et al. [2007].

Proposition 2.3. Let $\Phi = \{g_1, g_2, \ldots, g_m\}$ be a set of parfactors. Let $g_i = \langle \mathcal{C}_i, \mathcal{V}_i, \mathcal{F}_i \rangle$ and $g_j = \langle \mathcal{C}_j, \mathcal{V}_j, \mathcal{F}_j \rangle$ be parfactors from Φ. Suppose that:

(M1) $\forall f(\ldots) \in \mathcal{V}_i \forall f'(\ldots) \in \mathcal{V}_j \ (ground(f(\ldots)) : \mathcal{C}_i = ground(f'(\ldots)) : \mathcal{C}_j) \vee (ground(f(\ldots)) : \mathcal{C}_i \cap ground(f'(\ldots)) : \mathcal{C}_j = \emptyset)$. That is, sets of random variables represented by parameterized random variables from each parfactor are identical or disjoint.

(M2) All parameterized random variables $f(\ldots) \in \mathcal{V}_i$ and $f'(\ldots) \in \mathcal{V}_j$ such that $ground(f(\ldots)) : \mathcal{C}_i = ground(f'(\ldots)) : \mathcal{C}_j$, $f(\ldots)$ and $f'(\ldots)$ are identically parameterized by logical variables and the set of other logical variables present in parfactor g_i is disjoint with the set of logical variables present in parfactor g_j.

Let $g = \langle \mathcal{C}_i \cup \mathcal{C}_j, \mathcal{V}_i \cup \mathcal{V}_j, \mathcal{F}_i \odot \mathcal{F}_j \rangle$, $r_i = |g_i|/|g|$ and $r_j = |g_j|/|g|$. Then

$$\mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_i, g_j\} \cup \{\langle \mathcal{C}_i \cup \mathcal{C}_j, \mathcal{V}_i \cup \mathcal{V}_j, \mathcal{F}_i^{r_i} \odot \mathcal{F}_j^{r_j} \rangle\}). \tag{2.6}$$

Condition (M1) makes sure that the product of parfactors represents a set of factors of the same dimensionality. Condition (M2) guarantees correctness of set

unions in the definition of parfactor g. Similarly to lifted elimination, Equation 2.6 accounts for logical variables present in the product. The product parfactor g may represent more factors than a parfactor participating in the product, for example g_i. To compensate for this, the values of the factor in g_i are brought to the power $r_i = |g_i|/|g|$ before computing the final product.

Example 2.14. Consider the following set of parfactors:

$$\Phi = \{\langle \{A \neq B\}, \{f(A,B), h(B)\},$$

$f(A,B)$	$h(B)$	value
$false$	$false$	α_1
$false$	$true$	α_2
$true$	$false$	α_3
$true$	$true$	α_4

$\rangle$, [1]

$$\langle \emptyset, \{e(C), h(B)\},$$

$e(C)$	$h(B)$	value
$green$	$false$	β_1
$green$	$true$	β_2
$orange$	$false$	β_3
$orange$	$true$	β_4
red	$false$	β_5
red	$true$	β_6

$\rangle\}$, [2]

where $range(f) = range(h) = \{false, true\}$, $range(e) = \{green, orange, red\}$, $\mathcal{D}(A) = \mathcal{D}(B) = \{x_1, x_2, \ldots, x_n\}$, $\mathcal{D}(C) = \{y_1, y_2, \ldots, y_m\}$, and $\alpha_1, \ldots, \alpha_4, \beta_1, \ldots, \beta_6$ are real numbers. Assume we want to multiply parfactors [1] and [2]. Conditions (M1) and (M2) of Proposition 2.3 are satisfied. Parfactor [1] represents $n(n-1)$ factors, parfactor [2] represents nm factors and their product will represent $n(n-1)m$ factors. We need to take it into account while computing a factor component of the product. Let us define a new parfactor [3]:

$$\Big\langle \{A \neq B\}, \{f(A,B), e(C), h(B)\},$$

$f(A,B)$	$h(B)$	value
$false$	$false$	$\alpha_1^{\frac{n(n-1)}{n(n-1)m}}$
$false$	$true$	$\alpha_2^{\frac{n(n-1)}{n(n-1)m}}$
$true$	$false$	$\alpha_3^{\frac{n(n-1)}{n(n-1)m}}$
$true$	$true$	$\alpha_4^{\frac{n(n-1)}{n(n-1)m}}$

$\odot$

$e(C)$	$h(B)$	value
$green$	$false$	$\beta_1^{\frac{nm}{n(n-1)m}}$
$green$	$true$	$\beta_2^{\frac{nm}{n(n-1)m}}$
$orange$	$false$	$\beta_3^{\frac{nm}{n(n-1)m}}$
$orange$	$true$	$\beta_4^{\frac{nm}{n(n-1)m}}$
red	$false$	$\beta_5^{\frac{nm}{n(n-1)m}}$
red	$true$	$\beta_6^{\frac{nm}{n(n-1)m}}$

$\Big\rangle =$

$\Big\langle \{A \neq B\}, \{f(A,B), e(C), h(B)\},$

$f(A,B)$	$e(C)$	$h(B)$	value
$false$	$green$	$false$	$\alpha_1^{\frac{1}{m}} \beta_1^{\frac{1}{n-1}}$
$false$	$green$	$true$	$\alpha_2^{\frac{1}{m}} \beta_2^{\frac{1}{n-1}}$
$false$	$orange$	$false$	$\alpha_1^{\frac{1}{m}} \beta_3^{\frac{1}{n-1}}$
$false$	$orange$	$true$	$\alpha_2^{\frac{1}{m}} \beta_4^{\frac{1}{n-1}}$
$false$	red	$false$	$\alpha_1^{\frac{1}{m}} \beta_5^{\frac{1}{n-1}}$
$false$	red	$true$	$\alpha_2^{\frac{1}{m}} \beta_6^{\frac{1}{n-1}}$
$true$	$green$	$false$	$\alpha_3^{\frac{1}{m}} \beta_1^{\frac{1}{n-1}}$
$true$	$green$	$true$	$\alpha_4^{\frac{1}{m}} \beta_2^{\frac{1}{n-1}}$
$true$	$orange$	$false$	$\alpha_3^{\frac{1}{m}} \beta_3^{\frac{1}{n-1}}$
$true$	$orange$	$true$	$\alpha_4^{\frac{1}{m}} \beta_4^{\frac{1}{n-1}}$
$true$	red	$false$	$\alpha_3^{\frac{1}{m}} \beta_5^{\frac{1}{n-1}}$
$true$	red	$true$	$\alpha_4^{\frac{1}{m}} \beta_6^{\frac{1}{n-1}}$

$\Big\rangle$. [3]

From Proposition 2.3 $\mathcal{J}(\Phi) = \mathcal{J}(\{[3]\})$.

2.5.2.3 Splitting, expanding and propositionalizing

Condition (M1) from Proposition 2.3 for parfactor multiplication requires sets of random variables represented by parameterized random variables from parfactor participating in a product to be identical or disjoint. It can be satisfied through *splitting* parfactors on substitutions and *expanding* counting formulas. These two operations modify parameterized random variables which affects sets of random variables the parameterized random variables represent.

The first proposition characterizes the splitting operation. It is Proposition 6 from Milch et al. [2008].

Proposition 2.4. Let $\Phi = \{g_1, g_2, \ldots, g_m\}$ be a set of parfactors. Let $g_i = \langle \mathcal{C}_i, \mathcal{V}_i, \mathcal{F}_i \rangle$ be a parfactor from Φ. Let X be a logical variable present in g_i. Let $\{X/t\}$ be a substitution such that $t \notin \mathcal{E}_X^{\mathcal{C}_i}$ and either term t is a constant $t \in \mathcal{D}(X)$, or term t is a logical variable present in g_i such that $\mathcal{D}(t) = \mathcal{D}(X)$. Let $g_i[X/t]$ be a parfactor g_i with all occurrences of X replaced by term t and $g_i' = \langle \mathcal{C}_i \cup \{X \neq t\}, \mathcal{V}_i, \mathcal{F}_i \rangle$. Then

$$\mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_i\} \cup \{g_i[X/t], g_i'\}). \tag{2.7}$$

We call the operation described above a *split* of parfactor g_i on substitution $\{X/t\}$ and we call g_i' a *residual* parfactor.

Example 2.15. Consider the following parfactor:

$\langle \{A \neq B\}, \{f(A,B), h(B)\},$

$f(A,B)$	$h(B)$	value
$false$	$false$	α_1
$false$	$true$	α_2
$true$	$false$	α_3
$true$	$true$	α_4

$\rangle$, [1]

where $range(f) = range(h) = \{false, true\}$, $\mathcal{D}(A) = \mathcal{D}(B) = \{x_1, x_2, \ldots, x_n\}$, and $\alpha_1, \ldots, \alpha_4$ are real numbers. Parfactor [1] represents $n(n-1)$ factors, its factor component has size 4. Let us split parfactor [1] on substitution $\{B/x_1\}$. We obtain two new parfactors:

$\langle \{A \neq x_1\}, \{f(A,x_1), h(x_1)\},$

$f(A,x_1)$	$h(x_1)$	value
$false$	$false$	α_1
$false$	$true$	α_2
$true$	$false$	α_3
$true$	$true$	α_4

$\rangle$ [1′]

and

$\langle \{A \neq B, B \neq x_1\}, \{f(A,B), h(B)\},$

$f(A,B)$	$h(B)$	value
$false$	$false$	α_1
$false$	$true$	α_2
$true$	$false$	α_3
$true$	$true$	α_4

$\rangle$. [1″]

Parfactor [1′] represents $n-1$ factors, parfactor [1″] represents $(n-1)+(n-1)(n-2) = (n-1)^2$ factors, their factor component has size 4. From Proposition 2.4, $\mathcal{J}(\{[1]\}) = \mathcal{J}(\{[1'], [1'']\})$.

Given a parfactor $g = \langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ and a logical variable X present in one or more parameterized random variables from $\mathcal{V}$, we may need to *propositionalize* g on X, that is replace it by a set of parfactors of the form $g[X/c]$, one for each constant c from $\mathcal{D}(X) : \mathcal{C}$. Propositionalization can be thought of as splitting g on substitution $\{X/c\}$ for every $c \in \mathcal{D}(X) : \mathcal{C}$.

The next proposition characterizes the expanding operation. It is Proposition 7 from Milch et al. [2008].

Proposition 2.5. Let $\Phi\{g_1, g_2, \ldots, g_m\}$ be a set of parfactors. Let $g_i = \langle \mathcal{C}_i, \mathcal{V}_i, \mathcal{F}_i \rangle$ be a normal-form parfactor from Φ and $\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]$ be a counting formula from $\mathcal{V}_i$. Let t be a term such that $t \notin \mathcal{E}_A^{\mathcal{C}_A}$ and $t \in \mathcal{E}_Y^{\mathcal{C}}$ for each logical variable $Y \in \mathcal{E}_A^{\mathcal{C}_A}$. Let $g' = \langle \mathcal{C}_i, \mathcal{V}_i', \mathcal{F}_i' \rangle$, where $\mathcal{V}_i' = \mathcal{V}_i \setminus \{\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]\} \cup \{f(\ldots, t, \ldots), \#_{A:\mathcal{C}_A \cup \{A \neq t\}}[f(\ldots, A, \ldots)]\}$ and factor $\mathcal{F}_i'$ is a factor from the Cartesian product of ranges of parameterized random variables in $\mathcal{V}_i'$ to the reals. Let $\mathbf{v}$ be a value assignment to to all random variables but $\mathit{ground}(f(\ldots, A, \ldots)) : \mathcal{C}_i \cup \mathcal{C}_A$. Factor $\mathcal{F}_i'$ is defined as follows:

$$\mathbf{v}(\mathcal{F}_i')(x, h()) = \mathbf{v}(\mathcal{F}_i)(h'()), \tag{2.8}$$

where $x \in \mathit{range}(f)$, histogram $h() \in \mathit{range}(\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[f(\ldots, A, \ldots)])$, histogram $h'() \in \mathit{range}(\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)])$, and $h'()$ is obtained by taking $h()$ and adding 1 to the count for the value x. Then

$$\mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_i\} \cup \{g_i'\}). \tag{2.9}$$

Example 2.16. Consider the following parfactor:

$\langle \emptyset, \{\#_{A:\{A \neq B\}}[f(A)], h(B)\},$

$\#_{A:\{A \neq B\}}[f(A)]$	$h(B)$	value
$(\#_{false} = n-1, \#_{true} = 0)$	$false$	α_1
$(\#_{false} = n-1, \#_{true} = 0)$	$true$	α_2
$(\#_{false} = n-2, \#_{true} = 1)$	$false$	α_3
$(\#_{false} = n-2, \#_{true} = 1)$	$true$	α_4
$\vdots$	$\vdots$	$\vdots$
$(\#_{false} = 0, \#_{true} = n-1)$	$false$	α_{2n-1}
$(\#_{false} = o, \#_{true} = n-1)$	$true$	α_{2n}

$\rangle$. [1]

where $\mathcal{D}(A) = \mathcal{D}(B) = \{x_1, x_2, \ldots, x_n\}$ and $\mathit{range}(f) = \mathit{range}(h) = \{false, true\}$, and $\alpha_1, \ldots, \alpha_{2n}$ are real numbers. Parfactor [1] represents n factors, its factor com-

ponent has size $2n$. Let us expand counting formula $\#_{A:\{A\neq B\}}[f(A)]$ on constant x_1. We obtain a new parfactor:

$\langle \emptyset, \{\#_{A:\{A\neq B, A\neq x_1\}}[f(A)], f(x_1), h(B)\},$

$\#_{A:\{A\neq B, A\neq x_1\}}[f(A)]$	$f(x_1)$	$h(B)$	value
$(\#_{false} = n-2, \#_{true} = 0)$	$false$	$false$	α_1
$(\#_{false} = n-2, \#_{true} = 0)$	$false$	$true$	α_2
$(\#_{false} = n-2, \#_{true} = 0)$	$true$	$false$	α_3
$(\#_{false} = n-2, \#_{true} = 0)$	$true$	$true$	α_4
$(\#_{false} = n-3, \#_{true} = 1)$	$false$	$false$	α_3
$(\#_{false} = n-3, \#_{true} = 1)$	$false$	$true$	α_4
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$(\#_{false} = 0, \#_{true} = n-2)$	$true$	$false$	α_{2n-1}
$(\#_{false} = o, \#_{true} = n-2)$	$true$	$true$	α_{2n}

$\rangle$. [1′]

Parfactor [1′] represents n factors, its factor component has size $4(n-1)$. From Proposition 2.5, $\mathcal{J}(\{[1]\}) = \mathcal{J}(\{[1']\})$.

We can fully expand a counting formula $\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]$ by expanding it on all constants in $\mathcal{D}(A) : \mathcal{C}_A$.

2.5.2.4 Counting

Condition (S2) of Proposition 2.1 and condition (SC2) of Proposition 2.2 require that the set of logical variables of a parameterized random variable being eliminated from a parfactor is a superset of the union of logical variables of other parameterized random variables from this parfactor. When this condition does not hold, we can sometimes satisfy it by performing counting. Counting eliminates a free logical variable from a parfactor. The next proposition characterizes the counting operation. It is Proposition 3 from Milch et al. [2008].

Proposition 2.6. Let $\Phi = \{g_1, g_2, \ldots, g_m\}$ be a set of parfactors. Let $g_i = \langle \mathcal{C}_i, \mathcal{V}_i, \mathcal{F}_i \rangle$ be a normal-form parfactor from Φ and A be a logical variable such that there is exactly one parameterized random variable $f(\ldots,A,\ldots) \in \mathcal{V}_i$ for which $A \in param(f(\ldots,A,\ldots))$. Let $\mathcal{C}_A$ be a set of those constraints from $\mathcal{C}_i$ that involve A, let $\mathcal{V}_i' = \mathcal{V}_i \setminus \{f(\ldots,A,\ldots)\} \cup \{\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]\}$ and let $\mathcal{F}_i'$ be a factor from the Cartesian product of ranges of parameterized random variables in $\mathcal{V}_i'$ to the reals. Let $\mathbf{v}$ be a value assignment to all random variables but $ground(f(\ldots,X,\ldots)) : \mathcal{C}_i$.

Factor $\mathcal{F}_i'$ is defined as follows:

$$\mathbf{v}(\mathcal{F}_i')(h()) = \prod_{x \in range(f(\ldots,A,\ldots))} \mathbf{v}(\mathcal{F}_i))(x)^{h(x)}, \tag{2.10}$$

where histogram $h() \in range(\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)])$. Then

$$\mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_i\} \cup \{\langle \mathcal{C}_i \setminus \mathcal{C}_A, \mathcal{V}_i', \mathcal{F}_i' \rangle\}). \tag{2.11}$$

As suggested in Milch et al. [2008], a good way to think about Equations 2.10 and 2.11 is to consider $\mathcal{J}(\{g_i\})$. We can represent g_i as a product of parfactors obtained from parfactor g_i by propositionalizing the logical variable A. In this product, given an assignment of values to random variables, it does not matter which ground instance of $f(\ldots,A,\ldots)$ is assigned a particular value from $range(f)$, but only how many of them are assigned the value. The above property allows us to use a counting formula $\#_{A:\mathcal{C}_A}[f(\ldots,A,\ldots)]$ to collapse the product to a single parfactor. While the new parfactor has bigger factor component than the original parfactor, the logical variable A no longer occurs free in the new parfactor.

Example 2.17. Consider the following parfactor:

$$\Big\langle \{A \neq B\}, \{f(A), h(B)\}, \begin{array}{c|c|c} f(A) & h(B) & \text{value} \\ \hline false & false & \alpha_1 \\ false & true & \alpha_2 \\ true & false & \alpha_3 \\ true & true & \alpha_4 \end{array} \Big\rangle, \quad [1]$$

where $range(f) = range(h) = \{false, true\}$, $\mathcal{D}(A) = \mathcal{D}(B) = \{x_1, x_2, \ldots, x_n\}$, and $\alpha_1, \ldots, \alpha_4$ are real numbers. Parfactor [1] represents $n(n-1)$ factors, its factor component has size 4. Note, that we cannot eliminate $ground(f(A)) : \{A \neq B\}$ from $\mathcal{J}(\{[1]\})$ in a lifted manner since condition (S1) of Proposition 2.1 is not satisfied. The same applies to lifted elimination of $ground(h(A)) : \{A \neq B\}$. We can however apply Proposition 2.6 and eliminate either of two logical variables present in the parfactor through counting. Let us eliminate the logical variable A. We create a new parfactor where $f(A)$ is replaced with a counting formula:

$$\left\langle \emptyset, \{\#_{A:\{A\neq B\}}[f(A)], h(B)\}, \right.$$

$\#_{A:\{A\neq B\}}[f(A)]$	$h(B)$	value
$(\#_{false} = n-1, \#_{true} = 0)$	$false$	$(\alpha_1)^{n-1}(\alpha_3)^0$
$(\#_{false} = n-1, \#_{true} = 0)$	$true$	$(\alpha_2)^{n-1}(\alpha_4)^0$
$(\#_{false} = n-2, \#_{true} = 1)$	$false$	$(\alpha_1)^{n-2}(\alpha_3)^1$
$(\#_{false} = n-2, \#_{true} = 1)$	$true$	$(\alpha_2)^{n-2}(\alpha_4)^1$
$\vdots$	$\vdots$	$\vdots$
$(\#_{false} = 0, \#_{true} = n-1)$	$false$	$(\alpha_1)^0(\alpha_3)^{n-1}$
$(\#_{false} = o, \#_{true} = n-1)$	$true$	$(\alpha_2)^0(\alpha_4)^{n-1}$

$$\left. \right\rangle . [1']$$

Parfactor $[1']$ represents n factors, its factor component has size $2n$. From Proposition 2.6, $\mathcal{J}(\{[1]\}) = \mathcal{J}(\{[1']\})$.

The counting elimination algorithm of de Salvo Braz et al. [2007] is equivalent to introducing counting formulas as described above and eliminating them immediately. The C-FOVE algorithm, by introducing counting formulas explicitly and allowing them to be part of parfactors increases flexibility of lifted inference (see Section 2.5.2.7).

2.5.2.5 Unification

Several conditions present in propositions that theoretically characterize operations on parfactors refer to sets of random variables represented by parameterized random variables. Conditions are concerned about sets of random variables being identical or disjoint. Checking these conditions through analysis of these sets would defeat the goal of lifted inference, that is carrying inference without propositionalizing a first-order probabilistic model or its part. We need syntactic criteria that can be checked quickly. Poole [2003] showed how these conditions can be ensured efficiently using *unification* [Sterling and Shapiro, 1994]. Milch et al. [2008] extended his work to counting formulas. In the presence of constraints in parfactors, unification needs to be accompanied by constraint analysis and processing.

In this section we give an overview of unification and constraint processing necessary to verify and ensure conditions discussed above. We start with two parameterized random variables that are not counting formulas. We illustrate our description with multiple examples.

Consider two parfactors $\langle \mathcal{C}_1, \mathcal{V}_1, \mathcal{F}_1 \rangle$ and $\langle \mathcal{C}_2, \mathcal{V}_2, \mathcal{F}_2 \rangle$. Assume that we want to find out the relation between sets of random variables represented by parameterized

```
[00]  procedure replace(⟨C,V,F⟩)
[01]    input: parfactor ⟨C,V,F⟩;
[02]    output: parfactor ⟨C,V,F⟩[θ], where substitution θ replaces every logical
[03]                    variable constrained to a single constant with this constant,
[04]            error if ⟨C,V,F⟩ represents 0 factors;
[05]    add logical variables from C to the queue;
[06]    while queue not empty do
[07]      remove logical variable X from the queue;
[08]      case
[09]        |D(X) : {X ≠ t ∈ C : t is a constant}| = 0
[10]          return error;
[11]        end
[12]        |D(X) : {X ≠ t ∈ C : t is a constant}| = 1
[13]          set {x} := D(X) : {X ≠ t ∈ C : t is a constant};
[14]          remove unary constraints involving X from C;
[15]          add logical variables from the set {Y : X ≠ Y ∈ C} to the queue;
[16]          replace every occurrence of X in C and in V with x;
[17]        end
[18]        otherwise
[19]          // do nothing
[20]      end
[21]    end
[22]    return ⟨C,V,F⟩;
[23]  end
```

Figure 2.8: AC-3 algorithm for replacing logical variables with constants.

random variables $f(t_1^1,\ldots,t_k^1) \in \mathcal{V}_1$ and $f(t_1^2,\ldots,t_k^2) \in \mathcal{V}_2$ in respective parfactors, that is, the relation between sets of random variables $ground(f(t_1^1,\ldots,t_k^1)) : \mathcal{C}_1$ and $ground(f(t_1^2,\ldots,t_k^2)) : \mathcal{C}_2$. If the sets are not disjoint and are not identical, we want to find out splits that are required to make these sets identical. Note that standard parameterized random variables with different functors represent disjoint sets of random variables.

Sometimes the constraints in a parfactor are tight enough to uniquely identify an individual from the population of a logical variable. In this case it is useful to be able to replace the logical variable with a constant denoting the only individual that is possible (we will justify the need for this step later on). Note that binary constraints we are dealing with are inequality constraints like $X \neq Y$. The $X \neq Y$ constraint is a weak constraint, it only constraints the set of possible values for Y,

when the set of possible values of X contains a single element. Figure 2.8 shows the AC-3 algorithm [Mackworth, 1977] adapted to perform the task described above. The algorithm does not enumerate populations of involved logical variables, but it needs to know their sizes.

Example 2.18. Consider a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$, where constrains set $\mathcal{C} = \{A \neq x_1, A \neq x_2, A \neq x_3, A \neq B, A \neq C, B \neq x_1, B \neq x_3, B \neq C\}$ and $\mathcal{D}(A) = \mathcal{D}(B) = \mathcal{D}(C) = \{x_1, x_2, x_3, x_4\}$. The algorithm presented in Figure 2.8 detects that population of the logical variable A is constrained to x_4 and replaces A with x_4 inside the parfactor. We obtain the following set of constraints: $\{x_4 \neq B, x_4 \neq C, B \neq x_1, B \neq x_3, B \neq C\}$. Now B is constrained to x_2 and can be replaced by this constant and the constraint set reduces to $\{x_4 \neq C, x_2 \neq C\}$. In total, the algorithm applied two substitutions, $\{A/x_4\}$ and $\{B/x_2\}$, to the parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$.

If we change the population of A, B and C to $\{x_1, x_2, x_3, x_4, x_5\}$, then none of the logical variables is constrained to a single individual and the algorithm from Figure 2.8 will not change the parfactor.

The scope of logical variables is restricted to the enclosing parfactor. Two logical variables with the same name present in two different parfactors might be typed with different populations and be completely unrelated. For unification to work properly we need to remove such name clashes between considered parfactors. We rename logical variables in both parfactors so that the set of logical variables present in the first parfactor is disjoint with the set of logical variables present in the second parfactor.

Example 2.19. Consider set of parfactors Φ_0:

$$\Phi_0 = \{ \langle \{X \neq x_2\}, \{f(X,Y), q(Y)\}, \mathcal{F}_1 \rangle, \quad [1]$$
$$\langle \{X \neq Z, Z \neq x_1\}, \{p(X), f(x_1, Z)\}, \mathcal{F}_2 \rangle \}, \quad [2]$$

where the logical variable X in parfactor [1] is unrelated to the logical variable X from parfactor [2] and we need to separate them. Assume that in parfactor [1] $\mathcal{D}(X) = \{x_1, x_2, \ldots, x_n\}$ and $\mathcal{D}(Y) = \{y_1, y_2, \ldots, y_m\}$, and in parfactor [2] $\mathcal{D}(X) = \mathcal{D}(Z) = \{y_1, y_2, \ldots, y_m\}$. If n and m are greater than 2, the algorithm from Figure 2.8 will not modify the two parfactors and we can proceed with renaming. We

```
[00]  procedure MGU(f(t_1^1,...,t_k^1), f(t_1^2,...,t_k^2))
[01]    input: parameterized random variables f(t_1^1,...,t_k^1), f(t_1^2,...,t_k^2);
[02]    output: MGU θ of f(t_1^1,...,t_k^1) and f(t_1^2,...,t_k^2),
[03]              error if f(t_1^1,...,t_k^1) and f(t_1^2,...,t_k^2) do not unify;
[04]    set θ := {};
[05]    push equations t_1^1 = t_1^2,...,t_k^1 = t_k^2 to the stack;
[06]    while stack not empty do
[07]      pop equation t_i = t_j from the stack;
[08]      case
[09]        t_i and t_j are identical terms
[10]          // do nothing
[11]        end
[12]        t_i is a parameter
[13]          replace every occurrence of t_i in the stack and in θ with t_j;
[14]          add {t_i/t_j} to θ;
[15]        end
[16]        t_j is a parameter
[17]          replace every occurrence of t_j in the stack and in θ with t_i;
[18]          add {t_j/t_i} to θ;
[19]        end
[20]        otherwise
[21]          return error;
[22]      end
[23]    end
[24]    return θ;
[25]  end
```

Figure 2.9: MGU algorithm for parameterized random variables.

obtain a new set of parfactors Φ_1:

$$\Phi_1 = \{\, \langle \{X_1 \neq x_2\}, \{f(X_1, X_2), q(X_2)\}, \mathcal{F}_1 \rangle, \qquad [3]$$

$$\langle \{X_3 \neq X_4, X_4 \neq x_1\}, \{p(X_3), f(x_1, X_4)\}, \mathcal{F}_2 \rangle\}, \qquad [4]$$

where $\mathcal{D}(X_1) = \{x_1, x_2, \ldots, x_n\}$ and $\mathcal{D}(X_2) = \mathcal{D}(X_3) = \mathcal{D}(X_4) = \{y_1, y_2, \ldots, y_m\}$. We have $\mathcal{J}(\Phi_0) = \mathcal{J}(\Phi_1)$.

Assume that we have already performed the above two preprocessing operations on parfactors $\langle \mathcal{C}_1, \mathcal{V}_1, \mathcal{F}_1 \rangle$ and $\langle \mathcal{C}_2, \mathcal{V}_2, \mathcal{F}_2 \rangle$. We can now compute an MGU of parameterized random variables $f(t_1^1, \ldots, t_k^1) \in \mathcal{V}_1$ and $f(t_1^2, \ldots, t_k^2) \in \mathcal{V}_2$. An algorithm for computing MGU is presented in Figure 2.9. The algorithm is adapted

```
[00]  procedure checkMGU(θ, C)
[01]    input: MGU θ,
[02]           set of inequality constraints C;
[03]    output: true if θ is consistent with C, false otherwise;
[04]    set θ := {};
[05]    for each constraint X ≠ t from C do
[06]      if {{X/t}} ⊂ θ
[07]        return false;
[08]      if t is a parameter
[09]        if {{X/t1}, {t/t1}} ⊂ θ or {{t/X}} ⊂ θ
[10]          return false;
[11]    end
[12]    return true;
[13]  end
```

Figure 2.10: Algorithm for checking if an MGU is consistent with a set of inequality constraints.

from Sterling and Shapiro [1994]. The time and space complexity of the algorithm are $\mathcal{O}(k)$.

If parameterized random variables do not unify, they represent disjoint sets of random variables. If the MGU is empty, neither parameterized random variable is parameterized by logical variables and each represents the same set containing a single random variable. The above is true because we have renamed logical variables and logical variables present in one parameterized random variable are not present in the other one. Finally, if parameterized random variables unify and the MGU is not empty, we need to check if the MGU is consistent with the constraints in both parfactors. A check if an MGU is consistent with a constraints can be performed using an algorithm shown in Figure 2.10.

Example 2.20. Let us continue Example 2.19. Parameterized random variables $f(X_1, X_2)$ and $f(x_1, X_4)$ unify with MGU $\theta = \{X_1/x_1, X_2/X_4\}$. θ is consistent with constraints $\{X_1 \neq x_2\} \cup \{X_3 \neq X_4, X_4 \neq x_1\}$.

If non-empty MGU is not consistent with the constraints, parameterized random variables $f(t_1^1, \ldots, t_k^1)$ and $f(t_1^2, \ldots, t_k^2)$ represent disjoint sets of random variables. If non-empty MGU is consistent with the constraints in both parfactors, sets of random variables represented by the analyzed parameterized random variables

```
[00] procedure splitOnMGU(θ, ⟨C, V, F⟩)
[01]   input: MGU θ,
[02]          parfactor ⟨C, V, F⟩;
[03]   output: parfactor g obtained by splitting ⟨C, V, F⟩ on θ,
[04]           set of residual parfactors Ψ;
[05]   set g := ⟨C, V, F⟩
[06]   set Ψ := {};
[07]   for each substitution {X/t} from θ do
[08]     if X is a logical variable in g
[09]       if t is a constant or t is a logical variable in g
[10]         split g on {X/t};
[11]         set g to the result;
[12]         add the residual parfactor to Ψ;
[13]       else
[14]         replace all occurrences of X in g with t;
[15]   end
[16]   return g and Ψ;
[17] end
```

Figure 2.11: Algorithm for splitting a parfactor on an MGU.

are not disjoint and possibly are not identical. To make them identical, we split both parfactors on the MGU as described by Poole [2003]. The splitting algorithm is shown in Figure 2.11. Given parfactor g and MGU θ the algorithm returns a parfactor obtained from applying substitutions in θ to g as well as residual parfactors resulting from these operations.

After applying the MGU to $\langle \mathcal{C}_1, \mathcal{V}_1, \mathcal{F}_1 \rangle$ and $\langle \mathcal{C}_2, \mathcal{V}_2, \mathcal{F}_2 \rangle$, we obtain sets of residual parfactors and parfactors $\langle \mathcal{C}'_1, \mathcal{V}'_1, \mathcal{F}_1 \rangle$ and $\langle \mathcal{C}'_2, \mathcal{V}'_2, \mathcal{F}_2 \rangle$. Thanks to the splitting on the MGU, parameterized random variables $f(t^1_1, \ldots, t^1_k)$ from $\mathcal{V}_1$ and $f(t^2_1, \ldots, t^2_k)$ from $\mathcal{V}_2$ are replaced in $\mathcal{V}'_1$ and $\mathcal{V}'_2$ by the same parameterized random variable $f(t_1, \ldots, t_k)$. However, we want sets $ground(f(t_1, \ldots, t_k)) : \mathcal{C}'_1$ and $ground(f(t_1, \ldots, t_k)) : \mathcal{C}'_2$ to be identical and we need to process constraints in $\mathcal{C}'_1$ and $\mathcal{C}'_2$. Note that, by splitting parfactors on the MGU, not only the sets of random variables represented by targeted parameterized random variables became identical, but also the symbolic representation of the parameterized random variables is identical in both parfactors. The latter brings us closer to satisfying condition (M2) of Proposition 2.6 and allowing for multiplication of these two parfactors.

```
[00]  procedure splitOnConstraints(C_S, ⟨C, V, F⟩)
[01]    input: set of constraints C_S,
[02]           parfactor ⟨C, V, F⟩;
[03]    output: parfactor g obtained by splitting θ to ⟨C, V, F⟩,
[04]            set of by-product parfactors Ψ;
[05]    set g := ⟨C, V, F⟩
[06]    set Ψ := {};
[07]    for each constraint X ≠ t from C_S do
[08]      if (X ≠ t) ∉ C and X is a logical variable in g
[09]                   and (t is a constant or t is a logical variable in g)
[10]          split g on {X/t};
[11]          add the result to Ψ;
[12]          set g to the residual parfactor;
[13]    end
[14]    return g and Ψ;
[15]  end
```

Figure 2.12: Algorithm for splitting a parfactor on a set of constraints.

Example 2.21. Let us continue Examples 2.19 and 2.20. We split parfactors from set Φ_1:

$$\Phi_1 = \{\, \langle \{X_1 \neq x_2\}, \{f(X_1, X_2), q(X_2)\}, \mathcal{F}_1 \rangle, \quad [3]$$
$$\langle \{X_3 \neq X_4, X_4 \neq x_1\}, \{p(X_3), f(x_1, X_4)\}, \mathcal{F}_2 \rangle\}, \quad [4]$$

on MGU $\theta = \{X_1/x_1, X_2/X_4\}$. We obtain the following set of parfactors:

$$\Phi_2 = \{\, \langle \{X_3 \neq X_4, X_4 \neq x_1\}, \{p(X_3), f(x_1, X_4)\}, \mathcal{F}_2 \rangle, \quad [4]$$
$$\langle \emptyset, \{f(x_1, X_4), q(X_4)\}, \mathcal{F}_1 \rangle, \quad [5]$$
$$\langle \{X_1 \neq x_1, X_1 \neq x_2\}, \{f(X_1, X_2), q(X_2)\}, \mathcal{F}_1 \rangle\} \quad [6]$$

Parfactor [4] is not affected by splitting on θ, because it does not contain logical variables X_1 and X_2. From parfactor [3] we obtain parfactor [5] and a residual parfactor [6]. We have $\mathcal{J}(\Phi_1) = \mathcal{J}(\Phi_2)$. Note that $ground(f(x_1, X_4)) : \emptyset \neq ground(f(x_1, X_4)) : \{X_1 \neq x_1, X_1 \neq x_2\}$ and the set of random variables represented by parameterized random variable $f(x_1, X_4)$ in parfactor [4] is different from the set of random variables it represents in parfactor [5].

The final step processes constraints. Let $\mathcal{S}$ be a set of logical variables that are shared between parfactor $\langle \mathcal{C}_1', \mathcal{V}_1', \mathcal{F}_1 \rangle$ and parfactor $\langle \mathcal{C}_2', \mathcal{V}_2', \mathcal{F}_2 \rangle$.

Sets of random variables $ground(f(t_1,\ldots,t_k)) : \mathcal{C}_1'$ and $ground(f(t_1,\ldots,t_k)) : \mathcal{C}_2'$ might be different, because inequality constraints that have impact on the set of random variables represented by $f(t_1,\ldots,t_k)$ might be different in parfactors $\langle \mathcal{C}_1', \mathcal{V}_1', \mathcal{F}_1 \rangle$ and $\langle \mathcal{C}_2', \mathcal{V}_2', \mathcal{F}_2 \rangle$. These are constraints between a logical variable and a constant such that the logical variable is in $\mathcal{S}$ and constraints between two logical variables such that both logical variables are in $\mathcal{S}$. We can ignore constraints that do not involve logical variables from $\mathcal{S}$. Earlier, we replaced logical variables constrained to single individuals with appropriate constants (see Figure 2.8), and now we can ignore binary constraints that involve a logical variable from $\mathcal{S}$ and a logical variable not from $\mathcal{S}$.

An algorithm presented in Figure 2.12 inputs a set of constraints and a parfactor and modifies the parfactor so that it contains all relevant constraints from the given set of constraints. The algorithm also returns a set of by-product parfactors. Given $\langle \mathcal{C}_1', \mathcal{V}_1', \mathcal{F}_1 \rangle$ and $\langle \mathcal{C}_2', \mathcal{V}_2', \mathcal{F}_2 \rangle$, we apply the algorithm to set of constraints $\mathcal{C}_2'$ and parfactor $\langle \mathcal{C}_1', \mathcal{V}_1', \mathcal{F}_1 \rangle$ and obtain a parfactor $\langle \mathcal{C}_1\text{''}, \mathcal{V}_1\text{''}, \mathcal{F}_1 \rangle$. Then we apply the algorithm to set of constraints $\mathcal{C}_1'$ and parfactor $\langle \mathcal{C}_2', \mathcal{V}_2', \mathcal{F}_2 \rangle$ and obtain a parfactor $\langle \mathcal{C}_2\text{''}, \mathcal{V}_2\text{''}, \mathcal{F}_2 \rangle$. We have $ground(f(t_1,\ldots,t_k)) : \mathcal{C}_1\text{''} = ground(f(t_1,\ldots,t_k)) : \mathcal{C}_2\text{''}$, which was the goal of the process of unification.

Example 2.22. Let us continue Example 2.21. We have set of parfactors Φ_2:

$$\Phi_2 = \{ \langle \{X_3 \neq X_4, X_4 \neq x_1\}, \{p(X_3), f(x_1, X_4)\}, \mathcal{F}_2 \rangle, \quad [4]$$
$$\langle \emptyset, \{f(x_1, X_4), q(X_4)\}, \mathcal{F}_1 \rangle, \quad [5]$$
$$\langle \{X_1 \neq x_1, X_1 \neq x_2\}, \{f(X_1, X_2), q(X_2)\}, \mathcal{F}_1 \rangle \}. \quad [6]$$

We will use the algorithm from Figure 2.12 to modify parfactors [4] and [5] so that parameterized random variable $f(x_1, X_4)$ represent the same set of random variables in the modified parfactors. Since parfactor [5] does not contain any constraints, we only need to split parfactor [5] on constraints from parfactor [4]. The algorithm will ignore constraint $X_3 \neq X_4$ as the logical variable X_3 is not present in [5] and split on substitution $\{X_4/x_1\}$ induced by constraint $X_4 \neq x_1$. We obtain a

new set of parfactors:

$$\begin{aligned}\Phi_3 = \{\, &\langle\{X_3 \neq X_4, X_4 \neq x_1\}, \{p(X_3), f(x_1, X_4)\}, \mathcal{F}_2\rangle, && [4]\\ &\langle\{X_1 \neq x_1, X_1 \neq x_2\}, \{f(X_1, X_2), q(X_2)\}, \mathcal{F}_1\rangle && [6]\\ &\langle\emptyset, \{f(x_1, x_1), q(X_4)\}, \mathcal{F}_1\rangle, && [7]\\ &\langle\{X_4 \neq x_1\}, \{f(x_1, X_4), q(X_4)\}, \mathcal{F}_1\rangle\}. && [8]\end{aligned}$$

Parameterized random variable $f(x_1, X_4)$ represent the same set of random variables in parfactors [4] and [8]. We also have $\mathcal{J}(\Phi_2) = \mathcal{J}(\Phi_3)$.

Above, we talked only about standard parameterized random variables. Assume we want to find out relation between sets of random variables represented by a counting formula $\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]$ from a parfactor $\langle\mathcal{C}, \mathcal{V}, \mathcal{F}\rangle$ and another parameterized random variable (possibly also a counting formula, but not necessarily). The process is quite similar to what we described for standard parameterized random variables with two differences pointed out by Milch et al. [2008]. The first difference is that we should apply unification to $f(\ldots, A, \ldots)$ and combine constraints $\mathcal{C}_A$ with $\mathcal{C}$ instead of considering $\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]$ itself. It is because the set of random variables underlying $f(\ldots, A, \ldots)$ with respect to constraints $\mathcal{C}_A \cup \mathcal{C}$ is the same as for $\#_{A:\mathcal{C}_A}[f(\ldots, A, \ldots)]$ with constraints $\mathcal{C}$, even though they have different functors (see Section 2.4.1.1). The second difference is that where procedure called for splitting parameterized random variables, we expand counting formulas.

In this section we described how to compare and change sets of random variables represented by parameterized random variables. The described procedure operates at a syntactic level and it does not enumerate populations of involved logical variables, it only needs to know their sizes.

2.5.2.6 The C-FOVE algorithm

Milch et al. [2008] describe the C-FOVE algorithm in terms of the following *macro-operations*:

- SHATTER(Φ) - given a set of parfactors Φ, the shattering macro-operation performs all the splits and expansions (defined in Section 2.5.2.3) necessary to ensure that for any two parameterized random variables present in

parfactors from Φ, the sets of random variables represented by these two parameterized random variables are either identical or disjoint. The necessary splits in expansions are determined using unification as described in Section 2.5.2.5;

- GLOBAL-SUM-OUT(Φ, $f(\dots)$, $\mathcal{C}$) - given a set of parfactors Φ, a parameterized random variable $f(\dots)$ and a set of constraints $\mathcal{C}$, the macro-operation multiplies all parfactors from Φ containing parameterized random variables that represent $ground(f(\dots)) : \mathcal{C}$ (see Section 2.5.2.2) and eliminates random variables $ground(f(\dots)) : \mathcal{C}$ from the product (see Section 2.5.2.1); the macro-operation is only applicable if the product satisfies condition (S2) of Proposition 2.1 (or condition (SC2) of Proposition 2.2 if we are eliminating a counting formula), multiplication operations can always be performed because of initial shattering, this is the only macro-operation that eliminates random variables from $\mathcal{J}(\Phi)$;

- COUNTING-CONVERT(Φ, $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$, X) - given a set of parfactors Φ, a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ from Φ and a free logical variable X, such that X occurs in exactly one parameterized random variable from $\mathcal{V}$, the counting macro-operation eliminates X from the parfactor (see Section 2.5.2.4);

- PROPOSITIONALIZE(Φ, $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$, X) - given a set of parfactors Φ, a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ from Φ and a free logical variable X, the macro-operation propositionalizes the logical variable X (see Section 2.5.2.3), afterwards it performs shattering to ensure that the sets of random variables represented by parameterized random variables in Φ are still either identical or disjoint after the propositionalization;

- FULL-EXPAND(Φ, $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$, $\#_{A:\mathcal{C}_A}[f(\dots, A, \dots)]$) - given a set of parfactors Φ, a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ from Φ and a counting formula $\#_{A:\mathcal{C}_A}[f(\dots, A, \dots)]$, the macro-operation expands the given counting formula on every constant in $\mathcal{D}(A) : \mathcal{C}_A$ (see Section 2.5.2.3), afterwards it performs shattering to ensure that the sets of random variables represented by parameterized random variables in Φ are still either identical or disjoint after the expansion;

Given a set of parfactors Φ and a set of queried random variables $\mathbf{Q}$, which could be given in a form of a parameterized random variable and a set of constraints, the C-FOVE algorithm computes the marginal $\mathcal{J}_{\mathbf{Q}}(\Phi)$. It starts with a shattering macro-operation. Parfactors in Φ are also shattered against the random variables in Q. Next the C-FOVE algorithm eliminates non-queried random variables from Φ by iteratively performing one of the other four macro-operations. The macro-operations are chosen by a greedy-search with the cost of each operation defined as the total size of parfactors an operation creates, which in practice is equal to the size of factor components of these parfactors. While global elimination and counting operations are not always possible to perform because their preconditions might not be satisfied, propositionalization and full expansion can always be executed and, in an extreme case, fully propositionalize the set of parfactors. This implies C-FOVE's completeness.

It is worth pointing out, that similarly to the VE algorithm for inference in belief networks, the C-FOVE algorithm in directed models allows for pruning of random variables irrelevant to the query. See [Taghipour et al., 2009] for an example how pruning can be performed in lifted manner.

2.5.2.7 Example computation

In this section we present a simple lifted computation which illustrates the C-FOVE algorithm.

Consider the parfactors from Example 2.7 (page 23), which represent the ICL theory from Example 2.6 (page 18) and Figure 2.5 (page 19). Assume $\mathcal{D}(Lot) = \{lot_1, lot_2, \ldots, lot_n\}$ and that it is observed that grass is wet on lot_1, which can be represented by the following parfactor:

$$\langle \emptyset, \{wet\_grass(lot_1)\},\ \begin{array}{c|c} \mathcal{P}(wet\_grass(lot_1) = false) & \mathcal{P}(wet\_grass(lot_1) = true) \\ \hline 0.0 & 1.0 \end{array} \rangle.$$

Let Φ be a set of the three parfactors from Example 2.7 and the above parfactor (in what follows, we don't show details of factor components of parfactors):

$$\begin{aligned}
\Phi = \{ &\langle \emptyset, \{rain()\}, \mathcal{F}_1 \rangle, && [01]\\
&\langle \emptyset, \{sprinkler(Lot)\}, \mathcal{F}_2 \rangle, && [02]\\
&\langle \emptyset, \{rain(), sprinkler(Lot), wet_grass(Lot)\}, \mathcal{F}_3 \rangle, && [03]\\
&\langle \emptyset, \{wet_grass(lot_1)\}, \mathcal{F}_4 \rangle \}. && [04]
\end{aligned}$$

Assume we want to compute $\mathcal{J}_{ground(wet\_grass(Lot)):\{Lot \neq lot_1\}}(\Phi)$ using the C-FOVE algorithm. Note that this is the joint on $wet\_grass(Lot)$ for all lots except lot_1. Below we describe a run of the C-FOVE algorithm. After each step we show an updated set of parfactors Φ indexed with the step number. After initial shattering, before each step we list available macro-operations and their cost according to the C-FOVE heuristic.

First, C-FOVE invokes SHATTER(Φ) macro-operation. The macro-operation splits parfactor [03] on substitution $\{Lot/lot_1\}$ which creates parfactor [05] and residual parfactor [06]. Next, it splits parfactor [02] on $\{Lot/lot_1\}$ which creates parfactor [07] and residual parfactor [08]. After shattering, set Φ_1 is as follows:

$$\begin{aligned}
\Phi_1 = \{ &\langle \emptyset, \{rain()\}, \mathcal{F}_1 \rangle, && [01]\\
&\langle \emptyset, \{wet_grass(lot_1)\}, \mathcal{F}_4 \rangle, && [04]\\
&\langle \emptyset, \{rain(), sprinkler(lot_1), wet_grass(lot_1)\}, \mathcal{F}_3 \rangle, && [05]\\
&\langle \{Lot \neq lot_1\}, \{rain(), sprinkler(Lot), wet_grass(Lot)\}, \mathcal{F}_3 \rangle, && [06]\\
&\langle \emptyset, \{sprinkler(lot_1)\}, \mathcal{F}_2 \rangle, && [07]\\
&\langle \{Lot \neq lot_1\}, \{sprinkler(Lot)\}, \mathcal{F}_2 \rangle \}. && [08]
\end{aligned}$$

Note that parfactors created by the same splitting operation have identical factor components. A smart implementation would represent these factors using the same, single object. We discuss this issue in more detail in Section 5.3.

The following (parameterized random variable, set of constraints) pairs are present in set Φ_1: $(rain(), \emptyset), (sprinkler(lot_1), \emptyset), (sprinkler(Lot), \{Lot \neq lot_1\})$,

$(wet\_grass(lot_1), \emptyset)$, and $(wet\_grass(Lot), \{Lot \neq lot_1\})$. C-FOVE will eliminate all of them, except for the last one.

After the initial shattering, the C-FOVE algorithm has choice between performing the following macro-operations:

- GLOBAL-SUM-OUT(Φ_1, $wet\_grass(lot_1)$, $\emptyset$), which during the multiplication step would create a factor of size 8 and after summing out would create a factor of size 4; it would eliminate one random variable;
- GLOBAL-SUM-OUT(Φ_1, $sprinkler(lot_1)$, $\emptyset$), which during the multiplication step would create a factor of size 8 and after summing out would create a factor of size 4; it would eliminate one random variable;
- GLOBAL-SUM-OUT(Φ_1, $sprinkler(Lot)$, $\{Lot \neq lot_1\}$), which during the multiplication step would create a factor of size 8 and after summing out would create a factor of size 4; it would eliminate $n-1$ random variables;
- PROPOSITIONALIZE(Φ_1, [06], Lot), which would create $n-1$ identical factors of size 2 and, because of subsequent shattering, $n-1$ identical factors of size 2; it would not eliminate any random variables;
- PROPOSITIONALIZE(Φ_1, [08], Lot), which would create $n-1$ identical factors of size 8 and $n-1$ identical factors of size 2; it would not eliminate any random variables;
- COUNTING-CONVERT(Φ_1, [08], Lot), which would create a factor of size n; it would not eliminate any random variables.

The first three macro-operations have identical cost. Tie-breaking is not discussed by Milch et al. [2008]. Let us assume that C-FOVE uses the number of random variables that would be eliminated for tie-breaking (the more, the better) and if the number does not resolve the tie than the algorithm chooses one of equally good operations at random. Under this assumption, C-FOVE chooses the third macro-operation. The macro-operation first applies Proposition 2.3 and multiplies parfactors that involve $ground(sprinkler(Lot)) : \{Lot \neq lot_1\}$, that is, parfactors [06] and [08]. Both parfactors represent the same number of factors, namely $n-1$,

and no correction is necessary. The product is as follows:

$$\langle \{Lot \neq lot_1\}, \{rain(), sprinkler(Lot), wet\_grass(Lot)\}, \mathcal{F}_2 \odot \mathcal{F}_3 \rangle .$$

The macro-operation applies Proposition 2.1 and sums out $ground(sprinkler(Lot))$: $\{Lot \neq lot_1\}$ from the above product. No logical variable disappears from the parfactor and C-FOVE does not need to compensate for it. C-FOVE obtains the following parfactor:

$$\langle \{Lot \neq lot_1\}, \{rain(), wet\_grass(Lot)\}, \mathcal{F}_5 \rangle, \qquad [09]$$

where $\mathcal{F}_5 = \sum_{sprinkler(Lot)} \mathcal{F}_2 \odot \mathcal{F}_3$. The new set of parfactors is as follows:

$$\begin{aligned}
\Phi_2 = \{ & \langle \emptyset, \{rain()\}, \mathcal{F}_1 \rangle, && [01] \\
& \langle \emptyset, \{wet_grass(lot_1)\}, \mathcal{F}_4 \rangle, && [04] \\
& \langle \emptyset, \{rain(), sprinkler(lot_1), wet_grass(lot_1)\}, \mathcal{F}_3 \rangle, && [05] \\
& \langle \emptyset, \{sprinkler(lot_1)\}, \mathcal{F}_2 \rangle, && [07] \\
& \langle \{Lot \neq lot_1\}, \{rain(), wet_grass(Lot)\}, \mathcal{F}_5 \rangle \}. && [09]
\end{aligned}$$

Next, C-FOVE has choice between performing the following macro-operations:

- GLOBAL-SUM-OUT(Φ_2, $wet\_grass(lot_1)$, $\emptyset$), which during the multiplication step would create a factor of size 8 and after summing out would create a factor of size 4; it would eliminate one random variable;
- GLOBAL-SUM-OUT(Φ_2, $sprinkler(lot_1)$, $\emptyset$), which during the multiplication step would create a factor of size 8 and after summing out would create a factor of size 4; it would eliminate one random variable;
- PROPOSITIONALIZE(Φ_2, [09], Lot), which would create $n-1$ identical factors of size 4; it would not eliminate any random variables;
- COUNTING-CONVERT(Φ_2, [09], Lot), which would create a factor of size $2n$; it would not eliminate any random variables.

The first two macro-operations have identical cost. Let us assume that C-FOVE randomly chooses the second one. GLOBAL-SUM-OUT(Φ_2, $wet\_grass(lot_1)$, $\emptyset$) first multiplies parfactors [05] and [07] and obtains the following product:

$$\langle \emptyset, \{rain(), sprinkler(lot_1), wet\_grass(lot_1)\}, \mathcal{F}_2 \odot \mathcal{F}_3 \rangle .$$

The macro-operation sums out $ground(sprinkler(lot_1))$ from the above product and obtains the following parfactor:

$$\langle \emptyset, \{rain(), sprinkler(lot_1)\}, \sum_{sprinkler(lot_1)} \mathcal{F}_2 \odot \mathcal{F}_3 \rangle . \qquad [10]$$

Note that $= \sum_{sprinkler(lot_1)} \mathcal{F}_2 \odot \mathcal{F}_3 = \mathcal{F}_5$. Factor $\mathcal{F}_5$ has already been computed by the previous macro-operation. This overhead is due to shattering, which takes a brute-force approach to constraint processing. We discuss this issue in Section 5.3. The new set of parfactors is as follows:

$$\begin{aligned}
\Phi_3 = \{ & \langle \emptyset, \{rain()\}, \mathcal{F}_1 \rangle, && [01] \\
& \langle \emptyset, \{wet_grass(lot_1)\}, \mathcal{F}_4 \rangle, && [04] \\
& \langle \{Lot \neq lot_1\}, \{rain(), wet_grass(Lot)\}, \mathcal{F}_5 \rangle, && [09] \\
& \langle \emptyset, \{rain(), wet_grass(lot_1)\}, \mathcal{F}_5 \rangle \}. && [10]
\end{aligned}$$

Next, C-FOVE has choice between performing the following macro-operations:

- GLOBAL-SUM-OUT(Φ_3, $wet\_grass(lot_1)$, $\emptyset$), which during the multiplication step would create a factor of size 4 and after summing out would create a factor of size 2; it would eliminate one random variable;
- PROPOSITIONALIZE(Φ_3, [09], Lot), which would create $n-1$ identical factors of size 4; it would not eliminate any random variables;
- COUNTING-CONVERT(Φ_3, [09], Lot), which would create a factor of size $2n$; it would not eliminate any random variables.

C-FOVE chooses the first macro-operation. The macro-operation multiplies parfactors [04] and [10]. Their product is as follows:

$$\langle \emptyset, \{rain(), wet\_grass(lot_1)\}, \mathcal{F}_4 \odot \mathcal{F}_5 \rangle .$$

The macro-operation sums out $ground(wet\_grass(lot_1))$ from the above product and obtains the following parfactor:

$$\langle \emptyset, \{rain()\}, \mathcal{F}_6 \rangle, \qquad [11]$$

where $\mathcal{F}_6 = \sum_{wet\_grass(lot_1)} \mathcal{F}_4 \odot \mathcal{F}_5$. The new set of parfactors is as follows:

$$\begin{aligned} \Phi_4 = \{ & \langle \emptyset, \{rain()\}, \mathcal{F}_1 \rangle, && [01] \\ & \langle \{Lot \neq lot_1\}, \{rain(), wet\_grass(Lot)\}, \mathcal{F}_5 \rangle, && [09] \\ & \langle \emptyset, \{rain()\}, \mathcal{F}_6 \rangle \}. && [11] \end{aligned}$$

Next, C-FOVE can perform only one macro-operation:

- COUNTING-CONVERT(Φ_4, [09], *Lot*), which creates a factor of size $2n$; it does not eliminate any random variables.

This macro-operation applies Proposition 2.6 and performs counting on logical variable *Lot*. It obtains:

$$\langle \emptyset, \{rain(), \#_{Lot:\{Lot \neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_7 \rangle, \qquad [12]$$

where $\mathcal{F}_7$ is defined as in Equation 2.10. The new set of parfactors is as follows:

$$\begin{aligned} \Phi_5 = \{ & \langle \emptyset, \{rain()\}, \mathcal{F}_1 \rangle, && [01] \\ & \langle \emptyset, \{rain()\}, \mathcal{F}_6 \rangle, && [11] \\ & \langle \emptyset, \{rain(), \#_{Lot:\{Lot \neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_7 \rangle \}. && [12] \end{aligned}$$

A call to the SHATTER(Φ_5) macro-operation does not change the set Φ_5.

Next, C-FOVE has choice between performing the following macro-operations:

- GLOBAL-SUM-OUT(Φ_5, $rain()$, $\emptyset$), which during multiplication step would create a factor of size $2n$ and after summing out would create a factor of size n; it would eliminate one random variable;
- FULL-EXPAND(Φ_5, [12], $\#_{Lot:\{Lot\neq lot_1\}}[wet\_grass(Lot)]$), which would create a factor of size 2^n; it would not eliminate any random variables.

C-FOVE chooses the first macro-operation. The macro-operation multiplies parfactors [01], [11] and [12]. Their product is as follows:

$$\langle \emptyset, \{rain(), \#_{Lot:\{Lot\neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_1 \odot \mathcal{F}_6 \odot \mathcal{F}_7 \rangle .$$

Next, C-FOVE sums out $ground(rain())$ from the above product and obtains the following parfactor:

$$\langle \emptyset, \{\#_{Lot:\{Lot\neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_8 \rangle, \qquad [13]$$

where $\mathcal{F}_8 = \sum_{rain()} \mathcal{F}_1 \odot \mathcal{F}_6 \odot \mathcal{F}_7$. The new set of parfactors is as follows:

$$\Phi_6 = \{ \langle \emptyset, \{\#_{Lot:\{Lot\neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_8 \rangle \}. \qquad [13]$$

We have

$$\mathcal{J}_{ground(wet\_grass(Lot)):\{Lot\neq lot_1\}}(\Phi) = \mathcal{J}(\Phi_6) .$$

During computation, constants $lot_2, lot_3, \ldots, lot_n$ were not explicitly enumerated, we only needed to know that $\mathcal{D}(Lot) = n$. The biggest factor created during inference had size 8 (factor $\mathcal{F}_3$) for $n \leq 4$ and $2n$ (factor $\mathcal{F}_7$) for $n > 4$.

2.6 Summary

In this chapter we presented a brief overview of belief networks and the variable elimination algorithm. Belief networks and probabilistic inference in belief networks is discussed in great detail in [Darwiche, 2009] and [Koller and Friedman,

2009] as well as in general AI textbooks [Poole and Mackworth, 2010; Russell and Norvig, 2009].

Next, we discussed first-order probabilistic models. As focus of this thesis is on inference, rather than modeling, we gave only one example of first-order probabilistic formalism: ICL. Collections by Getoor and Taskar [2007] and De Raedt et al. [2008] contain chapters devoted to many other first-order probabilistic languages. Milch [2006] provides a very good summary of first-order probabilistic languages.

Finally, we gave an overview of current state of the art in exact lifted probabilistic inference.

While approximate lifted inference is not the focus of this thesis, it is worth mentioning that there is an ongoing effort in designing approximate lifted inference algorithms [de Salvo Braz et al., 2009; Kersting et al., 2009; Sen et al., 2009; Singla and Domingos, 2008].

In chapters to follow, we will present our contributions to this area. Our work aims to satisfy desiderata already mentioned in his chapter:

- the length of a specification of a first-order probabilistic model must be independent of the sizes of the populations in the model;
- the cost of inference should be logarithmic when possible and at most at most linear in the sizes of the populations in the model.

Chapter 3

Aggregation in Lifted Inference

Here's something to think about: How come you never see a headline like 'Psychic Wins Lottery'? — Jay Leno

3.1 Introduction

One aspect that arises in directed first-order probabilistic models is the need for aggregation that occurs when a parent parameterized random variable has logical variables that are not present in a child parameterized random variable. Previously available lifted inference algorithms do not allow a description of aggregation in first-order models that is independent of the sizes of the populations. In this thesis we introduce a new data structure, the *aggregation parfactor*, describe how to use it to represent aggregation in first-order models, and show how to perform efficient lifted inference in its presence.

We analyze the need for aggregation (Section 3.2) and describe how to model aggregation (Section 3.3) in directed first-order probabilistic models. Next, in Section 3.4, we introduce a new data structure, aggregation parfactors, and describe how to perform lifted inference in its presence. For clarity, we start with a simple form of aggregation parfactors and later present a generalized version (Section 3.4.3). In Section 3.5, through experiments, we show that aggregation parfactors can lead to gains in efficiency.

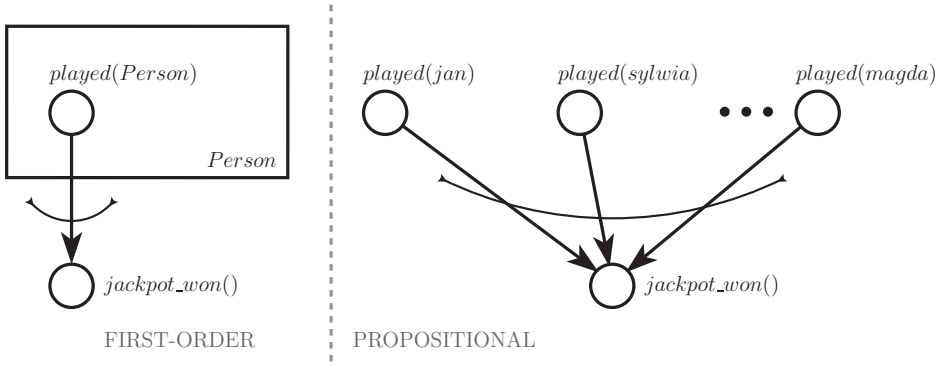


Figure 3.1: A first-order model from Example 3.1 and its equivalent belief network. Aggregation is denoted by curved arcs.

3.2 Need for aggregation

In a directed first-order probabilistic model, when a child parameterized random variable has a parent parameterized random variable with extra logical variables, in the grounding the child parameterized random variable has an unbounded number of parents. We need some aggregation operator to describe how the child parameterized random variable depends on the parent parameterized random variable. We illustrate this point with the following two simple examples.

Example 3.1. Consider the directed first-order probabilistic model and its grounding presented in Figure 3.1. The model is meant to represent that a person who fills in a single 6/49 lottery ticket has a chance of guessing correctly all six numbers and that when it happens, the jackpot is won. A person who does not play the lottery has no chances of winning. The model has two nodes: a parameterized random variable $played(Person)$ with range $\{false, true\}$, and a random variable $jackpot\_won()$ with range $\{false, true\}$ that is $true$ if some person guesses correctly all six numbers. We have $\mathcal{D}(Person) = \{jan, sylwia, \ldots, magda\}$ and $|\mathcal{D}(Person)| = n$.

A parameterized random variable $played(Person)$ represents the n random variables in the corresponding propositional model. Therefore, in the propositional model, the number of parent nodes influencing the node $jackpot\_won()$ is n. Their common effect aggregates in the child node.

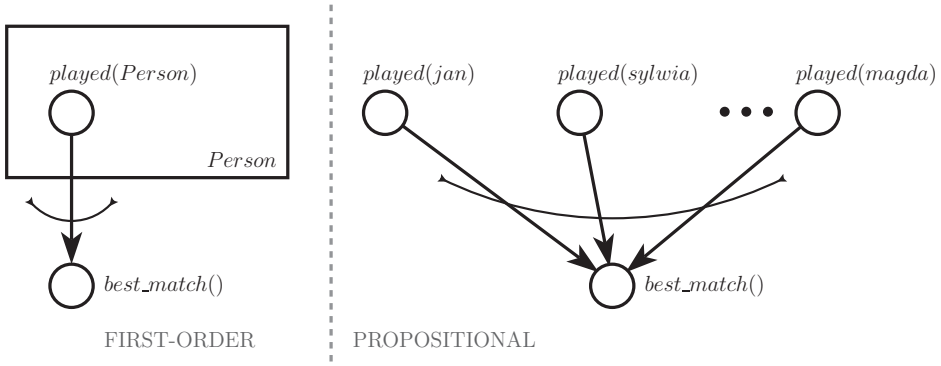


Figure 3.2: A first-order model from Example 3.2 and its equivalent belief network. Aggregation is denoted by curved arcs.

The next example illustrates aggregation over non-binary random variables.

Example 3.2. Consider the directed first-order probabilistic model and its grounding presented in Figure 3.2. It is a modified model from Example 3.1. A person who plays a 6/49 lottery has a chance of matching correctly zero, one, ..., five, or all six numbers. The model has two nodes: a parameterized random variables $played(Person)$ with range $\{false, true\}$, and a random variable $best\_match()$ with range $\{0,1,2,3,4,5,6\}$ that is equal to the highest number of matched lottery numbers among lottery participants.

3.3 Modeling aggregation

In this thesis we base aggregation on causal independence. We introduce causal independence in Section 3.3.1 and show how to use it to describe aggregation in Section 3.3.2.

3.3.1 Causal independence

We use the definition of causal independence from Zhang and Poole [1996].

Definition 3.1. Parent random variables $p_1, p_2, \ldots, p_n$ are said to be *causally independent* with respect to child random variable c if there exist random variables $\widetilde{p}_1, \widetilde{p}_2, \ldots, \widetilde{p}_n$ such that:

1. $\forall_{i\in[1,n]}$ range of $\widetilde{p}_i$ is a subset of the range of c, and

2. $\forall_{i\in[1,n]}$ $\widetilde{p}_i$ probabilistically depends on p_i and $\widetilde{p}_i$ is independent of $p_1,\ldots,$ $p_{i-1},p_{i+1},\ldots,p_n,\widetilde{p}_1,\ldots,\widetilde{p}_{i-1},\widetilde{p}_{i+1},\ldots,\widetilde{p}_n$ given p_i, and

3. there exists a commutative and associative binary operator $*$ over the range of c such that $c = \widetilde{p}_1 * \widetilde{p}_2 * \cdots * \widetilde{p}_n$.

We call $*$ the *base combination operator* of c.

The above definition covers common causal independence models such as noisy-OR [Pearl, 1986], noisy-MAX [Díez, 1993] and noisy-adders as special cases. The next example illustrates how Definition 3.1 can be used to describe a noisy-OR model.

Example 3.3. The *noisy-OR model* consists of a Boolean child node c and a set of n Boolean parents nodes $\mathbf{p} = \{p_1, p_2, \ldots, p_n\}$. Associated with each parent node p_i is its *causal strength* s_i, which gives the probability that the c is $true$ when p_i is $true$ independently of the values of other parents. Let $true(\mathbf{p})$ be a set of parent nodes which are $true$, then the conditional probability distribution specified by the noisy-or is given by:

$$\mathcal{P}(c = true|\mathbf{p}) = 1 - \prod_{i\in true(\mathbf{p})} (1 - s_i) \tag{3.1}$$

and

$$\mathcal{P}(c = false|\mathbf{p}) = \prod_{i\in true(\mathbf{p})} (1 - s_i). \tag{3.2}$$

Given Definition 3.1 the noisy-OR model can be described using n Boolean random variables $\widetilde{p}_i$, n conditional probability distributions:

p_i	$\mathcal{P}(\widetilde{p}_i = false)$	$\mathcal{P}(\widetilde{p}_i = true)$
$false$	1	0
$true$	$1 - s_i$	s_i

$, i \in [1,n],$

and the logical OR operator as the base combination operator.

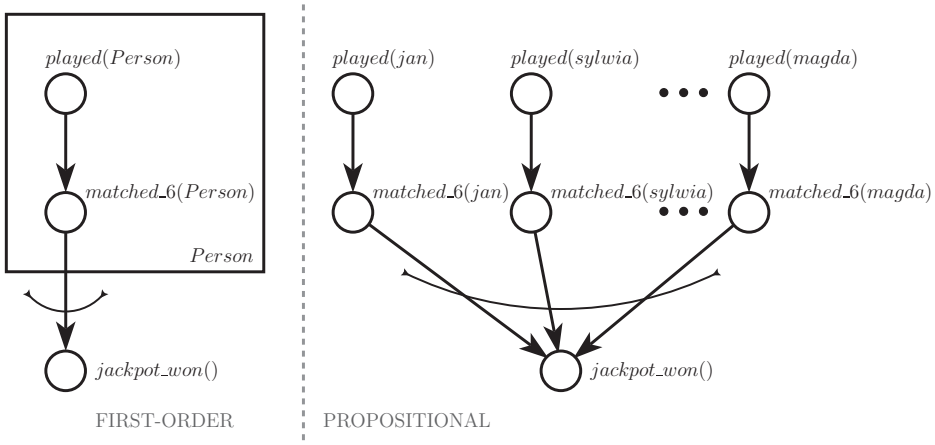


Figure 3.3: A first-order model with OR-based aggregation from Example 3.4 and its equivalent belief network.

3.3.2 Causal independence-based aggregation

In this thesis, to describe aggregation we assume that the range of the parent parameterized random variable is a subset of the range of the child parameterized random variable, and use a commutative and associative deterministic binary operator over the range of the child parameterized random variable as an aggregation operator $\otimes$. Given probabilistic input to the parent parameterized random variable, we can construct any causal independence model covered by the definition of causal independence from Section 3.3.1. In other words, this allows any causal independence model to act as underlying mechanism for aggregation in directed first-order models.

While some first-order probabilistic formalisms allow for richer forms of aggregation, see for example work of Jaeger [2002] on *combination functions*, the above mechanism allows us to satisfy desiderata stated in Section 2.6 and, as we will see in Section 3.4, integrate aggregation into calculus of parfactors.

Example 3.4. Let us come back to Example 3.1. We add to the model a parameterized random variable $matched\_6(Person)$ with range $\{false, true\}$. It is a noisy version of the parameterized random variable $played(Person)$. If $played(Person)$ has value $false$, then $matched\_6(Person)$ also has value $false$. If $played(Person)$

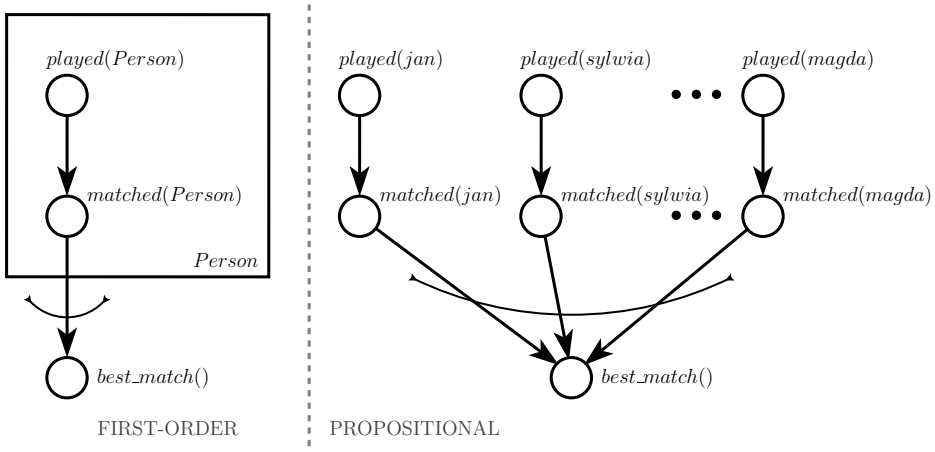


Figure 3.4: A first-order model with MAX-based aggregation from Example 3.4 and its equivalent belief network.

has value *true*, then $matched\_6(Person)$ has value *true* with probability equal to a chance of guessing correctly all six numbers. The common effect of random variables represented by $matched\_6(Person)$ aggregates in $jackpot\_won()$. We use logical OR as an aggregation operator to describe the (deterministic) conditional probability distribution $\mathcal{P}(jackpot\_won()|matched\_6(Person))$. The modified model is presented in Figure 3.3.

Similarly, we can adapt the model from Example 3.2 by adding a parameterized random variable $matched(Person)$ with range $\{0,1,2,3,4,5,6\}$ and using the MAX operator as an aggregation operator to describe the probability distribution $\mathcal{P}(best\_match()|matched(Person))$. The modified model is presented in Figure 3.4.

Aggregation can also be used with more abstract operators.

Example 3.5. Consider the directed first-order probabilistic model and its grounding presented in Figure 3.5. The model has three nodes: parameterized random variables $played(Person)$ and $matched\_6(Person)$, both with range $\{false, true\}$, and a random variable $jackpot\_winners()$ with range $\{0,1,2,$ *many*$\}$ that represents the number of lottery participants that correctly matched all six numbers.

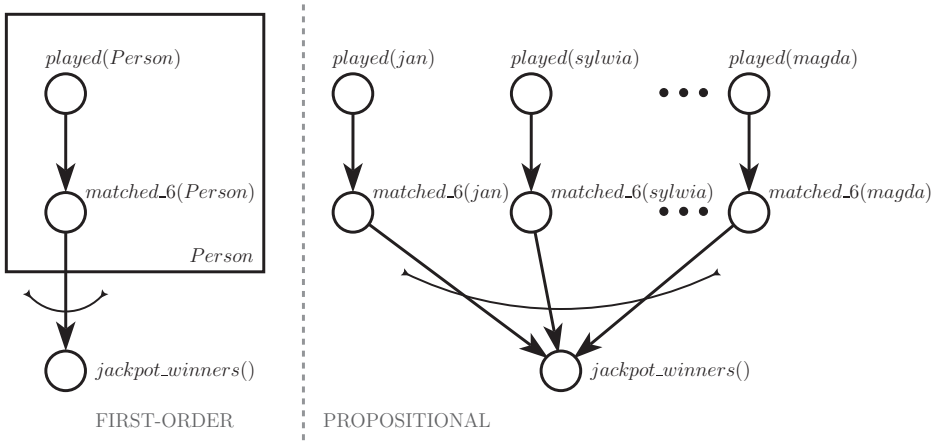


Figure 3.5: A first-order model from Example 3.5 and its equivalent belief network.

Let us define $\mathrm{SUM}_{|3} : \{0,1,2,many\} \times \{0,1,2,many\} \to \{0,1,2,many\}$ as follows:

$$\mathrm{SUM}_{|3}(x,y) = \begin{cases} x+y, & \text{if } x,y \in \{0,1,2\} \text{ and } x+y \leq 2; \\ many, & \text{otherwise.} \end{cases}$$

The operator $\mathrm{SUM}_{|3}$ can be interpreted as sum capped at 3. We use $\mathrm{SUM}_{|3}$ as an aggregation operator to describe $\mathcal{P}(jackpot\_winners() | matched\_6(Person))$.

In this thesis we require that the directed first-order probabilistic models satisfy the following conditions:

(1) for each parameterized random variable, its parent has at most one extra logical variable

(2) if a parameterized random variable $c(\dots)$ has a parent $p(\dots,A,\dots)$ with an extra logical variable A, then:

   (a) $p(\dots,A,\dots)$ is the only parent of $c(\dots)$

   (b) the range of p is a subset of the range of c

   (c) $c(\dots)$ is a deterministic function of the parent: $c(\dots) = p(\dots,a_1,\dots) \otimes \dots \otimes p(\dots,a_n,\dots) = \bigotimes_{a \in \mathcal{D}(A)} p(\dots,a,\dots)$, where $\otimes$ is a commutative and associative deterministic binary operator over the range of c.

At first the above conditions seem to be very restrictive, but they in fact are not. There is no need to define the aggregation over more than one logical variable due to the associativity and commutativity of the $\otimes$ operator. We can obtain more complicated distributions by introducing auxiliary parameterized random variables and combining multiple aggregations.

Example 3.6. Consider a parent parameterized random variable $p(A,B,C)$ and a child parameterized random variable $c(C)$. We can describe a $\otimes$-based aggregation over A and B, $c(C) = \bigotimes_{(a,b)\in\mathcal{D}(A)\times\mathcal{D}(B)} p(A,B,C)$ using an auxiliary parameterized random variable $c'(B,C)$ such that c' has the same range as c. Let $c'(B,C) = \bigotimes_{a\in\mathcal{D}(A)} p(A,B,C)$, then $c(C) = \bigotimes_{b\in\mathcal{D}(B)} c'(B,C)$.

Similarly, with the use of auxiliary nodes, we can construct a distribution that combines an aggregation with influence from other parent nodes or even combines multiple aggregations generated with different operators.

In the rest of the chapter, we assume that the discussed models satisfy conditions (1) and (2), for ease of presentation and with no loss of generality.

3.4 Aggregation parfactors

In this section we introduce a new data structure, aggregation parfactors, describe how to use it to represent aggregation in first-order models, and show how to perform efficient lifted inference in its presence. We need to introduce a new data structure, because parfactors, including parfactors on counting formulas, are not adequate representations, as their size would depend on the population size of the extra logical variable.

Example 3.7. Consider the first-order model presented in Figure 3.3, which is also discussed in Examples 3.1 and 3.4. We cannot represent the conditional probability distribution $\mathcal{P}(jackpot\_won()|matched\_6(Person))$ with a parfactor $\langle\emptyset,\ \{matched\_6(Person),\ jackpot\_won()\},\ \mathcal{F}\rangle$ as even simple noisy-OR cannot be represented as a product of factors represented by this parfactor.

A parfactor $\langle\emptyset,\{matched\_6(jan),\ldots,matched\_6(magda),jackpot\_won()\},\mathcal{F}\rangle$ is not an adequate input representation of this distribution because its size would depend on $|\mathcal{D}(Person)|$. The same applies to $\langle\emptyset,\{\#_{Person:\emptyset}[matched\_6(Person)]$,

jackpot_won()}, $\mathcal{F}\rangle$ as the size of the range of $\#_{Person:\emptyset}[matched\_6(Person)]$ depends on $|\mathcal{D}(Person)|$.

An analogous problem arises for the first-order models presented in Figures 3.4 and 3.5.

Definition 3.2.
An *aggregation parfactor* is a hextuple $\langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$, where

- $p(\ldots,A,\ldots)$ and $c(\ldots)$ are parameterized random variables
- the range of p is a subset of the range of c
- A is the only logical variable in $p(\ldots,A,\ldots)$ that is not in $c(\ldots)$
- $\mathcal{C}$ is a set of inequality constraints not involving A
- $\mathcal{F}_p$ is a factor from the range of p to real numbers
- $\otimes$ is a commutative and associative deterministic binary operator over the range of c
- $\mathcal{C}_A$ is a set of inequality constraints involving A, such that $(\mathcal{D}(A) : \mathcal{C}_A) \neq \emptyset$.

An aggregation parfactor $\langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ represents a set of factors, one factor $\mathcal{F}_{pc}$ for each ground substitution $\mathcal{G}$ to all logical variables in $param(c(\ldots))$ that satisfies the constraints in $\mathcal{C}$. Each factor $\mathcal{F}_{pc}$ is a mapping from the Cartesian product $\times_{a \in \mathcal{D}(A):\mathcal{C}_A} range(p) \times range(c)$ to the reals, which, given an assignment of values to random variables $\mathbf{v}$, is defined as follows:

$$\mathcal{F}_{pc}(\mathbf{v}(p(\ldots,a_1,\ldots)),\ldots,\mathbf{v}(p(\ldots,a_n,\ldots)),\mathbf{v}(c(\ldots))) =$$
$$\begin{cases} \prod\limits_{a \in \{a_1,\ldots,a_n\}} \mathcal{F}_p{}^{\frac{r_p}{r_c}}(\mathbf{v}(p(\ldots,a,\ldots))), & \text{if } \bigotimes\limits_{a \in \{a_1,\ldots,a_n\}} \mathbf{v}(p(\ldots,a,\ldots)) = \mathbf{v}(c(\ldots)); \\ 0, & \text{otherwise,} \end{cases}$$

where $\mathcal{D}(A) : \mathcal{C}_A = \{a_1,\ldots,a_n\}$, $r_p = |ground(p(\ldots,a,\ldots)) : \mathcal{C}|$, $a \in \mathcal{D}(A) : \mathcal{C}_A$ and $r_c = |ground(c(\ldots)) : \mathcal{C}|$.

The space required to represent an aggregation parfactor does not depend on the size of the set $\mathcal{D}(A) : \mathcal{C}_A$. It is $\mathcal{O}(n^2 \log n)$ where n is the size of $range(c)$, as the operator $\otimes$ can be represented as a factor from $range(c) \times range(c)$ to $range(c)$.

Exponent $\frac{r_p}{r_c}$ compensates for the possibility that parameterized random variable $c(\ldots)$ might be parameterized by logical variables not present in $p(\ldots,A,\ldots)$.

It is also important to notice that $\mathcal{D}(A):\mathcal{C}_A$ might vary for different ground substitutions $\mathcal{G}$ if the set $\mathcal{C}\cup\mathcal{C}_A$ is not in normal form (see Section 2.5.1.1). We comment on this issue in Section 5.2.3.

When an aggregation parfactor $\langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A\rangle$ is used to describe aggregation in a first-order model, the factor $\mathcal{F}_p$ will be a constant function with the value 1. We use $\mathbf{1}$ to denote such factors. An aggregation parfactor created during inference may have a non-trivial $\mathcal{F}_p$ component (see Section 3.4.2).

Example 3.8. Consider the first-order model from Figure 3.3. The conditional probability distribution $\mathcal{P}(jackpot\_won()|matched\_6(Person))$ can be represented with an aggregation parfactor $\langle \emptyset, matched\_6(Person), jackpot\_won(), \mathbf{1}, \mathrm{OR}, \emptyset\rangle$. The size of the representation does not depend on the population size of the logical variable *Person*. Parameterized random variable $jackpot\_won()$ represents one random variable therefore the aggregation parfactor represents one factor:

$matched\_6(jan)$	$\ldots$	$matched\_6(magda)$	$jackpot\_won()$	value
$false$	$\ldots$	$false$	$false$	1
$false$	$\ldots$	$false$	$true$	0
$false$	$\ldots$	$true$	$false$	0
$false$	$\ldots$	$true$	$true$	1
$\vdots$	$\vdots\vdots\vdots$	$\vdots$	$\vdots$	$\vdots$
$true$	$\ldots$	$true$	$false$	0
$true$	$\ldots$	$true$	$true$	1

.

Similarly, for the model from Figure 3.4, $\mathcal{P}(best\_match()|matched(Person))$ can be represented with $\langle \emptyset, matched(Person), best\_match(), \mathbf{1}, \mathrm{MAX}, \emptyset\rangle$, for the model from from Figure 3.5, $\mathcal{P}(jackpot\_winners()|matched\_6(Person))$ can be represented with $\langle \emptyset, matched\_6(Person), jackpot\_winners(), \mathbf{1}, \mathrm{SUM}_{|3}, \emptyset\rangle$.

In the rest of this thesis, Φ denotes a set of parfactors and aggregation parfactors. The notation introduced in Section 2.5 remains valid under the extended meaning of the symbol Φ.

3.4.1 Conversion to parfactors

In this section we show how aggregation parfactors can be converted to parfactors that in turn can be used during inference with C-FOVE.

3.4.1.1 Conversion using counting formulas

Consider an aggregation parfactor $\langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$. Since $\otimes$ is an associative and commutative operator, given an assignment of values to random variables $\mathbf{v}$, it does not matter which of the parameterized random variables $p(\dots,a,\dots)$, $a \in \mathcal{D}(A) : \mathcal{C}_A$ are assigned each value from $range(p)$, but only how many of them are assigned each value. This property was a motivation for the *counting elimination* algorithm [de Salvo Braz et al., 2007] (see Section 2.5.2.4) and counting formulas [Milch et al., 2008] (see Section 2.4.1.1), and allows us to convert aggregation parfactors to a product of two parfactors, where one of the parfactors is a parfactor on a counting formula:

Proposition 3.1. Let $g_A = \langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ be an aggregation parfactor from Φ such that set $\mathcal{C} \cup \mathcal{C}_A$ is in normal form. Let $\mathcal{F}_\#$ be a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)]) \times range(c)$ to $\{0,1\}$. Given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\dots,A,\dots)) : \mathcal{C} \cup \mathcal{C}_A$, the function is defined as follows:

$$\mathcal{F}_\#(h(), \mathbf{v}(c(\dots))) = \begin{cases} 1, & \text{if } \bigotimes_{x \in range(p)} \bigotimes_{i=1}^{h(x)} x = \mathbf{v}(c(\dots)); \\ 0, & \text{otherwise,} \end{cases}$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)])$.
Then

$$\mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_A\} \cup \{ \langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p \rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots)\}, \mathcal{F}_\# \rangle \}).$$

Proof. It suffices to show that

$$\mathcal{J}(\{g_A\}) = \mathcal{J}(\{\langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle\}) \tag{3.3}$$

equals

$$\mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p \rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots)\}, \mathcal{F}_\# \rangle\}). \tag{3.4}$$

We are going to transform expression (3.4) into expression (3.3) by propositionalizing logical variable A. Let $\mathcal{D}(A) : \mathcal{C}_A = \{a_1, a_2, \ldots, a_n\}$.

$$\begin{aligned}
&\mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\ldots, A, \ldots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots, A, \ldots)], c(\ldots)\}, \mathcal{F}_\#\rangle\}) = \\
&\qquad \text{(by } n \text{ applications of Proposition 2.4)} \\
&= \mathcal{J}(\{\langle \mathcal{C}, \{p(\ldots, a_1, \ldots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C}, \{p(\ldots, a_2, \ldots)\}, \mathcal{F}_p\rangle, \ldots, \\
&\qquad \langle \mathcal{C}, \{p(\ldots, a_n, \ldots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots, A, \ldots)], c(\ldots)\}, \mathcal{F}_\#\rangle\}\}) = \\
&\qquad \text{(by } n-1 \text{ applications of Proposition 2.3)} \\
&= \mathcal{J}(\{\langle \mathcal{C}, \{p(\ldots, a_1, \ldots), p(\ldots, a_2, \ldots), \ldots, p(\ldots, a_n, \ldots)\}, \mathcal{F}_1\rangle, \\
&\qquad \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots, A, \ldots)], c(\ldots)\}, \mathcal{F}_\#\rangle\}\}), \qquad (3.5)
\end{aligned}$$

where factor $\mathcal{F}_1$ is a mapping from the Cartesian product $\times_{a \in \mathcal{D}(A):\mathcal{C}_A} range(p)$ to the reals, which, given an assignment of values to random variables $\mathbf{v}$, is defined as follows:

$$\mathbf{v}(\mathcal{F}_1) = \prod_{a \in \mathcal{D}(A):\mathcal{C}_A} \mathcal{F}_p(\mathbf{v}(p(\ldots, a, \ldots))).$$

Next, we apply n times Proposition 2.5 to expression (3.5) and obtain:

$$\begin{aligned}
&\mathcal{J}(\{\langle \mathcal{C}, \{p(\ldots, a_1, \ldots), p(\ldots, a_2, \ldots), \ldots, p(\ldots, a_n, \ldots)\}, \mathcal{F}_1\rangle, \\
&\qquad \langle \mathcal{C}, \{p(\ldots, a_1, \ldots), p(\ldots, a_2, \ldots), \ldots, p(\ldots, a_n, \ldots), c(\ldots)\}, \mathcal{F}_2\rangle\}). \qquad (3.6)
\end{aligned}$$

Factor $\mathcal{F}_2$ is a mapping from the Cartesian product $\times_{a \in \mathcal{D}(A):\mathcal{C}_A} range(p) \times range(c)$ to the reals, which, given an assignment of values to random variables $\mathbf{v}$, is defined as follows:

$$\mathbf{v}(\mathcal{F}_2) = \begin{cases} 1, & \text{if } \bigotimes_{a \in \mathcal{D}(A):\mathcal{C}_A} \mathbf{v}(p(\ldots, a, \ldots)) = \mathbf{v}(c(\ldots)); \\ 0, & \text{otherwise.} \end{cases}$$

Finally, we apply Proposition 2.3 to parfactors from expression (3.6) and obtain:

$$\mathcal{J}(\{\langle \mathcal{C}, \{p(\ldots, a_1, \ldots), p(\ldots, a_2, \ldots), \ldots, p(\ldots, a_n, \ldots), c(\ldots)\}, \mathcal{F}_1{}^{\frac{r_p}{r_c}} \odot \mathcal{F}_2\rangle\}),$$

where $r_p = |\,ground(p(\ldots,a,\ldots)):\mathcal{C}\,|$, $a \in \mathcal{D}(A):\mathcal{C}_A$, and $r_c = |\,ground(c(\ldots)):\mathcal{C}\,|$. Parfactor $\langle \mathcal{C}, \{p(\ldots,a_1,\ldots), p(\ldots,a_2,\ldots), \ldots, p(\ldots,a_n,\ldots), c(\ldots)\}, \mathcal{F}_1 \overset{\frac{r_p}{r_c}}{\odot} \mathcal{F}_2 \rangle$ represents a set of factors, one for each ground substitution $\mathcal{G}$ to all logical variables in $param(p(\ldots,A,\ldots)) \cup param(c(\ldots)) \setminus \{A\} = param(c(\ldots))$ that satisfies the constraints in $\mathcal{C}$. Each factor $\mathcal{F}_\mathcal{G}$ is a mapping from the Cartesian product $\times_{a \in \mathcal{D}(A):\mathcal{C}_A} range(p) \times range(c)$ to the reals, which, given an assignment of values to random variables $\mathbf{v}$, is defined as follows:

$$\mathbf{v}(\mathcal{F}_\mathcal{G}) = \begin{cases} \prod\limits_{a \in \mathcal{D}(A):\mathcal{C}_A} \mathcal{F}_p{}^{\frac{r_p}{r_c}}(\mathbf{v}(p(\ldots,a,\ldots))), & \text{if } \bigotimes\limits_{a \in \mathcal{D}(A):\mathcal{C}_A} \mathbf{v}(p(\ldots,a,\ldots)) = \mathbf{v}(c(\ldots)); \\ 0, & \text{otherwise.} \end{cases}$$

From Definition 3.2 such set of factors can be represented as an aggregation parfactor $\langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$. Therefore

$$\mathcal{J}(\{\langle \mathcal{C}, \{p(\ldots,a_1,\ldots), p(\ldots,a_2,\ldots), \ldots, p(\ldots,a_n,\ldots), c(\ldots)\}, \mathcal{F}_3 \rangle\}) =$$
$$\mathcal{J}(\{\langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle\}),$$

which completes the proof. □

If the set $\mathcal{C} \cup \mathcal{C}_A$ is not in normal form (Section 2.5.1.1) we will need to use the splitting operation described in Section 3.4.2.1 to convert the aggregation parfactor to a set of aggregation parfactors with constraint sets in normal form (see algorithm presented in Figure 5.4 on page 162).

The Proposition 3.1 shows how an aggregation parfactor can be replaced by a parfactors involving a counting formula. The following example illustrates how it is done in practice.

Example 3.9. Consider the aggregation parfactor introduced in Example 3.8:

$$\langle \emptyset, matched\_6(Person), jackpot\_won(), \mathbf{1}, \text{OR}, \emptyset, \rangle.$$

Let $n = |\mathcal{D}(Person)|$. The conversion described in Proposition 3.1 creates two parfactors:

$$\langle \emptyset, \{matched\_6(Person)\}, \mathbf{1} \rangle \quad \text{and}$$
$$\langle \emptyset, \{\#_{Person:\emptyset}[matched\_6(Person)], jackpot\_won()\}, \mathcal{F}_{\#} \rangle,$$

$\mathcal{F}_{\#}$ is a factor from the Cartesian product $range(\#_{Person:\emptyset}[matched\_6(Person)]) \times range(jackpot\_won)$ to $\{0,1\}$. The range of $\#_{Person:\emptyset}[matched\_6(Person)]$ is a set of histograms $h()$ with a bucket for value $false$, a bucket for value $true$ and entries adding up to n:

$$\{(\#_{false} = n, \#_{true} = 0), (\#_{false} = n-1, \#_{true} = 1), \ldots, (\#_{false} = 0, \#_{true} = n)\}.$$

The range of $jackpot\_won$ is $\{false, true\}$. The factor $\mathcal{F}_{\#}$ is defined as follows:

$$\mathcal{F}_{\#}(h(), \mathbf{v}(jackpot\_won())) = \begin{cases} 1, & \text{if } \underset{x \in \{false,true\}}{\text{OR}} \overset{h(x)}{\underset{i=1}{\text{OR}}} x = \mathbf{v}(jackpot\_won()); \\ 0, & \text{otherwise.} \end{cases}$$

Note that the expression $\underset{x \in \{false,true\}}{\text{OR}} \overset{h(x)}{\underset{i=1}{\text{OR}}} x$ evaluates to $false$ if $h(false) = n$ and to $true$ if $h(false) < n$. Below we show a tabular representation of factor $\mathcal{F}_{\#}$:

$\#_{Person:\emptyset}[matched\_6(Person)]$	$jackpot\_won()$	value
$(\#_{false} = n \quad, \#_{true} = 0)$	$false$	1
$(\#_{false} = n \quad, \#_{true} = 0)$	$true$	0
$(\#_{false} = n-1, \#_{true} = 1)$	$false$	0
$(\#_{false} = n-1, \#_{true} = 1)$	$true$	1
$\vdots$	$\vdots$	$\vdots$
$(\#_{false} = 0 \quad, \#_{true} = n)$	$false$	0
$(\#_{false} = 0 \quad, \#_{true} = n)$	$true$	1

.

The size of the factor $\mathcal{F}_{\#}$ is $n+1$.

3.4.1.2 Conversion for MAX and MIN operators

If in an aggregation parfactor $\otimes$ is the MAX operator (which includes the OR operator as a special case), we can use a factorization presented by Díez and Galán

[2003] to convert the aggregation parfactor to parfactors without counting formulas. The factorization is an example of the *tensor rank-one decomposition* of a conditional probability distribution [Savicky and Vomlel, 2007].

The factorization of Díez and Galán can be used to convert an aggregation parfactor to a pair of standard parfactors as follows:

Proposition 3.2. Let $g_A = \langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \mathrm{MAX}, \mathcal{C}_A \rangle$ be an aggregation parfactor from Φ, where MAX operator is induced by a total ordering $\prec$ of $range(c)$ and set $\mathcal{C} \cup \mathcal{C}_A$ is in normal form. Let $\mathbf{s}()$ be a successor function induced by $\prec$. Let $c'(\dots)$ be an auxiliary parameterized random variable with the same parameterization and the same range as c. Let $\mathcal{F}_c$ be a factor from the Cartesian product $range(p) \times range(c)$ to real numbers that, given an assignment of values to random variables $\mathbf{v}$, is defined as follows:

$$\mathcal{F}_c(\mathbf{v}(p(\dots,A,\dots)), \mathbf{v}(c'(\dots))) = \begin{cases} {\mathcal{F}_p}^{\frac{r_c}{r_p}}(\mathbf{v}(p(\dots,A,\dots))), \\ \qquad\qquad \text{if } \mathbf{v}(p(\dots,A,\dots)) \preccurlyeq \mathbf{v}(c'(\dots)); \\ 0, \qquad \text{otherwise,} \end{cases} \tag{3.7}$$

where $r_p = |\, ground(p(\dots,a,\dots)) : \mathcal{C}\,|$, $a \in \mathcal{D}(A) : \mathcal{C}_A$, and $r_c = |\, ground(c(\dots)) : \mathcal{C}\,|$. Let $\mathcal{F}_\Delta$ be a factor from the Cartesian product $range(p) \times range(c)$ to $\{-1, 0, 1\}$ that, given $\mathbf{v}$, is defined as follows:

$$\mathcal{F}_\Delta(\mathbf{v}(c(\dots)), \mathbf{v}(c'(\dots))) = \begin{cases} 1, & \text{if } \mathbf{v}(c(\dots)) = \mathbf{v}(c'(\dots)); \\ -1, & \text{if } \mathbf{v}(c(\dots)) = \mathbf{s}(\mathbf{v}(c'(\dots))); \\ 0, & \text{otherwise.} \end{cases}$$

Then

$$\mathcal{J}(\Phi) = \sum_{ground(c'(\dots))} \mathcal{J}(\Phi \setminus \{g_A\} \cup \{\, \langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots), c'(\dots)\}, \mathcal{F}_c \rangle, \\ \langle \mathcal{C}, \{c(\dots), c'(\dots)\}, \mathcal{F}_\Delta \rangle\}).$$

Proof. It is sufficient to prove that

$$\mathcal{J}(\{g_A\}) = \mathcal{J}(\{\langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \text{MAX}, \mathcal{C}_A\rangle\}) \tag{3.8}$$

is equal to

$$\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C}\cup\mathcal{C}_A, \{p(\dots,A,\dots), c'(\dots)\}, \mathcal{F}_c\rangle, \langle \mathcal{C}, \{c(\dots), c'(\dots)\}, \mathcal{F}_\Delta\rangle\}). \tag{3.9}$$

We start with expression (3.8) and convert the aggregation parfactor to parfactors. By Proposition 3.1

$$\begin{aligned}
&\mathcal{J}(\{\langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \text{MAX}, \mathcal{C}_A\rangle\}) = \\
&\mathcal{J}(\{\langle \mathcal{C}\cup\mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots)\}, \mathcal{F}_\#\rangle\}).
\end{aligned}$$

$\mathcal{F}_\#$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)]) \times range(c)$ to a set $\{0,1\}$ that, given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\dots,A,\dots)) : \mathcal{C}\cup\mathcal{C}_A$, is defined as follows:

$$\mathcal{F}_\#(h(), \mathbf{v}(c(\dots))) = \begin{cases} 1, & \text{if } \max\limits_{x\in range(p), h(x)>0} x = \mathbf{v}(c(\dots)); \\ 0, & \text{otherwise,} \end{cases}$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)])$.

Next, we transform expression (3.9) and represent the first parfactor as a product of two parfactors. From (3.7) and Proposition 2.3

$$\begin{aligned}
&\mathcal{J}(\{\langle \mathcal{C}\cup\mathcal{C}_A, \{p(\dots,A,\dots), c'(\dots)\}, \mathcal{F}_c\rangle) = \\
&\mathcal{J}(\{\langle \mathcal{C}\cup\mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p{}^{\frac{r_c}{r_p}\frac{r_p}{r_c}}\rangle, \langle \mathcal{C}\cup\mathcal{C}_A, \{p(\dots,A,\dots), c'(\dots)\}, \mathcal{F}_1\rangle) = \\
&\mathcal{J}(\{\langle \mathcal{C}\cup\mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C}\cup\mathcal{C}_A, \{p(\dots,A,\dots), c'(\dots)\}, \mathcal{F}_1\rangle),
\end{aligned}$$

where $\mathcal{F}_1$ is a factor from the Cartesian product $range(p) \times range(c)$ to real numbers that, given an assignment of values to random variables $\mathbf{v}$, is defined as fol-

lows:

$$\mathcal{F}_1(\mathbf{v}(p(\ldots,A,\ldots)),\mathbf{v}(c'(\ldots))) = \begin{cases} 1, & \text{if } \mathbf{v}(p(\ldots,A,\ldots)) \preccurlyeq \mathbf{v}(c'(\ldots)); \\ 0, & \text{otherwise.} \end{cases}$$

After applying the above transformations to expressions (3.8) and (3.9) we are reduced to proving that

$$\begin{aligned} &\mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\ldots,A,\ldots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots)\}, \mathcal{F}_{\#}\rangle\}) = \\ &\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\ldots,A,\ldots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C} \cup \mathcal{C}_A, \{p(\ldots,A,\ldots), c'(\ldots)\}, \mathcal{F}_1\rangle, \\ &\qquad \langle \mathcal{C}, \{c(\ldots), c'(\ldots)\}, \mathcal{F}_\Delta\rangle\}). \end{aligned}$$

The above is equivalent to proving that

$$\begin{aligned} &\mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots)\}, \mathcal{F}_{\#}\rangle\}) = \\ &\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\ldots,A,\ldots), c'(\ldots)\}, \mathcal{F}_1\rangle, \langle \mathcal{C}, \{c(\ldots), c'(\ldots)\}, \mathcal{F}_\Delta\rangle\}). \end{aligned} \tag{3.10}$$

We are going to transform the right hand side of the Equation 3.10 into the left hand side. We start with counting over logical variable A. By Proposition 2.6

$$\begin{aligned} &\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\ldots,A,\ldots), c'(\ldots)\}, \mathcal{F}_1\rangle, \langle \mathcal{C}, \{c(\ldots), c'(\ldots)\}, \mathcal{F}_\Delta\rangle\}) = \\ &\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c'(\ldots)\}, \mathcal{F}_2\rangle, \langle \mathcal{C}, \{c(\ldots), c'(\ldots)\}, \mathcal{F}_\Delta\rangle\}). \end{aligned}$$

$\mathcal{F}_2$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)]) \times range(c')$ to a set $\{0,1\}$ that, given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\ldots,A,\ldots)) : \mathcal{C} \cup \mathcal{C}_A$, is defined as follows:

$$\mathcal{F}_2(h(),\mathbf{v}(c'(\ldots))) = \begin{cases} 1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \preccurlyeq \mathbf{v}(c'(\ldots)); \\ 0, & \text{otherwise.} \end{cases}$$

Next, we perform multiplication. By Proposition 2.3

$$\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c'(\ldots)\}, \mathcal{F}_2\rangle, \langle \mathcal{C}, \{c(\ldots), c'(\ldots)\}, \mathcal{F}_\Delta\rangle\}) =$$
$$\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots), c'(\ldots)\}, \mathcal{F}_3\rangle\}),$$

$\mathcal{F}_3 = \mathcal{F}_2 \odot \mathcal{F}_\Delta$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)]) \times range(c) \times range(c')$ to $\{-1,0,1\}$ that, given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\ldots,A,\ldots)) : \mathcal{C} \cup \mathcal{C}_A$, is defined as follows:

$$\mathcal{F}_3(h(), \mathbf{v}(c(\ldots)), \mathbf{v}(c'(\ldots))) =$$
$$= \begin{cases} 1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \preccurlyeq \mathbf{v}(c'(\ldots)) \wedge \mathbf{v}(c(\ldots)) = \mathbf{v}(c'(\ldots)); \\ -1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \preccurlyeq \mathbf{v}(c'(\ldots)) \wedge \mathbf{v}(c(\ldots)) = \mathbf{s}(\mathbf{v}(c'(\ldots))); \\ 0, & \text{otherwise;} \end{cases} =$$

(in the first case $\mathbf{v}(c(\ldots)) = \mathbf{v}(c'(\ldots))$ and we replace $\mathbf{v}(c'(\ldots))$ by $\mathbf{v}(c(\ldots))$ in the inequality)

$$= \begin{cases} 1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \preccurlyeq \mathbf{v}(c(\ldots)) \wedge \mathbf{v}(c(\ldots)) = \mathbf{v}(c'(\ldots)); \\ -1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \preccurlyeq \mathbf{v}(c'(\ldots)) \wedge \mathbf{v}(c(\ldots)) = \mathbf{s}(\mathbf{v}(c'(\ldots))); \\ 0, & \text{otherwise;} \end{cases} =$$

(in the second case $\mathbf{v}(c(\ldots)) = \mathbf{s}(\mathbf{v}(c'(\ldots)))$ and we replace $\mathbf{v}(c'(\ldots))$ by $\mathbf{v}(c(\ldots))$ and $\preccurlyeq$ by $\prec$ in the inequality)

$$= \begin{cases} 1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \preccurlyeq \mathbf{v}(c(\ldots)) \wedge \mathbf{v}(c(\ldots)) = \mathbf{v}(c'(\ldots)); \\ -1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \prec \mathbf{v}(c(\ldots)) \wedge \mathbf{v}(c(\ldots)) = \mathbf{s}(\mathbf{v}(c'(\ldots))); \\ 0, & \text{otherwise;} \end{cases} =$$

(we split the first case into two cases)

$$= \begin{cases} 1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x = \mathbf{v}(c(\dots)) \wedge \mathbf{v}(c(\dots)) = \mathbf{v}(c'(\dots)); \\ 1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \prec \mathbf{v}(c(\dots)) \wedge \mathbf{v}(c(\dots)) = \mathbf{v}(c'(\dots)); \\ -1, & \text{if } \max\limits_{x \in range(p), h(x)>0} x \prec \mathbf{v}(c(\dots)) \wedge \mathbf{v}(c(\dots)) = \mathbf{s}(\mathbf{v}(c'(\dots))); \\ 0, & \text{otherwise}, \end{cases} \tag{3.11}$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)])$.

Finally, we perform summation. By Proposition 2.1

$$\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots), c'(\dots)\}, \mathcal{F}_3\rangle\}) =$$
$$\mathcal{J}(\{\sum_{ground(c'(\dots))} \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots), c'(\dots)\}, \mathcal{F}_3\rangle\}) =$$
$$\mathcal{J}(\{\left\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots)\}, \sum\nolimits_{c'(\dots)} \mathcal{F}_3 \right\rangle\}).$$

Factor $\sum_{c'(\dots)} \mathcal{F}_3$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)]) \times range(c)$ to the reals. Let us analyze factor $\mathcal{F}_3$ as it is defined via four cases in expression (3.11). We fix value of $\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)]$ to $h()$ and value of $c(\dots)$ to y.

Assume that $h()$ and y satisfy equality in the first case from (3.11). There is only one value in $range(c')$ equal to y and $\mathcal{F}_3$ has value 1, for all other values of $range(c')$, $\mathcal{F}_3$ has value 0.

Next, assume that $h()$ and y satisfy inequality in the second and third case from (3.11). Note that, since there exists value $x \in range(p) \subset range(c')$ such that $x \prec y$, then y is a successor of exactly one element from $range(c')$. When $c'(\dots)$ is equal to this element, $\mathcal{F}_3$ has value -1. When $c'(\dots)$ is equal to y, $\mathcal{F}_3$ has value 1. For all other values from $range(c')$, $\mathcal{F}_3$ has value 0.

Finally, for all other $h()$ and y, regardless of the value of $c'(\dots)$, $\mathcal{F}_3$ returns value 0.

Based on the above analysis, given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\ldots,A,\ldots)) : \mathcal{C} \cup \mathcal{C}_A$, we have

$$\left(\sum_{c'(\ldots)} \mathcal{F}_3\right)(h(), \mathbf{v}(c(\ldots))) = \begin{cases} 1, & \text{if } \max_{x \in range(p), h(x)>0} x = \mathbf{v}(c(\ldots)); \\ 0, & \text{otherwise,} \end{cases}$$
$$= \mathcal{F}_{\#}(h(), \mathbf{v}(c(\ldots))),$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)])$.

The above gives us

$$\mathcal{J}(\{\left\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots)\}, \sum_{c'(\ldots)} \mathcal{F}_3 \right\rangle\}) =$$
$$\mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots)\}, \mathcal{F}_{\#}\rangle\})$$

and finishes a proof of Equation 3.10 and a proof of the proposition. □

Example 3.10 illustrates the decomposition introduced in the Proposition 3.2.

Example 3.10. Let us consider the aggregation parfactor introduced in Example 3.8:

$$\langle \emptyset, matched\_6(Person), jackpot\_won(), \mathbf{1}, \text{OR}, \emptyset \rangle.$$

The conversion described in Proposition 3.2 introduces an auxiliary parameterized random variable $jackpot\_won'()$, where $jackpot\_won'$ has range $\{false, true\}$. The aggregation parfactor is replaced with two parfactors:

$$\langle \emptyset, \{matched\_6(Person), jackpot\_won'()\}, \mathcal{F}_{jackpot\_won} \rangle \quad \text{and}$$
$$\langle \emptyset, \{jackpot\_won(), jackpot\_won'()\}, \mathcal{F}_{\Delta} \rangle .$$

$\mathcal{F}_{jackpot\_won}$ is a factor from the Cartesian product $\{false, true\} \times \{false, true\}$ to real numbers:

$matched\_6(Person)$	$jackpot\_won'()$	value
$false$	$false$	$\mathbf{1}(false)$
$false$	$true$	$\mathbf{1}(false)$
$true$	$false$	0
$true$	$true$	$\mathbf{1}(true)$

.

$\mathcal{F}_\Delta$ is a factor from the Cartesian product $\{false, true\} \times \{false, true\}$ to a set $\{-1, 0, 1\}$:

$jackpot\_won()$	$jackpot\_won'()$	value
$false$	$false$	1
$false$	$true$	0
$true$	$false$	-1
$true$	$true$	1

.

Note that the size of factors $\mathcal{F}_{jackpot\_won}$ and $\mathcal{F}_\Delta$ is independent of $|\mathcal{D}(Person)|$.

An analogous proposition holds for the MIN operator. In both cases, as illustrated by Examples 3.9 and 3.10 and experiments in Section 3.5, the above conversion is advantageous to the conversion described in Section 3.4.1.1, which uses counting formulas.

3.4.2 Operations on aggregation parfactors

In the previous section we showed how aggregation parfactors can be used during a modeling phase and then, during inference with the C-FOVE algorithm, once populations are known, aggregation parfactors can be converted to parfactors. Such a solution allows us to take advantage of the modeling properties of aggregation parfactors and C-FOVE inference capabilities. It is also possible to exploit aggregation parfactors during inference. In this section we describe operations on aggregation parfactors that can be added to the C-FOVE algorithm. These operations can delay or even avoid conversion of aggregation parfactors to parfactors involving counting formulas. This in turn, as we will see in Section 3.5, can result in more efficient inference.

3.4.2.1 Splitting

The C-FOVE algorithm applies substitutions to parfactors to handle observations and queries and to enable the multiplication of parfactors. As this operation results in the creation of a residual parfactor, it is called splitting. Below we present how aggregation parfactors can be split on substitutions. We start with splitting on a substitution that does not involve the aggregation logical variable:

Proposition 3.3. Let $g_A = \langle \mathcal{C}, p(\ldots, A, \ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ be an aggregation parfactor from Φ. Let $\{X/t\}$ be a substitution such that $(X \neq t) \notin \mathcal{C}$ and $X \in$

param$(c(\dots))$. Let term t be a constant from $\mathcal{D}(X)$, or a logical variable such that $t \in$ *param*$(c(\dots))$. Let $g_A[X/t]$ be a parfactor g_A with all occurrences of X replaced by term t.
Then

$$\mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_A\} \cup \{g_A[X/t], \langle \mathcal{C} \cup \{X \neq t\}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle\}).$$

Proof. It suffices to show that a set of factors represented by the aggregation parfactor g_A is equal to the union of sets of factors represented by the aggregation parfactors $g_A[X/t]$ and $\langle \mathcal{C} \cup \{X \neq t\}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$.

From Definition 3.2 we know that g_A represents a set of factors, one for each ground substitution $\mathcal{G}$ to all logical variables in *param*$(c(\dots))$ that satisfies the constraints in $\mathcal{C}$.

Assume that the term t is a constant. Each ground substitution $\mathcal{G}$ either substitutes X with t or substitutes X with some other constant from $\mathcal{D}(X)$. The former substitutions result in a set of factors equal to the set of factors represented by $g_A[X/t]$ while the latter substitutions result in a set of factors equal to the set of factors represented by $\langle \mathcal{C} \cup \{X \neq t\}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$.

Assume that the term t is a logical variable. Each ground substitution $\mathcal{G}$ either substitutes X and t with the same constant or substitutes X and t with different constants. The former substitutions result in a set of factors equal to the set of factors represented by $g_A[X/t]$ while the latter substitutions result in a set of factors equal to the set of factors represented by $\langle \mathcal{C} \cup \{X \neq t\}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$. □

Proposition 3.3 allows us to split an aggregation parfactor on a substitution that does not involve the aggregation logical variable. Below we show how to split on a substitution that involves the aggregation logical variable A and a constant. After such operation the individuals from $\mathcal{D}(A) : \mathcal{C}$ are represented in two data structures: an aggregation parfactor and a standard parfactor. We have to make sure that after splitting $c(\dots)$ is still equal to a $\otimes$-based aggregation over the whole $\mathcal{D}(A) : \mathcal{C}$. The following proposition describes how it can be done using an auxiliary parameterized random variable:

Proposition 3.4. Let $g_A = \langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ be an aggregation parfactor from Φ. Let $\{A/t\}$ be a substitution such that $(A \neq t) \notin \mathcal{C}_A$ and term t is a constant from $\mathcal{D}(A)$, or a logical variable from $param(p(\ldots,A,\ldots)) \setminus \{A\}$. Let $c'(\ldots)$ be an auxiliary parameterized random variable with the same parameterization and range as $c(\ldots)$. Let $\mathcal{C}_A[A/t]$ be a set of constraints $\mathcal{C}_A$ with all occurrences of A replaced by term t. Let $\mathcal{F}_c$ be a factor from the Cartesian product $range(p) \times range(c') \times range(c)$ to real numbers. Given an assignment of values to random variables $\mathbf{v}$, $\mathcal{F}_c$ is defined as follows:

$$\mathcal{F}_c(\mathbf{v}(p(\ldots,A,\ldots)), \mathbf{v}(c'(\ldots)), \mathbf{v}(c(\ldots))) = \begin{cases} \mathcal{F}_p{}^{\frac{r_c}{r_p}}(p(\ldots,t,\ldots)), & \text{if } \mathbf{v}(c(\ldots)) = \\ & \mathbf{v}(p(\ldots,t,\ldots)) \otimes \mathbf{v}(c'(\ldots)); \\ 0, & \text{otherwise,} \end{cases}$$

where $r_p = |\, ground(p(\ldots,a,\ldots)) : \mathcal{C}\,|$, $a \in \mathcal{D}(A) : \mathcal{C}_A$, and $r_c = |\, ground(c(\ldots)) : \mathcal{C}\,|$. Then

$$\mathcal{J}(\Phi) = \sum_{ground(c'(\ldots))} \mathcal{J}(\Phi \setminus \{g_A\} \cup \{\langle \mathcal{C}, p(\ldots,A,\ldots), c'(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \cup \{A \neq t\}\rangle, \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\ldots,t,\ldots), c'(\ldots), c(\ldots)\}, \mathcal{F}_c\rangle\}).$$

Proof. It suffices to show that

$$\mathcal{J}(\{g_A\}) = \mathcal{J}(\langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle) \tag{3.12}$$

is equal to

$$\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C}, p(\ldots,A,\ldots), c'(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \cup \{A \neq t\}\rangle, \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\ldots,t,\ldots), c'(\ldots), c(\ldots)\}, \mathcal{F}_c\rangle\}). \tag{3.13}$$

We start with expression (3.12) and convert the aggregation parfactor to parfactors. By Proposition 3.1

$$\begin{aligned}&\mathcal{J}(\{\langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A\rangle\}) = \\ &\mathcal{J}(\{\langle \mathcal{C}\cup\mathcal{C}_A, \{p(\ldots,A,\ldots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots)\}, \mathcal{F}_{\#1}\rangle\}).\end{aligned} \tag{3.14}$$

$\mathcal{F}_{\#1}$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)]) \times range(c)$ to a set $\{0,1\}$ that, given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\ldots,A,\ldots)) : \mathcal{C}\cup\mathcal{C}_A$, is defined as follows:

$$\mathcal{F}_{\#1}(h(), \mathbf{v}(c(\ldots))) = \begin{cases} 1, & \text{if } \bigotimes_{x\in range(p)} \bigotimes_{i=1}^{h(x)} x = \mathbf{v}(c(\ldots)); \\ 0, & \text{otherwise,} \end{cases} \tag{3.15}$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)])$.

Next, we transform expression (3.13). We convert the aggregation parfactor to parfactors. By Proposition 3.1

$$\begin{aligned}&\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C}, p(\ldots,A,\ldots), c'(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A\cup\{A\neq t\}\rangle, \\ &\qquad\qquad \langle \mathcal{C}\cup\mathcal{C}_A[A/t], \{p(\ldots,t,\ldots), c'(\ldots), c(\ldots)\}, \mathcal{F}_c\rangle\}) = \\ &\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C}\cup\mathcal{C}_A\cup\{A\neq t\}, \{p(\ldots,A,\ldots)\}, \mathcal{F}_p\rangle, \\ &\qquad\qquad \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A\cup\{A\neq t\}}[p(\ldots,A,\ldots)], c'(\ldots)\}, \mathcal{F}_{\#2}\rangle, \\ &\qquad\qquad \langle \mathcal{C}\cup\mathcal{C}_A[A/t], \{p(\ldots,t,\ldots), c'(\ldots), c(\ldots)\}, \mathcal{F}_c\rangle\}),\end{aligned} \tag{3.16}$$

where $\mathcal{F}_{\#2}$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A\cup\{A\neq t\}}[p(\ldots,A,\ldots)]) \times range(c')$ to a set $\{0,1\}$ that, given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\ldots,A,\ldots)) : \mathcal{C}\cup\mathcal{C}_A\cup\{A\neq t\}$, is defined as follows:

$$\mathcal{F}_{\#2}(h(), \mathbf{v}(c'(\ldots))) = \begin{cases} 1, & \text{if } \bigotimes_{x\in range(p)} \bigotimes_{i=1}^{h(x)} x = \mathbf{v}(c'(\ldots)); \\ 0, & \text{otherwise,} \end{cases}$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\ldots,A,\ldots)])$.

Further, the third parfactor from (3.16) can be represented as a product of two parfactors. By Proposition 2.3

$$\begin{aligned}
&\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A \cup \{A \neq t\}, \{p(\ldots,A,\ldots)\}, \mathcal{F}_p\rangle,\\
&\qquad \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\ldots,A,\ldots)], c'(\ldots)\}, \mathcal{F}_{\#2}\rangle,\\
&\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\ldots,t,\ldots), c'(\ldots), c(\ldots)\}, \mathcal{F}_c\rangle\}) =\\
&\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A \cup \{A \neq t\}, \{p(\ldots,A,\ldots)\}, \mathcal{F}_p\rangle,\\
&\qquad \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\ldots,A,\ldots)], c'(\ldots)\}, \mathcal{F}_{\#2}\rangle,\\
&\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\ldots,t,\ldots)\}, \mathcal{F}_p{}^{\frac{r_c}{r_p}\frac{r_p}{r_c}}\rangle,\\
&\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\ldots,t,\ldots), c'(\ldots), c(\ldots)\}, \mathcal{F}_{c2}\rangle\}) =\\
&\sum_{ground(c'(\ldots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A \cup \{A \neq t\}, \{p(\ldots,A,\ldots)\}, \mathcal{F}_p\rangle,\\
&\qquad \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\ldots,A,\ldots)], c'(\ldots)\}, \mathcal{F}_{\#2}\rangle,\\
&\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\ldots,t,\ldots)\}, \mathcal{F}_p\rangle,\\
&\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\ldots,t,\ldots), c'(\ldots), c(\ldots)\}, \mathcal{F}_{c2}\rangle\}), \qquad (3.17)
\end{aligned}$$

$\mathcal{F}_{c2}$ is a factor from the Cartesian product $range(p) \times range(c') \times range(c)$ to real numbers that, given an assignment of values to random variables $\mathbf{v}$, is defined as follows:

$$\mathcal{F}_{c2}(\mathbf{v}(p(\ldots,A,\ldots)), \mathbf{v}(c'(\ldots)), \mathbf{v}(c(\ldots))) = \begin{cases} 1, & \text{if } \mathbf{v}(c(\ldots)) = \\ & \quad \mathbf{v}(p(\ldots,t,\ldots)) \otimes \mathbf{v}(c'(\ldots)); \\ 0, & \text{otherwise.} \end{cases}$$

The first and the third parfactor from (3.17) can be combined into one parfactor (as if we were reversing a splitting operation). By Proposition 2.4

$$\begin{aligned}
&\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A \cup \{A \neq t\}, \{p(\dots,A,\dots)\}, \mathcal{F}_p\rangle, \\
&\qquad\qquad \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)], c'(\dots)\}, \mathcal{F}_{\#2}\rangle, \\
&\qquad\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\dots,t,\dots)\}, \mathcal{F}_p\rangle, \\
&\qquad\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\dots,t,\dots), c'(\dots), c(\dots)\}, \mathcal{F}_{c2}\rangle\}) = \\
&\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p\rangle, \\
&\qquad\qquad \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)], c'(\dots)\}, \mathcal{F}_{\#2}\rangle, \\
&\qquad\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\dots,t,\dots), c'(\dots), c(\dots)\}, \mathcal{F}_{c2}\rangle\}). \qquad (3.18)
\end{aligned}$$

After replacing expression (3.12) with (3.14) and expression (3.13) with (3.18) we are reduced to proving that

$$\begin{aligned}
&\mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p\rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots)\}, \mathcal{F}_{\#1}\rangle\}) = \\
&\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p\rangle, \\
&\qquad\qquad \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)], c'(\dots)\}, \mathcal{F}_{\#2}\rangle, \\
&\qquad\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\dots,t,\dots), c'(\dots), c(\dots)\}, \mathcal{F}_{c2}\rangle\}).
\end{aligned}$$

The above is equivalent to proving that

$$\begin{aligned}
&\mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots)\}, \mathcal{F}_{\#1}\rangle\}) = \\
&\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)], c'(\dots)\}, \mathcal{F}_{\#2}\rangle, \\
&\qquad\qquad \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\dots,t,\dots), c'(\dots), c(\dots)\}, \mathcal{F}_{c2}\rangle\}). \qquad (3.19)
\end{aligned}$$

We are going to transform the right hand side of the Equation 3.19 into the left hand side. We start with multiplying the two parfactors. By Proposition 2.3

$$\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)], c'(\dots)\}, \mathcal{F}_{\#2}\rangle, \langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{p(\dots,t,\dots), c'(\dots), c(\dots)\}, \mathcal{F}_{c2}\rangle\}) =$$
$$\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)], p(\dots,t,\dots), c'(\dots), c(\dots)\}, \mathcal{F}_{\#3}\rangle\}), \tag{3.20}$$

where $\mathcal{F}_{\#3}$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)]) \times range(p) \times range(c') \times range(c)$ to a set $\{0,1\}$ that, given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\dots,A,\dots)) : \mathcal{C} \cup \mathcal{C}_A \cup \{A \neq t\}$, is defined as follows:

$$\mathcal{F}_{\#3}(h(), \mathbf{v}(p(\dots,t,\dots)), \mathbf{v}(c'(\dots)), \mathbf{v}(c(\dots))) =$$
$$\begin{cases} 1, & \text{if } \bigotimes_{x \in range(p)} \bigotimes_{i=1}^{h(x)} x = \mathbf{v}(c'(\dots)) \wedge \mathbf{v}(c(\dots)) = \mathbf{v}(p(\dots,t,\dots)) \otimes \mathbf{v}(c'(\dots)); \\ 0, & \text{otherwise;} \end{cases} =$$
$$\begin{cases} 1, & \text{if } \bigotimes_{x \in range(p)} \bigotimes_{i=1}^{h(x)} x = \mathbf{v}(c'(\dots)) \wedge \bigotimes_{x \in range(p)} \bigotimes_{i=1}^{h(x)} x \otimes \mathbf{v}(p(\dots,t,\dots)) = \mathbf{v}(c(\dots)), \\ 0, & \text{otherwise,} \end{cases} \tag{3.21}$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)])$.

We continue transformation by performing summation in (3.20). By Proposition 2.1

$$\sum_{ground(c'(\dots))} \mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)], p(\dots,t,\dots), c'(\dots), c(\dots)\}, \mathcal{F}_{\#3}\rangle\}) =$$
$$\mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\dots,A,\dots)], p(\dots,t,\dots), c(\dots)\}, \sum_{c'(\dots)} \mathcal{F}_{\#3}\rangle\}). \tag{3.22}$$

$\sum_{c'(\ldots)} \mathcal{F}_{\#3}$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\ldots,A,\ldots)]) \times range(p) \times range(c)$ to a set $\{0,1\}$. Let us consider factor $\mathcal{F}_{\#3}$ as it is defined in (3.21). If we fix value of $\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)]$ to $h()$, there is only one value y in $range(c')$ such that $\bigotimes_{i=1}^{h(x)} x = y$. Therefore, given an assignment of values $\mathbf{v}$ to all random variables but $ground(p(\ldots,A,\ldots)) : \mathcal{C} \cup \mathcal{C}_A \cup \{A \neq t\}$, $\sum_{c'(\ldots)} \mathcal{F}_{\#3}$ is defined as follows:

$$\sum_{c'(\ldots)} \mathcal{F}_{\#3}(h(), \mathbf{v}(p(\ldots,t,\ldots)), \mathbf{v}(c(\ldots))) = \begin{cases} 1, & \text{if } \bigotimes_{x \in range(p)} \bigotimes_{i=1}^{h(x)} x \otimes \mathbf{v}(p(\ldots,t,\ldots)) = \mathbf{v}(c(\ldots)), \\ 0, & \text{otherwise,} \end{cases} \tag{3.23}$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\ldots,A,\ldots)])$. We will denote $\sum_{c'(\ldots)} \mathcal{F}_{\#3}$ by $\mathcal{F}_{\#4}$.

Let us define factor $\mathcal{F}_{\#5}$ from the Cartesian product $range(\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)]) \times range(c)$ to a set $\{0,1\}$ as follows:

$$\mathcal{F}_{\#5}(h'(), y) = \mathcal{F}_{\#4}(h(), x, y), \tag{3.24}$$

where $x \in range(p)$, $y \in range(c)$, histogram $h() \in range(\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\ldots,A,\ldots)])$, histogram $h'() \in range(\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)])$, and $h'()$ is obtained by taking $h()$ and adding 1 to the count for the value x. From (3.15), (3.23) and (3.24) we have $\mathcal{F}_{\#5} = \mathcal{F}_{\#1}$. Note that Equation 3.24 reassembles Equation 2.8 from Proposition 2.5. Indeed, we can further transform (3.22) by applying Proposition 2.5 as if we were reversing an expansion of a counting formula:

$$\begin{aligned} &\mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A[A/t], \{\#_{A:\mathcal{C}_A \cup \{A \neq t\}}[p(\ldots,A,\ldots)], p(\ldots,t,\ldots), c(\ldots)\}, \mathcal{F}_{\#4}\rangle\}) = \\ &\mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots)\}, \mathcal{F}_{\#5}\rangle\}) = \\ &\mathcal{J}(\{\langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots)\}, \mathcal{F}_{\#1}\rangle\}). \end{aligned}$$

The above and finishes a proof of Equation 3.19 and a proof of the proposition. □

Example 3.11 illustrates Proposition 3.4.

Example 3.11. Consider Example 3.2. As discussed in Example 3.8 the conditional probability distribution $\mathcal{P}(best\_match()|matched(Person))$ can be represented with $\langle \emptyset, matched(Person), best\_match(), \mathbf{1}, \text{MAX}, \emptyset \rangle$.

Assume that we have observed that *sylwia* matched 5 numbers. Before we can project this observation onto our model, we need to split, among others, the above aggregation parfactor on substitution $\{Person/sylwia\}$. We follow the procedure described in Theorem 3.4 and introduce an auxiliary parameterized random variable $best\_match'()$. It is equal to the maximum number of matched lottery numbers among all individuals from the population of the logical variable *Person* except for *sylwia*. An aggregation parfactor $\langle \emptyset, matched(Person), best\_match'(), \mathbf{1}, \text{MAX}, \{Person \neq sylwia\} \rangle$ captures this dependency. Recall that parameterized random variable $best\_match()$ is equal to the maximum number of matched lottery numbers among all individuals from the population of the logical variable *Person*, including *sylwia*. Hence, it is a maximum of $matched(sylwia)$ and $best\_match'()$. This relation is represented by a parfactor $\langle \emptyset, \{matched(sylwia), best\_match'(), best\_match()\}, \mathcal{F}_{best\_match} \rangle$, where $\mathcal{F}_{best\_match()}$ is a factor from the Cartesian product $\{0,1,\ldots,6\} \times \{0,1,\ldots,6\} \times \{0,1,\ldots,6\}$ to real numbers:

$matched(sylwia)$	$best\_match'()$	$best\_match()$	value
0	0	0	1
0	0	1	0
⋮	⋮	⋮	⋮
0	0	6	0
0	1	0	0
0	1	1	1
⋮	⋮	⋮	⋮
0	1	6	0
⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮
6	6	0	0
6	6	1	0
⋮	⋮	⋮	⋮
6	6	6	1

Splitting presented in Proposition 3.4 corresponds to the expansion of a counting formula in C-FOVE. The case where a substitution is of the form $\{X/A\}$ can be handled in a similar fashion as described in Proposition 3.4.

3.4.2.2 Multiplication

The C-FOVE algorithm multiplies parfactors to enable elimination of parameterized random variables. An aggregation parfactor can be multiplied by a parfactor on $p(\dots,A,\dots)$:

Proposition 3.5. Let $g_A = \langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ be an aggregation parfactor from Φ and $g_1 = \langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_1 \rangle$ be a parfactor from Φ. Let $g_2 = \langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p \odot \mathcal{F}_1, \otimes, \mathcal{C}_A \rangle$.
Then

$$\mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_A, g_1\} \cup \{g_2\}).$$

We call g_2 the product of g_A and g_1.

Proof. It suffices to show that

$$\begin{aligned}\mathcal{J}(\{g_A, g_1\}) = \mathcal{J}(\{ &\langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle, \\ &\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_1 \rangle\}) \end{aligned} \tag{3.25}$$

is equal to

$$\mathcal{J}(\{\langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p \odot \mathcal{F}_1, \otimes, \mathcal{C}_A \rangle\}). \tag{3.26}$$

We start with expression (3.25). First we convert the aggregation parfactor g_A to parfactors and then multiply one of the resulting parfactors by g_1.

$$\begin{aligned}&\mathcal{J}(\{ \langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle, \langle \mathcal{C}_1, \{p(\dots,A,\dots)\}, \mathcal{F}_1 \rangle\}) = \\ &\quad \text{(by Proposition 3.1)} \\ &\mathcal{J}(\{ \langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p \rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots)\}, \mathcal{F}_{\#1} \rangle, \\ &\quad \langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_1 \rangle\}) = \\ &\quad \text{(by Proposition 2.3)} \\ &\mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\dots,A,\dots)\}, \mathcal{F}_p \odot \mathcal{F}_1 \rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)], c(\dots)\}, \mathcal{F}_{\#1} \rangle\}).\end{aligned}$$

$\mathcal{F}_{\#1}$ is a factor from the Cartesian product $range(\#_{A:\mathcal{C}_A}[p(\dots,A,\dots)]) \times range(c)$ to a set $\{0,1\}$ that, given an assignment of values $\mathbf{v}$ to all random variables but

$ground(p(\ldots,A,\ldots)) : \mathcal{C} \cup \mathcal{C}_A$, is defined as follows:

$$\mathcal{F}_{\#1}(h(), \mathbf{v}(c(\ldots))) = \begin{cases} 1, & \text{if } \bigotimes_{x \in range(p)} \bigotimes_{i=1}^{h(x)} x = \mathbf{v}(c(\ldots)); \\ 0, & \text{otherwise,} \end{cases} \tag{3.27}$$

where $h()$ is a histogram from $range(\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)])$.

By Proposition 3.1

$$\mathcal{J}(\{\langle \mathcal{C} \cup \mathcal{C}_A, \{p(\ldots,A,\ldots)\}, \mathcal{F}_p \odot \mathcal{F}_1 \rangle, \langle \mathcal{C}, \{\#_{A:\mathcal{C}_A}[p(\ldots,A,\ldots)], c(\ldots)\}, \mathcal{F}_{\#1} \rangle\})$$

is equal to expression (3.26) which finishes a proof of the proposition. □

Below we provide an example of multiplication between an aggregation parfactor and a parfactor.

Example 3.12. Consider the model from Figure 3.3 and Example 3.4. Probability distributions $\mathcal{P}(played(Person))$ and $\mathcal{P}(matched\_6(Person)|played(Person))$ can be represented with a parfactor $\langle \emptyset, \{played(Person)\}, \mathcal{F}_{played} \rangle$ and a parfactor $\langle \emptyset, \{played(Person), matched\_6(Person)\}, \mathcal{F}_{matched\_6} \rangle$, respectively. $\mathcal{F}_{played}$ is a factor from set $\{false, true\}$ to the reals:

$played(Person)$	value
$false$	0.95
$true$	0.05

.

$\mathcal{F}_{matched\_6}$ is a factor from the Cartesian product $\{false, true\} \times \{false, true\}$ to the reals:

$played(Person)$	$matched\_6(Person)$	value
$false$	$false$	1.00000000
$false$	$true$	0.00000000
$true$	$false$	0.99999993
$true$	$true$	0.00000007

.

As shown in Example 3.8, $\mathcal{P}(jackpot\_won()|matched\_6(Person))$ can be represented with $\langle \emptyset, matched\_6(Person), jackpot\_won(), \mathbf{1}, \text{OR}, \emptyset \rangle$.

Let Φ be a set of the three above parfactors. Assume that we want to compute $\mathcal{J}_{ground(jackpot\_won())}(\Phi)$.

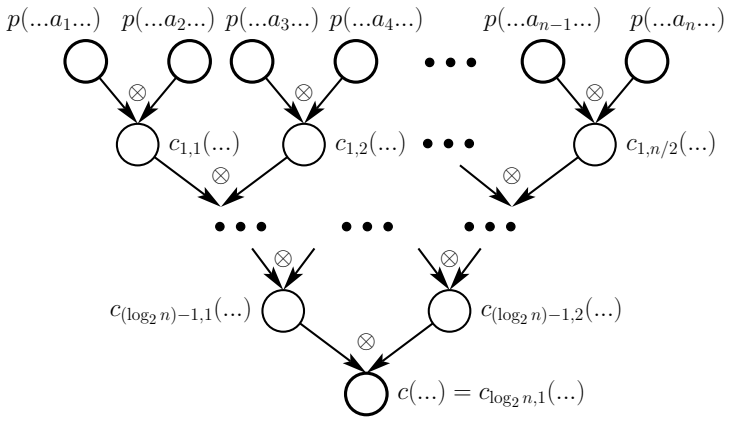


Figure 3.6: Decomposed aggregation.

We multiply the first two parfactors, sum out random variables from the set $ground(played(Person))$ and obtain $\langle \emptyset, \{matched\_6(Player)\}, \mathcal{F}_{sum} \rangle$, where $\mathcal{F}_{sum}$ is a factor from set $\{false, true\}$ to the reals:

$matched\_6(Person)$	value
$false$	0.9999999965
$true$	0.0000000035

.

Now we need to multiply $\langle \emptyset, matched\_6(Person), jackpot\_won(), \mathbf{1}, \text{OR}, \emptyset \rangle$ by $\langle \emptyset, \{matched\_6(Player)\}, \mathcal{F}_{sum} \rangle$. We apply results of Proposition 3.5 and obtain an aggregation parfactor $\langle \emptyset, matched\_6(Person), jackpot\_won(), \mathcal{F}_{sum}, \text{OR}, \emptyset \rangle$.

This simple example involved a trivial factor multiplication $\mathbf{1} \odot \mathcal{F}_{sum} = \mathcal{F}_{sum}$; in general, we might need to multiply two non-trivial factors.

3.4.2.3 Summing out

The C-FOVE algorithm sums out random variables to compute the marginal. Below we show how in some cases we can sum out $p(\ldots, A, \ldots)$ directly from an aggregation parfactor $\langle \emptyset, p(\ldots, A, \ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$.

Consider an aggregation parfactor $\langle \emptyset, p(\ldots, A, \ldots), c(\ldots), \mathcal{F}_p, \otimes, \emptyset \rangle$, assume that $param(c(\ldots)) = param(p(\ldots, A, \ldots)) \setminus \{A\}$. Operator $\otimes$ is associative and we can decompose the aggregation into binary tree of applications of the operator $\otimes$. For simplicity of the discussion, let us first assume that $n = |\mathcal{D}(A) : \mathcal{C}_A|$ is

a power of two. Figure 3.6 illustrates this case. Let $c_{i,j}$ be a functor with range equal to the range of the functor c and $c_{i,j}(\dots)$ be a parameterized random variable such that $\mathit{param}(c_{i,j}(\dots)) = \mathit{param}(p(\dots,A,\dots)) \setminus \{A\}$ for $i = 0,\dots,\log_2 n$; $j = 1,\dots,n/2^i$. Let

$$\begin{aligned}
&c_{0,j}(\dots) = p(\dots,a_j,\dots), && \text{for } j = 1,\dots,n;\\
&c_{i,j}(\dots) = c_{i-1,2j-1}(\dots) \otimes c_{i-1,2j}(\dots), && \text{for } i = 1,\dots,\log_2 n;\ j = 1,\dots,n/2^i;\\
&c(\dots) = c_{\log_2 n,1}(\dots).
\end{aligned}$$

As desired, we obtain:

$$\begin{aligned}
c(\dots) = c_{\log_2 n,1}(\dots) = c_{\log_2 n-1,1}(\dots) \otimes c_{\log_2 n-1,2}(\dots) = \dots =\\
= \bigotimes_{j=1}^{n} p(c_{0,j}(\dots)) = \bigotimes_{j=1}^{n} p(\dots,a_j,\dots).
\end{aligned}$$

When $p(\dots,A,\dots)$ represents a set of random variables that can be treated as independent, the results at each level of the tree shown in Figure 3.6 are identical, therefore we need to compute them only once.

When $n = |\,\mathcal{D}(A) : \mathcal{C}_A\,|$ is an arbitrary natural number, we can use a *square-and-multiply* method [Piṅgala, 200 B.C.], whose time complexity is logarithmic in $|\,\mathcal{D}(A) : \mathcal{C}_A\,|$, to eliminate $p(\dots,A,\dots)$ from an aggregation parfactor. This method is formalized in Proposition 3.6.

Proposition 3.6. Let $g_A = \langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ be an aggregation parfactor from Φ. Assume that that set of constraints $\mathcal{C} \cup \mathcal{C}_A$ is in normal form and that $\mathit{param}(c(\dots)) = \mathit{param}(p(\dots,A,\dots)) \setminus \{A\}$. Assume that no other parfactor or aggregation parfactor in Φ involves parameterized random variables that represent random variables from $\mathit{ground}(p(\dots,A,\dots))$.

Let $m = \lfloor \log_2 |\,\mathcal{D}(A) : \mathcal{C}_A\,| \rfloor$ and $b_m \dots b_0$ be the binary representation of $|\,\mathcal{D}(A) : \mathcal{C}_A\,|$. Let $(\mathcal{F}_0,\dots,\mathcal{F}_m)$ be a sequence of factors from range of c to the reals, defined

recursively as follows:

$$\mathcal{F}_0(x) = \begin{cases} \mathcal{F}_p(x), & \text{if } x \in \mathit{range}(p); \\ 0, & \text{otherwise,} \end{cases}$$

$$\mathcal{F}_k(x) = \begin{cases} \sum\limits_{\substack{y,z \in \mathit{range}(c) \\ y \otimes z = x}} \mathcal{F}_{k-1}(y)\,\mathcal{F}_{k-1}(z), & \text{if } b_{m-k} = 0; \\ \sum\limits_{\substack{w,y,z \in \mathit{range}(c) \\ w \otimes y \otimes z = x}} \mathcal{F}_p(w)\,\mathcal{F}_{k-1}(y)\,\mathcal{F}_{k-1}(z), & \text{if } b_{m-k} = 1. \end{cases}$$

Then

$$\sum_{\mathit{ground}(p(\ldots,A,\ldots))} \mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_a\} \cup \{\langle \mathcal{C}, \{c(\ldots)\}, \mathcal{F}_m \rangle\}).$$

Note that $\mathcal{F}_0, \ldots, \mathcal{F}_m$ are functions stored as factors. Therefore expression $\mathcal{F}_{k-1}(y)\,\mathcal{F}_{k-1}(z)$ requires only one recursive step that computes factor $\mathcal{F}_{k-1}$. Once $\mathcal{F}_{k-1}$ is computed it is applied twice, to y and to z.

In the worst case (when the binary representation of $|\mathcal{D}(A) : \mathcal{C}_A|$ is $11\ldots1$) elimination of $p(\ldots,A,\ldots)$ from an aggregation parfactor $\langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ requires $2(\lfloor \log_2 |\mathcal{D}(A) : \mathcal{C}_A| \rfloor)(\mathit{range}(c))^3$ applications of the $\otimes$ operator, the same number of multiplications, and $(\lfloor \log_2 |\mathcal{D}(A) : \mathcal{C}_A| \rfloor)((\mathit{range}(c))^3 - 1)$ additions. The example below illustrates Proposition 3.6.

Example 3.13. We continue Example 3.12 and apply Proposition 3.6 to eliminate $\mathit{matched\_6}(\mathit{Person})$ from $\langle \emptyset, \mathit{matched\_6}(\mathit{Person}), \mathit{jackpot\_won}(), \mathcal{F}_{\mathit{sum}}, \mathrm{OR}, \emptyset \rangle$, where $\mathcal{F}_{\mathit{sum}}$ is a factor from set $\{\mathit{false}, \mathit{true}\}$ to the reals:

$\mathit{matched\_6}(\mathit{Person})$	value
false	0.9999999965
true	0.0000000035

.

Assume that $n = |\mathcal{D}(\mathit{Person})| = 5$. Thus $m = 2$ and $b_2 = 1, b_1 = 0, b_0 = 1$. Let us compute $(\mathcal{F}_0, \mathcal{F}_1, \mathcal{F}_2)$. We have $\mathcal{F}_0 = \mathcal{F}_{\mathit{sum}}$. Since $b_1 = 0$, $\mathcal{F}_1$ is defined as follows:

$$\mathcal{F}_1(x) = \sum_{\substack{y,z \in \{\mathit{false}, \mathit{true}\} \\ y \,\mathrm{OR}\, z = x}} \mathcal{F}_0(y)\,\mathcal{F}_0(z),$$

where $x \in \{false, true\}$. This gives

$$\begin{aligned}
\mathcal{F}_1(false) &= \mathcal{F}_0(false)\,\mathcal{F}_0(false) \approx 0.999999993 \\
\mathcal{F}_1(true) &= \mathcal{F}_0(false)\,\mathcal{F}_0(true) + \mathcal{F}_0(true)\,\mathcal{F}_0(false) + \mathcal{F}_0(true)\,\mathcal{F}_0(true) \\
&\approx 0.000000007.
\end{aligned}$$

As $b_0 = 0$, $\mathcal{F}_2$ is defined as follows:

$$\mathcal{F}_2(x) = \sum_{\substack{w,y,z \in \{false,true\} \\ w \,\mathrm{OR}\, y \,\mathrm{OR}\, z = x}} \mathcal{F}_1(w)\,\mathcal{F}_1(y)\,\mathcal{F}_1(z),$$

where $x \in \{false, true\}$. This gives

$$\begin{aligned}
\mathcal{F}_2(false) &= \mathcal{F}_0(false)\,\mathcal{F}_1(false)\,\mathcal{F}_1(false) \approx 0.9999999825 \\
\mathcal{F}_2(true) &= \mathcal{F}_0(false)\,\mathcal{F}_1(false)\,\mathcal{F}_1(true) + \mathcal{F}_0(false)\,\mathcal{F}_1(true)\,\mathcal{F}_1(false) \\
&\quad + \mathcal{F}_0(false)\,\mathcal{F}_1(true)\,\mathcal{F}_1(true) + \mathcal{F}_0(true)\,\mathcal{F}_1(false)\,\mathcal{F}_1(false) \\
&\quad + \mathcal{F}_0(true)\,\mathcal{F}_1(false)\,\mathcal{F}_1(true) + \mathcal{F}_0(true)\,\mathcal{F}_1(true)\,\mathcal{F}_1(false) \\
&\quad + \mathcal{F}_0(true)\,\mathcal{F}_1(true)\,\mathcal{F}_1(true) \approx 0.0000000175.
\end{aligned}$$

and $\mathcal{J}_{ground(jackpot\_won())}(\Phi) = \mathcal{J}(\langle \emptyset, \{jackpot\_won()\}, \mathcal{F}_2 \rangle)$.

Operations presented above required 20 applications of operator OR, 20 multiplications and 8 additions. For $n = |\mathcal{D}(Person)| = 19771128$, we would need 264 applications of operator OR, 264 multiplications and 118 additions to compute the result:

$matched\_6(Person)$	value
$false$	≈ 0.933141
$true$	≈ 0.066859

.

As expected, $\mathcal{P}(jackpot\_won() = true)$ increases as $|\mathcal{D}(Person)|$ grows.

Proposition 3.6 does not allow parameterized random variable $c(\dots)$ in an aggregation parfactor to have extra logical variables that are not present in parameterized random variable $p(\dots, A, \dots)$. The C-FOVE algorithm handles extra logical variables by introducing counting formulas on these logical variables. Then it can proceed with standard summation. We cannot apply the same approach to aggrega-

tion parfactors as newly created counting formulas could have ranges incompatible with the range of the aggregation operator. We need a special summation procedure, described below in Proposition 3.7.

Proposition 3.7. Let $g_A = \langle \mathcal{C}, p(\ldots,A,\ldots), c(\ldots,E,\ldots), \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ be an aggregation parfactor from Φ. Assume that set of constraints $\mathcal{C} \cup \mathcal{C}_A$ is in normal form and that $param(c(\ldots)) \setminus \{E\} = param(p(\ldots,A,\ldots)) \setminus \{A\}$. Assume that no other parfactor or aggregation parfactor in Φ involves parameterized random variables that represent random variables from $ground(p(\ldots,A,\ldots))$.

Let $m = \lfloor \log_2 |\, \mathcal{D}(A) : \mathcal{C}_A \,| \rfloor$ and $b_m \ldots b_0$ be the binary representation of $|\, \mathcal{D}(A) : \mathcal{C}_A \,|$. Let $(\mathcal{F}_0, \ldots, \mathcal{F}_m)$ be a sequence of factors from range of c to real numbers, defined recursively as follows:

$$\mathcal{F}_0(x) = \begin{cases} \mathcal{F}_p(x), & \text{if } x \in range(p); \\ 0, & \text{otherwise,} \end{cases}$$

$$\mathcal{F}_k(x) = \begin{cases} \sum\limits_{\substack{y,z \in range(c) \\ y \otimes z = x}} \mathcal{F}_{k-1}(y)\, \mathcal{F}_{k-1}(z), & \text{if } b_{m-k} = 0; \\ \sum\limits_{\substack{w,y,z \in range(c) \\ w \otimes y \otimes z = x}} \mathcal{F}_p(w)\, \mathcal{F}_{k-1}(y)\, \mathcal{F}_{k-1}(z), & \text{otherwise.} \end{cases}$$

Let $\mathcal{C}_E$ be a set of constraints from $\mathcal{C}$ that involve E. Let $\mathcal{F}_\#$ be a factor from the range of counting formula $\#_{E:\mathcal{C}_E}[c(\ldots,E,\ldots)]$ to real numbers defined as follows:

$$\mathcal{F}_\#(h()) = \begin{cases} \mathcal{F}_m(x), & \text{if } \exists x \in range(c)\ h(x) = |\, \mathcal{D}(E) : \mathcal{C}_E \,|; \\ 0, & \text{otherwise,} \end{cases}$$

where $h()$ is a histogram from $range(\#_{E:\mathcal{C}_E}[c(\ldots,E,\ldots)])$.
Then

$$\sum_{ground(p(\ldots,A,\ldots))} \mathcal{J}(\Phi) = \mathcal{J}(\Phi \setminus \{g_a\} \cup \{\langle \mathcal{C} \setminus \mathcal{C}_E, \{\#_{E:\mathcal{C}_E}[c(\ldots,E,\ldots)]\}, \mathcal{F}_\# \rangle\}).$$

If set $\mathcal{C} \cup \mathcal{C}_A$ is not in normal form, then $|\, \mathcal{D}(A) : \mathcal{C}_A \,|$ might vary for different ground substitutions to all logical variables in $p(\ldots,A,\ldots)$ and we will not be

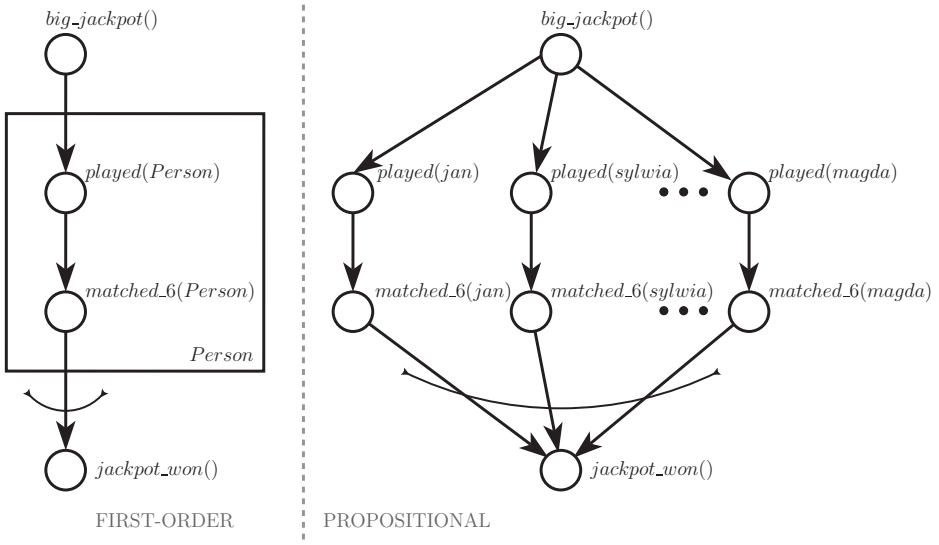


Figure 3.7: A first-order model from Example 3.14 and its equivalent belief network. Aggregation is denoted by curved arcs.

able to apply Propositions 3.6 and 3.7. We can bring constraints in the aggregation parfactor to a normal form by splitting it on appropriate substitutions (see algorithm presented in Figure 5.4 on page 162). Once the constraints are in normal form, $|\mathcal{D}(A):\mathcal{C}_A|$ does not change for different ground substitutions. Another approach is to compute $|\mathcal{D}(A):\mathcal{C}_A|$ conditioned on logical variables in $p(\ldots,A,\ldots)$ with a constraint solver and use this information when summing out $p(\ldots,A,\ldots)$ (see Section 5.2.3 and Section 5.4).

3.4.3 Generalized aggregation parfactors

Propositions 3.6 and 3.7 require that random variables represented by $p(\ldots,A,\ldots)$ are independent. They are dependent if they either have a common ancestor in the grounding or a common observed descendant. If during inference we eliminate the common ancestor or condition on the observed descendant before we eliminate $p(\ldots,A,\ldots)$ through aggregation, we may introduce a counting formula on $p(\ldots,A,\ldots)$. While we can always delay conditioning, with current form of aggregation parfactors sometimes we cannot delay eliminating the common ancestor.

Example 3.14. Consider the directed first-order probabilistic model and its grounding presented in Figure 3.7. It is a modification of the model from Example 3.1, represented with causal independence-based aggregation, as in Example 3.4. The new model has additional parameterized random variable $big\_jackpot()$ with range $\{false, true\}$ which is a parent of parameterized random variable $played(Person)$. Assume that people are more likely to play the lottery when $big\_jackpot()$ is $true$. Ground instances of parameterized random variable $played(Person)$ are no longer independent. Distributions $\mathcal{P}(big\_jackpot())$, $\mathcal{P}(played(Person)|big\_jackpot())$, $\mathcal{P}(matched\_6(Person)|played(Person))$, $\mathcal{P}(jackpot\_won()|matched\_6(Person))$ can be represented with parfactors:

$$\begin{aligned}
\Phi_0 = \{ & \langle \emptyset, \{big_jackpot()\}, \mathcal{F}_{big_jackpot} \rangle, && [1] \\
& \langle \emptyset, \{big_jackpot(), played(Person)\}, \mathcal{F}_{played} \rangle, && [2] \\
& \langle \emptyset, \{played(Person), matched_6(Person)\}, \mathcal{F}_{matched_6} \rangle, && [3] \\
& \langle \emptyset, matched_6(Person), jackpot_won(), \mathbf{1}, \mathrm{OR}, \emptyset \rangle \}, && [4]
\end{aligned}$$

respectively. $\mathcal{F}_{big\_jackpot}$ is a factor from set $\{false, true\}$ to the reals:

$big\_jackpot()$	value
$false$	0.8
$true$	0.2

.

$\mathcal{F}_{played}$ is a factor from the Cartesian product $\{false, true\} \times \{false, true\}$ to the reals:

$big\_jackpot()$	$played(Person)$	value
$false$	$false$	0.95
$false$	$true$	0.05
$true$	$false$	0.85
$true$	$true$	0.15

.

$\mathcal{F}_{matched\_6}$ is a factor from the Cartesian product $\{false, true\} \times \{false, true\}$ to the reals:

$played(Person)$	$matched\_6(Person)$	value
$false$	$false$	1.00000000
$false$	$true$	0.00000000
$true$	$false$	0.99999993
$true$	$true$	0.00000007

.

Let $n = |\mathcal{D}(Person)| = 5$ (as in Example 3.13). Assume that we want to compute $\mathcal{J}_{ground(jackpot\_won())}(\Phi_0)$, which requires eliminating three parameterized random

variables: $big\_jackpot()$, $played(Person)$ and $matched\_6(Person)$. Eliminating $big\_jackpot()$ would introduce counting formula $\#_{Person:\emptyset}[played(Person)]$, which would prevent as from performing aggregation in logarithmic time. We cannot eliminate $matched\_6(Person)$, because it is present in two parfactors, [3] and [4], which cannot be multiplied as they do not satisfy conditions of Proposition 3.5. Our only choice is eliminating $played(Person)$, which involves multiplying parfactors [2] and [3] and summing out $played(Person)$ from the product, all in lifted manner. We obtain an updated set of parfactors:

$$\begin{aligned}\Phi_1 = \{ &\langle \emptyset, \{big\_jackpot()\}, \mathcal{F}_{big\_jackpot} \rangle, && [1]\\ &\langle \emptyset, \{big\_jackpot(), matched\_6(Person)\}, \mathcal{F}_{matched\_6'} \rangle, && [5]\\ &\langle \emptyset, matched\_6(Person), jackpot\_won(), \mathbf{1}, \mathrm{OR}, \emptyset \rangle \}, && [4]\end{aligned}$$

where $\mathcal{F}_{matched\_6'}$ is a factor from the Cartesian product $\{false, true\} \times \{false, true\}$ to the reals:

$big\_jackpot()$	$matched\_6(Person)$	value
$false$	$false$	0.9999999965
$false$	$true$	0.0000000035
$true$	$false$	0.999999989
$true$	$true$	0.000000011

.

We have $\sum_{ground(played(Person))} \mathcal{J}(\Phi_0) = \mathcal{J}(\Phi_1)$.

At this stage, we are left with two parameterized random variables to eliminate: $big\_jackpot()$ and $matched\_6(Person)$. As before, eliminating $big\_jackpot()$ first would introduce a counting formula and we cannot eliminate $matched\_6(Person)$, because it is present in two parfactors, [4] and [5], which cannot be multiplied.

Models like the one described in Example 3.14 do not allow us to apply the results of Propositions 3.6 and 3.7 and perform efficient lifted aggregation. We address this problem by introducing a generalized version of the aggregation parfactor data structure. The generalized version not only can represent aggregation-based dependency between parameterized random variables $p(\ldots, A, \ldots)$ and $c(\ldots)$, but also describes how this aggregation depends on a set of *context parameterized random variables* $\mathcal{V}$. The latter dependency is captured by factor $\mathcal{F}_{p \cup \mathcal{V}}$, a generalized version of factor $\mathcal{F}_p$ from the aggregation parfactor data structure.

Definition 3.3.
A *generalized aggregation parfactor* is a septuple

$$\langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{V}, \mathcal{F}_{p\cup\mathcal{V}}, \otimes, \mathcal{C}_A \rangle,$$

where

- $p(\dots,A,\dots)$ and $c(\dots)$ are parameterized random variables
- the range of p is a subset of the range of c
- A is the only logical variable in $p(\dots,A,\dots)$ that is not in $c(\dots)$
- $\mathcal{V}$ is a set of parameterized random variables, such that: for any two parameterized random variables $f_i(\dots)$, $f_j(\dots)$ from $\mathcal{V}$ we have $(\mathit{ground}(f_i(\dots)) : \mathcal{C}\cup\mathcal{C}_A) \cap (\mathit{ground}(f_j(\mathcal{C})) : \mathcal{C}\cup\mathcal{C}_A) = \emptyset$; and for any $f_i(\dots)$ from $\mathcal{V}$ we have $(\mathit{ground}(f_i(\dots)) : \mathcal{C}\cup\mathcal{C}_A) \cap (\mathit{ground}(p(\dots,A,\dots)) : \mathcal{C}\cup\mathcal{C}_A) = \emptyset$ as well as $(\mathit{ground}(f_i(\dots)) : \mathcal{C}\cup\mathcal{C}_A) \cap \mathit{ground}(c(\dots)) : \mathcal{C} = \emptyset$
- $\mathcal{C}$ is a set of inequality constraints not involving A
- $\mathcal{F}_{p\cup\mathcal{V}}$ is a factor from the Cartesian product of ranges of parameterized random variables in $\{p(\dots,A,\dots)\}\cup\mathcal{V}$ to real numbers.
- $\otimes$ is a commutative and associative deterministic binary operator over the range of c
- $\mathcal{C}_A$ is a set of inequality constraints involving A, such that $(\mathcal{D}(A) : \mathcal{C}_A) \neq \emptyset$.

A generalized aggregation parfactor $\langle \mathcal{C}, p(\dots,A,\dots), c(\dots), \mathcal{V}, \mathcal{F}_p, \otimes, \mathcal{C}_A \rangle$ represents a set of factors, one factor $\mathcal{F}_{p\mathcal{V}c}$ for each ground substitution $\mathcal{G}$ to all logical variables in a set $\bigcup_{f_i(\dots)\in\mathcal{V}} \mathit{param}(f_i(\dots)) \cup \mathit{param}(c(\dots))$ that satisfies the constraints in $\mathcal{C}\cup\mathcal{C}_A$. Each factor $\mathcal{F}_{p\mathcal{V}c}$ is a mapping from the Cartesian product $\times_{a\in\mathcal{D}(A):\mathcal{C}_A} \mathit{range}(p) \times_{f_i(\dots)\in\mathcal{V}} \mathit{range}(f_i) \times \mathit{range}(c)$ to the reals, which, given an assignment of values to all random variables $\mathbf{v}$, is defined as follows:

$$\mathcal{F}_{p\mathcal{V}c}(\mathbf{v}(p(\dots,a_1,\dots)),\dots,\mathbf{v}(p(\dots,a_n,\dots)),\mathbf{v}(f_1(\dots)),\dots,\mathbf{v}(f_m(\dots)),\mathbf{v}(c(\dots))) =$$
$$\begin{cases} \prod\limits_{a\in\{a_1,\dots,a_n\}} \mathcal{F}_{p\cup\mathcal{V}}{}^{\frac{rp}{rc}}(\mathbf{v}(p(\dots,a,\dots)),\mathbf{v}(f_1(\dots)),\dots,\mathbf{v}(f_m(\dots))), \\ \qquad\qquad \text{if} \bigotimes\limits_{a\in\{a_1,\dots,a_n\}} \mathbf{v}(p(\dots,a,\dots)) = \mathbf{v}(c(\dots)); \\ 0, \qquad\qquad \text{otherwise,} \end{cases}$$

where $\mathcal{D}(A):\mathcal{C}_A=\{a_1,\ldots,a_n\}$, $\mathcal{V}=\{f_1(\ldots),\ldots,f_m(\ldots)\}$, $r_c=|\,ground(c(\ldots)):\mathcal{C}\,|$ and $r_p=|\,ground(p(\ldots,a,\ldots)):\mathcal{C}\,|$, $a\in\mathcal{D}(A):\mathcal{C}_A$.

Propositions 3.1–3.7 from Sections 3.4.1 and 3.4.2 can be adapted to generalized parameterized parfactors. The only major changes are that a generalized parfactor can be multiplied by any parfactor on $p(\ldots,A,\ldots)$ and context parameterized random variables and that summing out of $p(\ldots,A,\ldots)$ from a generalized aggregation parfactor involves repeating computation described in Section 3.4.2.3 for each value assignment to context parameterized random variables. We illustrate this with the following example:

Example 3.15. Let us come back to Example 3.14. Assume that we performed all the steps described there, the only difference being that parfactor [4] is a generalized aggregation parfactor:

$$\begin{aligned}\Phi_1=\{\,&\langle\emptyset,\{big\_jackpot()\},\mathcal{F}_{big\_jackpot}\rangle, && [1]\\ &\langle\emptyset,\{big\_jackpot(),matched\_6(Person)\},\mathcal{F}_{matched\_6'}\rangle, && [5]\\ &\langle\emptyset,matched\_6(Person),jackpot\_won(),\emptyset,\mathbf{1},\mathrm{OR},\emptyset\rangle\}. && [4]\end{aligned}$$

We can now multiply parfactors [4] and [5]:

$$\begin{aligned}\Phi_2=\{\,&\langle\emptyset,\{big\_jackpot()\},\mathcal{F}_{big\_jackpot}\rangle, && [1]\\ &\langle\emptyset,matched\_6(Person),jackpot\_won(),\{big\_jackpot()\},\\ &\qquad\qquad\mathcal{F}_{matched\_6'},\mathrm{OR},\emptyset\rangle\}. && [6]\end{aligned}$$

We have $\mathcal{J}(\Phi_1)=\mathcal{J}(\Phi_2)$.

The above step involved a trivial multiplication $\mathbf{1}\odot\mathcal{F}_{matched\_6'}=\mathcal{F}_{matched\_6'}$; in general, we might need to multiply two non-trivial factors.

Recall that $n=|\,\mathcal{D}(Person)\,|=5$ and $\mathcal{F}_{matched\_6'}$ is a factor from the Cartesian product $\{false,true\}\times\{false,true\}$ to the reals:

$big\_jackpot()$	$matched\_6(Person)$	value
$false$	$false$	0.9999999965
$false$	$true$	0.0000000035
$true$	$false$	0.999999989
$true$	$true$	0.000000011

.

Now we eliminate $matched\_6(Person)$ from a generalized aggregation parfactor [6]. The computation involves two steps, one for the case where context parameterized random variable $big\_jackpot()$ is $false$ and on for the case where it is $true$. The first step manipulates numbers from the first two rows of factor $\mathcal{F}_{matched\_6'}$ and is identical to the computation presented in Example 3.13. The second step manipulates numbers from the last two rows of factor $\mathcal{F}_{matched\_6'}$ and otherwise is identical to the first step. We obtain a parfactor [7]:

$$\Phi_3 = \{ \langle \emptyset, \{big\_jackpot()\}, \mathcal{F}_{big\_jackpot} \rangle, \quad [1]$$
$$\langle \emptyset, \{big\_jackpot(), jackpot\_won()\}, \mathcal{F}_{jackpot\_won} \rangle \}, \quad [7]$$

$\mathcal{F}_{jackpot\_won}$ is a factor from the Cartesian product $\{false, true\} \times \{false, true\}$ to the reals:

$big\_jackpot()$	$jackpot\_won()$	value
$false$	$false$	0.9999999825
$false$	$true$	0.0000000175
$true$	$false$	0.9999999450
$true$	$true$	0.0000000550

.

We have $\sum_{ground(matched\_6(Person))} \mathcal{J}(\Phi_2) = \mathcal{J}(\Phi_3)$.

Finally, we eliminate $big\_jackpot()$, which involves multiplying parfactors [1] and [7] and summing out $big\_jackpot()$ from the product. We obtain an updated set of parfactors:

$$\Phi_4 = \{ \langle \emptyset, \{jackpot\_won()\}, \mathcal{F}_{jackpot\_won'} \rangle \}, \quad [8]$$

where $\mathcal{F}_{jackpot\_won'}$ is a factor from set $\{false, true\}$ to the reals:

$jackpot\_won()$	value
$false$	0.999999975
$true$	0.000000025

.

We have $\mathcal{J}(\Phi_4) = \mathcal{J}_{ground(jackpot\_won())}(\Phi_0)$.

3.5 Experiments

In this section we compare how the performance of different ways of representing aggregation in directed first-order probabilistic models scales as the populations sizes of logical variables grow.

We compared the performance of variable elimination (VE), variable elimination with the noisy-MAX factorization [Díez and Galán, 2003] (VE-FCT), C-FOVE, C-FOVE with the lifted noisy-MAX factorization described in Section 3.4.1.2 (C-FOVE-FCT), and C-FOVE with aggregation parfactors (AC-FOVE). We used Java implementations of the above algorithms on an Intel Core 2 Duo 2.66GHz processor with 1GB of memory made available to the JVM.

3.5.1 Memory usage

In the first experiment we investigated how memory usage changes as the population size of the aggregation logical variable increases. For our tests we used small first-order models introduced through this chapter. While they are definitely toy models by themselves, aggregation components that are present in these models could be parts of much bigger, more realistic models. If some of the tested algorithms cannot perform efficient inference in our small models, they will not be able to perform inference in bigger ones.

We tested all five algorithms on the following test instances:

(a) the model introduced in Example 3.1 (depicted in Figure 3.1), compute the marginal of the parameterized random variable $jackpot\_won()$;

(b) the model introduced in Example 3.2 (Figure 3.2), compute the marginal of the parameterized random variable $best\_match()$;

(c) the model introduced in Example 3.5 (Figure 3.5), compute the marginal of the parameterized random variable $jackpot\_winners()$[1];

(d) the model introduced in Example 3.14 (depicted in Figure 3.7), compute the marginal of the parameterized random variable $jackpot\_won()$.

For all instances we varied the population size n of the logical variable *Person* from 1 to $20,000,000$. We recorded the maximum value of n for which aggregation was possible given 1GB of memory. The results are presented in Figure 3.8.

The space complexity for VE is exponential in n, and the algorithm could not handle models with large population sizes. The space complexity for VE-FCT is linear in n and it performed much better than standard VE. For models (a), (c) and

[1]The VE-FCT and C-FOVE-FCT algorithms could not be tested on this model as aggregation is based on the $\text{SUM}_{|3}$ operator.

Model	VE	VE-FCT	C-FOVE	C-FOVE-FCT	AC-FOVE
(a)	25	$\approx 1.0E4$	$> 2.0E7$	$> 2.0E7$	$> 2.0E7$
(b)	23	$\approx 1.0E4$	$\approx 7.0E3$	$> 2.0E7$	$> 2.0E7$
(c)	24	N/A	$> 2.0E7$	N/A	$> 2.0E7$
(d)	24	$\approx 1.0E4$	$> 2.0E7$	$> 2.0E7$	$> 2.0E7$

Figure 3.8: The maximum size of the population size of the logical variable *Person* for which aggregation was possible.

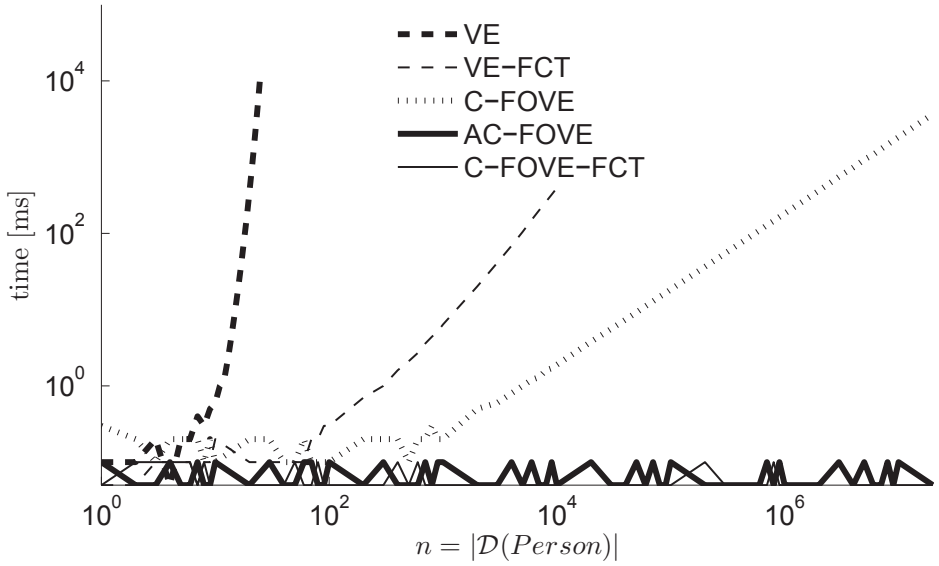


Figure 3.9: Performance on model (a) (with OR-based aggregation).

(d), C-FOVE is also linear in n, but C-FOVE does lifted inference and it achieved better results than VE-FCT, which performs inference at the propositional level. In the model (b), the space complexity for C-FOVE is $\mathcal{O}(n^6)$, and C-FOVE could not handle as large populations as VE-FCT. C-FOVE-FCT and AC-FOVE, for which the space complexity is independent of n, performed best.

Note that for C-FOVE-FCT and AC-FOVE the time complexity is logarithmic in n, for other algorithms the time complexity is the same as the space complexity.

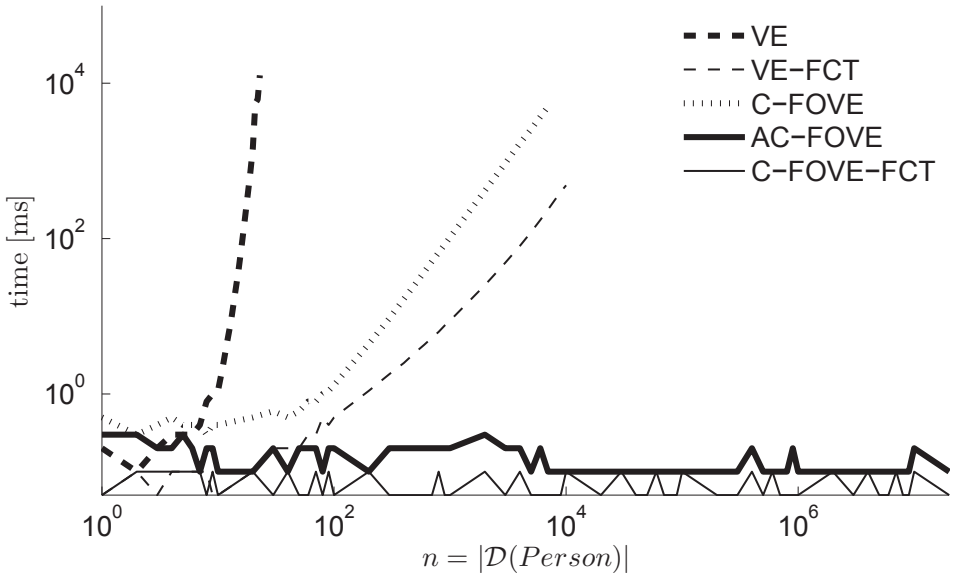


Figure 3.10: Performance on model (b) (with MAX-based aggregation).

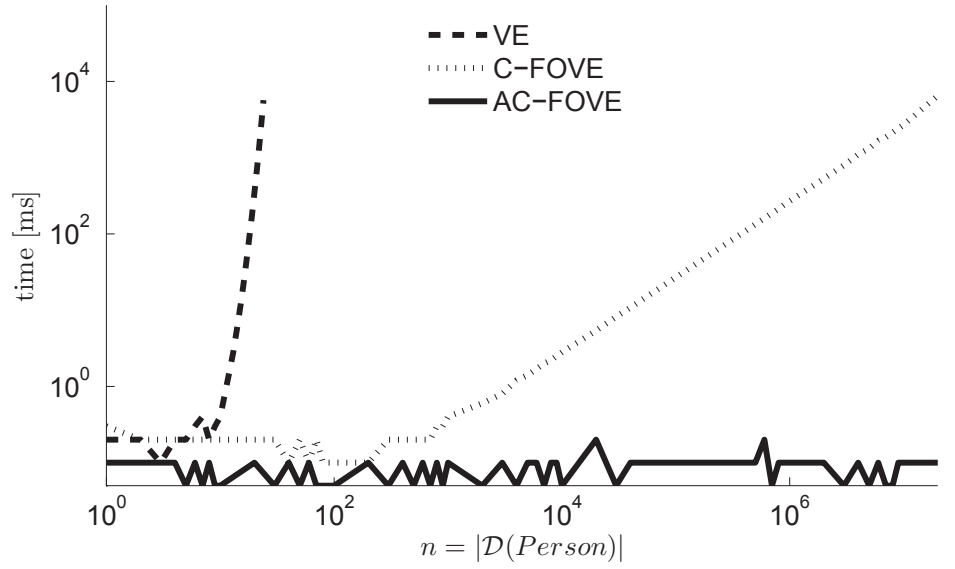


Figure 3.11: Performance on model (c) (with $\text{SUM}_{|3}$-based aggregation).

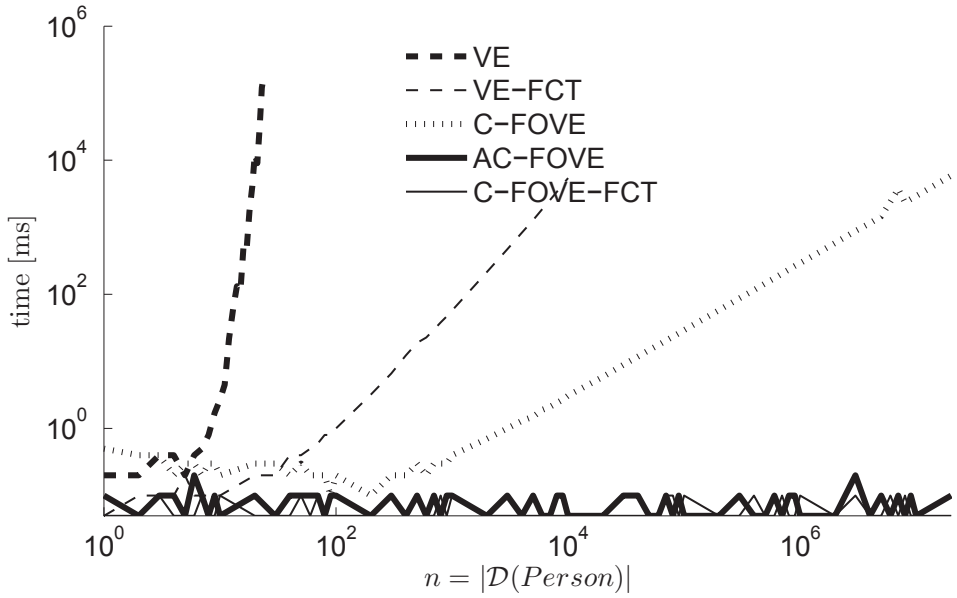


Figure 3.12: Performance on model (d) (generalized aggregation parfactors).

For completeness, on Figures 3.9-3.12 we also present the average time over 10 runs of each algorithm for tested instances. While due to the small sizes of models parts of plots below 1ms are very noisy, we can see that the small memory footprint C-FOVE-FCT and AC-FOVE does not come at a cost of high computation time.

3.5.2 Social network experiment

For this experiment we used an ICL theory [Poole, 2008] from Carbonetto et al. [2009] that explains how people alter their smoking habits within their social network. The theory (without a probability distribution) is shown in Figure 3.13. The population of logical variables X and Y represents the set of people. The theory accounts for various interdependencies between smoking and friendship within this group of people. For example, non-smokers might convince their friend to stop smoking, or two smokers might be more likely to become friends. Possible cyclic dependencies between people are resolved by "switch" parameterized random variable ind(X) with range $\{false, true\}$.

[00] **C** = {{ind(X), ¬ind(X)},
[01] {ns-fr(X,Y), ¬ns-fr(X,Y)},
[02] {diff-sm-fr(X,Y), ¬diff-sm-fr(X,Y)},
[03] {sm-fr(X,Y), ¬sm-fr(X,Y)},
[04] {fr-at-rnd-sm(X,Y), ¬fr-at-rnd-sm(X,Y)},
[05] {fr-at-rnd-nsm(X,Y), ¬fr-at-rnd-nsm(X,Y)},
[06] {fr-at-rnd(X,Y), ¬fr-at-rnd(X,Y)},
[06] {ind-sm(X), ¬ind-sm(X)},
[07] {no-adv-sm(X), ¬no-adv-sm(X)},
[08] {nsm-adv-sm(X), ¬nsm-adv-sm(X)},
[09] {sm-adv-sm(X), ¬sm-adv-sm(X)},
[10] {ctr-adv-sm(X), ¬ctr-adv-sm(X)},
[11] {noise-1(X,Y), ¬noise-1(X,Y)},
[12] {noise-2(X,Y), ¬noise-2(X,Y)}}

[13] **F** = {friends(X,Y) ⟵ X≻Y ∧ friends(Y,X),
[14] friends(X,Y) ⟵ X≺Y ∧ ind(X) ∧ ind(Y) ∧ ¬smokes(X) ∧ ¬smokes(Y) ∧ ns-fr(X,Y),
[15] friends(X,Y) ⟵ X≺Y ∧ ind(X) ∧ ind(Y) ∧ ¬smokes(X) ∧ smokes(Y) ∧ diff-sm-fr(X,Y),
[16] friends(X,Y) ⟵ X≺Y ∧ ind(X) ∧ ind(Y) ∧ smokes(X) ∧ ¬smokes(Y) ∧ diff-sm-fr(X,Y),
[17] friends(X,Y) ⟵ X≺Y ∧ ind(X) ∧ ind(Y) ∧ smokes(X) ∧ smokes(Y) ∧ sm-fr(X,Y),
[18] friends(X,Y) ⟵ X≺Y ∧ ind(X) ∧ ¬ind(Y) ∧ ¬smokes(X) ∧ fr-at-rnd-nsm(X,Y),
[19] friends(X,Y) ⟵ X≺Y ∧ ind(X) ∧ ¬ind(Y) ∧ smokes(X) ∧ fr-at-rnd-sm(X,Y),
[20] friends(X,Y) ⟵ X≺Y ∧ ¬ind(X) ∧ ind(Y) ∧ ¬smokes(Y) ∧ fr-at-rnd-nsm(Y,X),
[21] friends(X,Y) ⟵ X≺Y ∧ ¬ind(X) ∧ ind(Y) ∧ smokes(Y) ∧ fr-at-rnd-sm(Y,X),
[22] friends(X,Y) ⟵ X≺Y ∧ ¬ind(X) ∧ ¬ind(Y) ∧ fr-at-rnd(X,Y),
[23] smokes(X) ⟵ ind(X) ∧ ind-sm(X),
[24] smokes(X) ⟵ ¬ind(X) ∧ ¬sm-adv-sm-fr(X) ∧ ¬nsm-adv-nsm-fr(X) ∧ no-adv-sm(X),
[25] smokes(X) ⟵ ¬ind(X) ∧ ¬sm-adv-sm-fr(X) ∧ nsm-adv-nsm-fr(X) ∧ nsm-adv-sm(X),
[26] smokes(X) ⟵ ¬ind(X) ∧ sm-adv-sm-fr(X) ∧ ¬nsm-adv-nsm-fr(X) ∧ sm-adv-sm(X),
[27] smokes(X) ⟵ ¬ind(X) ∧ sm-adv-sm-fr(X) ∧ nsm-adv-nsm-fr(X) ∧ ctr-adv-sm(X),
[28] sm-adv-sm-fr(X) ⟵ ∃Y friends(X,Y) ∧ ind(Y) ∧ smokes(Y) ∧ noise-1(X,Y),
[29] nsm-adv-nsm-fr(X) ⟵ ∃Y friends(X,Y) ∧ ind(Y) ∧ ¬smokes(Y) ∧ noise-2(X,Y)}

Figure 3.13: ICL theory (without a probability distribution) for the smoking-friendship model.

If ind(X) is $true$, X makes an independent decision to smoke or not to smoke. If ind(X) is $false$, X's decision may be influenced by X's friends. The role of parameterized random variable ind(X) is illustrated in Figure 3.14. Aggregation is present in the theory in lines [28] and [29] which model how a person aggregates advice from smoking and non-smoking friends. For the population size n, the equivalent propositional graphical model has $3n^2+n$ nodes and $12n^2-9n$ arcs. Parameters

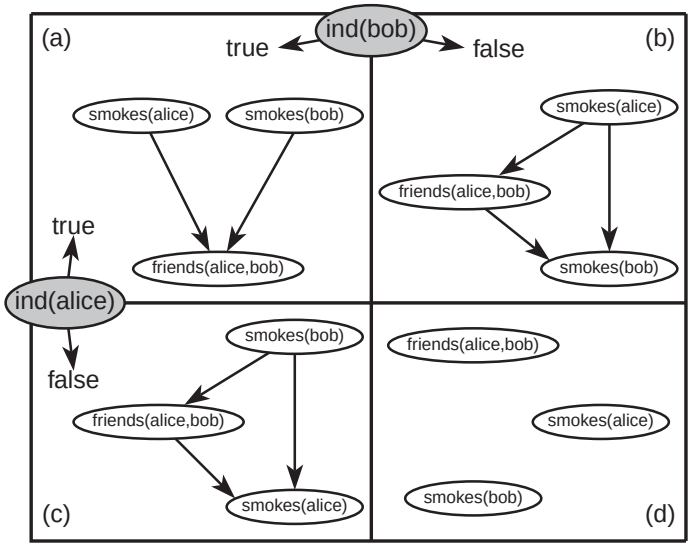


Figure 3.14: Illustration of how ind(X) works.

of the probability distribution forming the theory were learned from data of smoking and drug habits among teenagers attending a school in Scotland [Pearson and Michell, 2000] using methods described by Carbonetto et al. [2009].

In our experiment we varied the populations size n from 2 to 140 and for each value, we computed a marginal probability of a single individual being a smoker. Figure 3.15 shows the average time over 10 runs of tested algorithms for each population size. The VE, VE-FCT and C-FOVE algorithms failed to solve instances with a population size greater than 8, 10, and 11, respectively, because they run out of available memory (1GB). AC-FOVE was able to handle efficiently much larger instances and it ran out of memory for a population size of 159. The AC-FOVE algorithm performed equally to the C-FOVE-FCT algorithm except for small populations. It is important to remember that the C-FOVE-FCT algorithm, unlike AC-FOVE, can only be applied to MAX and MIN-based aggregation.

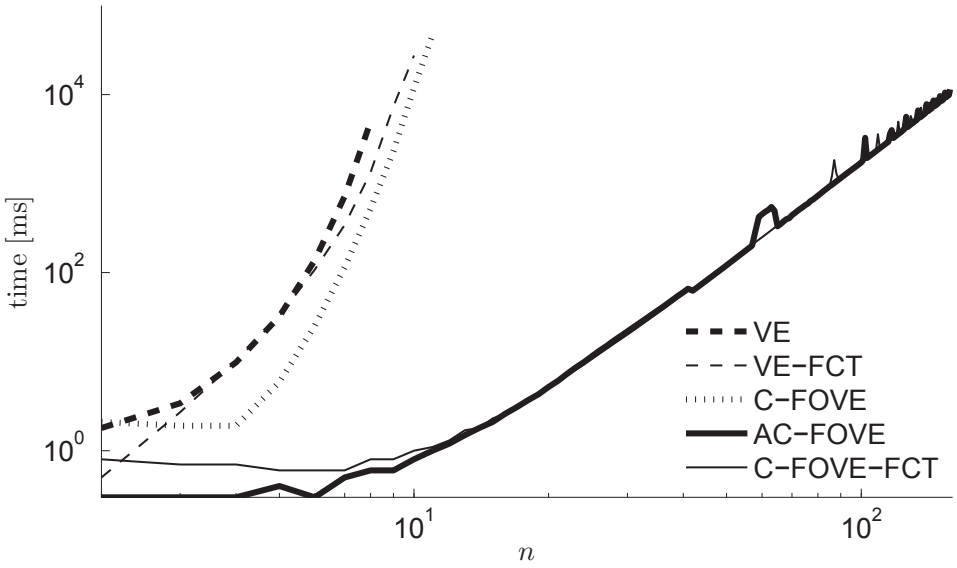


Figure 3.15: Performance on the smoking-friendship model.

3.6 Conclusions

In this chapter we demonstrated the use of aggregation parfactors to represent aggregation in directed first-order probabilistic models, and how aggregation parfactors can be incorporated into the C-FOVE algorithm. Theoretical analysis and empirical tests showed that in some cases, lifted inference with aggregation parfactors leads to significant gains in efficiency.

Chapter 4

Solver for #CSP with Inequality Constraints

When angry count to ten before you speak. If very angry, count to one hundred. — Thomas Jefferson
When angry, count to four; when very angry, swear. — Mark Twain

4.1 Introduction

Lifted probabilistic inference requires counting the number of solutions to binary CSPs (i.e., solving #CSP) with inequality constraints (either between a pair of variables or between a variable and a constant). Variables in these CSP instances typically have large domain sizes. Instances of these problems are described in a lifted manner, that is, the only constants named explicitly are those, for which there exist unary constraints. In order to be efficient, a lifted probabilistic inference engine also requires a lifted answer from a #CSP solver. A lifted answer groups together constants which contribute to the same count.

Existing algorithms for counting the number of solutions to constraint satisfaction problems do not accept lifted descriptions as input or produce lifted descriptions as output. Moreover, the complexity of these algorithms is dominated by the domain size, which also makes them unsuitable for lifted probabilistic inference

where we want to solve the whole problem in time logarithmic in the domain size where possible.

In this chapter we design and analyze a counting algorithm that takes as an input a lifted description of a CSP and returns the answer described in a lifted manner and performs well in presence of large domain sizes.

We provide background information in Section 4.2. In particular, we introduce relevant concepts in Section 4.2.1 and describe a #CSP algorithm from [Dechter, 2003, Section 13.3.3], which is a starting point for our algorithm, in Section 4.2.2.

Our counting algorithm is described in Section 4.3.2 and analyzed theoretically in Section 4.3.4. Empirical tests on random CSP instances are presented in Section 4.3.5. We analyze the impact of the presented algorithm on probabilistic inference in Chapter 5 in Section 5.4. Appendices A to D provide insights into our implementation of the algorithm.

The scope of notation introduced in this chapter is limited to this chapter, Example 5.12 from Chapter 5 and Appendices B to D.

4.2 Background

Constraint Satisfaction Problems (CSPs), first introduced by Montanari [1974], are used for representing problems in many areas. Besides the most widely studied *decision* and *search* variants, one can pose the question "How many solutions exist?" for a particular CSP. This *counting* variant of CSP is known as #CSP. It belongs to the #P class of problems introduced by Valiant [1979], which is the class of all counting problems associated with polynomially balanced, polynomial-time decidable relations [Papadimitriou, 1994]. Binary #CSP was proven to be complete for the #P class [Roth, 1996]. Bulatov and Dalmau [2003] described some tractable subclasses of #CSPs, but these are very restrictive.

Among counting problems, #SAT receives the most attention [Bayardo Jr. and Pehoushek, 2000; Birnbaum and Lozinskii, 1999; Dahllöf et al., 2002, 2005; Dubois, 1991; Zhang, 1996]. However, some approximate and exact algorithms for #CSP have recently been proposed. Meisels et al. [2000] rephrased #CSP in terms of probability updating in Bayesian networks, and applied methods for approximating probabilities in Bayesian networks to approximate the number of solutions.

Angelsmark et al. [2002] represented binary CSP in terms of 2-SAT instances, and used the algorithm described in [Dahllöf et al., 2002] to obtain the number of solutions. In an improved version Angelsmark and Jonsson [2003] translated binary CSP to weighted 2-SAT, and the counting problem was solved with the algorithm described in [Dahllöf et al., 2005]. The translation was done using the *partitioning method*, which works by partitioning the domains of variables in CSP into a number of disjoint subsets. The time complexity of the improved algorithm for n variables and domain size equal to d approaches $\mathcal{O}((0.6224d)^n)$ as d grows. The space complexity is polynomial.

The number of solutions to subproblems of a given CSP can be used as a heuristic for solving the whole problem. Dechter and Pearl [1987] counted solutions with a variant of the variable elimination algorithm. Horsch and Havens [2000] used *solution probabilities*, which can guide search algorithms for solving CSPs. Kask et al. [2004a,b] approximated the number of solutions with the *Iterative Join-Graph Propagation* method [Dechter et al., 2002] and used it as a heuristic to solve CSPs. Refalo [2004] presented a generic search heuristic based on the *impact of a variable*.

Pesant [2005] proposed a structural approach to #CSP. He derived polynomial-time evaluations of the number of solutions of individual constraints of several types, that can be used to approximate the total number of solutions or to guide search heuristics.

Dechter [2003, Section 13.3.3] presented a method for solving #CSP with a variable elimination algorithm. The time and space complexity of the algorithm are equal to $\mathcal{O}(rd^{w^*(\rho)})$, where r is the number of constraints and $w^*(\rho)$ is the induced width of the associated constraint graph as a function of the elimination ordering ρ. In this chapter we show how this variable elimination algorithm can be modified to count the number of solutions to binary CSPs with inequality constraints and large domains.

The class of the problems needed for lifted probabilistic inference, that is the class of CSP with only inequality constraints is known as *list-coloring* problem [Biggs, 1993]. If values from the domains of the variables are interpreted as colors, the problem can be understood as the problem of coloring graph nodes such that two nodes with an edge between them have different colors. Björklund et al.

[2009] provide the algorithm with $2^n n^{\mathcal{O}(1)}$ time and space complexity, where n is the number of variables.

The more restricted class of the problem, where variables have the same domain of size k is known as *k-coloring* problem [Biggs, 1993]. It can be interpreted as a problem of coloring a graph nodes using k colors such that two nodes with an edge between them have different colors. It only applies to lifted probabilistic inference, if the encountered CSP does not involve unary constraints. The partitioning method mentioned earlier has been successfully used to solve this problem [Angelsmark and Thapper, 2006]. The corresponding counting problem is known as the problem of computing *chromatic polynomial* [Biggs, 1993]. The fastest known algorithm due to Björklund et al. [2009] has $2^n n^{\mathcal{O}(1)}$ time and space complexity. Given polynomial space, they can find the smallest k for which the original decision problem has positive answer (the *chromatic number* [Biggs, 1993]) in $\mathcal{O}(2.2461^n)$ time.

4.2.1 Constraint satisfaction problems

The content of this section is based on [Dechter, 2003].

A CSP *instance* is a triple $\mathsf{P} = (\mathsf{X}, \mathsf{D}, \mathsf{C})$, where $\mathsf{X} = \{X_1, \ldots, X_n\}$ is a finite set of *n variables*, D is a function that maps each variable X_i to the set $\mathsf{D}(X_i)$ of possible values it can take (*domain* of X_i) and $\mathsf{C} = \{C_1, \ldots, C_r\}$ is a finite set of *constraints*. Each constraint C_j is a relation over a set $\mathsf{S}(C_j) = \{Y_1, \ldots, Y_m\}$ of variables from X, $C_j \subseteq \mathsf{D}(Y_1) \times \cdots \times \mathsf{D}(Y_m)$. The set $\mathsf{S}(C_j)$ is called the *scope* of C_j and m is called the *arity* of the constraint C_j. In a *binary* CSP instance, all the constraints are unary ($m = 1$) or binary ($m = 2$).

A tuple $\langle x_1, \ldots, x_i \rangle \in \mathsf{D}(X_{l_1}) \times \cdots \times \mathsf{D}(X_{l_i})$ *satisfies* a constraint C_j if $\mathsf{S}(C_j) \subseteq \{X_{l_1}, \ldots, X_{l_i}\}$ and the projection of $\langle x_1, \ldots, x_i \rangle$ on $\mathsf{S}(C_j)$ is an element of C_j.

A tuple $\langle x_1, \ldots, x_n \rangle \in \mathsf{D}(X_1) \times \cdots \times \mathsf{D}(X_n)$ is a *solution of* $\mathsf{P} = (\mathsf{X}, \mathsf{D}, \mathsf{C})$ if it satisfies all constraints in C.

A tuple $\langle x_1, \ldots, x_n \rangle \in \mathsf{D}(X_1) \times \cdots \times \mathsf{D}(X_n)$ is a *consistent extension* in P of the tuple $\langle x_{i_1}, \ldots, x_{i_m} \rangle \in \mathsf{D}(X_{i_1}) \times \cdots \times \mathsf{D}(X_{i_m})$ to the variables $\{X_1, \ldots, X_n\} \setminus \{X_{i_1}, \ldots, X_{i_m}\}$, if it is a solution of P and $\forall j \in \{1, \ldots, m\} \exists l \in \{1, \ldots, n\}$ such that $(x_{i_j} = x_l) \wedge (X_{i_j} = X_l)$.

In this chapter, we restrict our focus to discrete, finite CSP instances, where all variables in P have discrete and finite domains.

The CSP instance $\mathsf{P} = (\mathsf{X}, \mathsf{D}, \mathsf{C})$ can be represented by a *constraint graph*: each variable is represented by a node, and two nodes are connected if they are in the scope of the same constraint from C (in the case of a binary CSP, arcs correspond directly to the constraints).

4.2.2 Variable elimination for #CSP

In this section, we describe the variable elimination algorithm for #CSP (#VE) presented in [Dechter, 2003, Section 13.3.3].

Assume we are given a CSP instance $\mathsf{P} = (\mathsf{X}, \mathsf{D}, \mathsf{C})$. Each constraint C_i with scope $\mathsf{S}(C_i) = \{X_{i_1}, \ldots, X_{i_m}\}$ is represented by a single factor $\mathcal{F}_{C_i}$ on variables X_{i_1}, $\ldots$, X_{i_m}, which has the value 1 for satisfying tuples and 0 otherwise. The number of solutions to P, $|\mathsf{P}|$ is equal to:

$$|\mathsf{P}| = \sum_{X_1} \ldots \sum_{X_n} \mathcal{F}_{C_1}(\mathsf{S}(C_1)) \odot \cdots \odot \mathcal{F}_{C_r}(\mathsf{S}(C_r)).$$

Computing the product $\mathcal{F}_{C_1}(\mathsf{S}(C_1)) \odot \cdots \odot \mathcal{F}_{C_r}(\mathsf{S}(C_r))$ is not tractable, but the #VE algorithm takes advantage of the (possible) sparseness of the associated constraint graph, and using the distribution law, distributes factors that are not functions of X_i outside of the sum $\sum_{X_i}$, for $i = n, \ldots, 1$. Figure 4.1 presents the pseudo-code involved. Note that this is exactly the same algorithms as the VE algorithm for inference in belief networks (see Section 2.3.2 and Figure 2.2 on page 10). The only difference is the way the initial factors are created.

At each step of the computation, the product of all factors that exist at this step represents the number of consistent extensions associated with the previously eliminated variables. Initially, no variables are eliminated and, as described above, we have 0–1 valued factors corresponding to the original constraints. Their product simply enumerates all possible assignments of values to variables and assigns 1 to tuples that are solutions to the input CSP instance and 0 to other tuples.

At the end, if we decide to eliminate all variables (i.e., if $\mathsf{E} = \mathsf{X}$), this algorithm returns a factor on the empty set of variables, which is simply a number equal to the

```
[00]  procedure #CSP_VE(P, E, H)
[01]    input: CSP instance P = (X, D, C),
[02]           set of variables to eliminate E ⊆ X;
[03]           elimination ordering heuristic H;
[04]    output: factor representing the solution for each value of X \ E;
[05]    set F := initialize(P);
[06]    while there is a factor in F involving variable from E do
[07]      select variable Y ∈ E according to H;
[08]      set F := eliminate(Y, F);
[09]      set E := E \ {Y};
[10]    end
[11]    return ⊙_{F_i ∈ F} F_i;
[12]  end

[13]  procedure initialize(P)
[14]    input: CSP instance P = (X, D, C);
[15]    output: representation of P as set of factors F;
[16]    set F := ∅;
[17]    for i := 1 to r do
[18]      create factor F_i on S(C_i) with value 1 for tuples satisfying C_i and
[19]                                  with value 0 otherwise;
[20]      set F := F ∪ {F_i};
[21]    end
[22]    return F;
[23]  end

[24]  procedure eliminate(Y, F)
[25]    input: variable to be eliminated Y,
[26]           set of factors F;
[27]    output: set of factors F with Y summed out;
[28]    partition F = {F_1, ..., F_n} into {F_1, ..., F_m} that do not contain Y and
[29]                                       {F_{m+1}, ..., F_n} that do contain Y;
[30]    return {F_1, ..., F_m, Σ_Y F_{m+1} ⊙ ··· ⊙ F_n};
[31]  end
```

Figure 4.1: #VE algorithm for #CSP from Dechter [2003].

number of solutions to the input CSP instance. If we decide to eliminate only some of the variables ($\mathsf{E} \subset \mathsf{X}$), this algorithm returns a factor representing the number of solutions for each combination of values of variables $\mathsf{X} \setminus \mathsf{E}$.

Assume that we have eliminated all variables ($\mathsf{E} = \mathsf{X}$), and that domains of the variables have the same size $d = |\mathsf{D}(X_1)| = \cdots = |\mathsf{D}(X_n)|$. The time and space

complexity of the algorithm are determined by the size of the biggest factor, which depends on the induced width $w^*(\rho)$ of the associated constraint graph

$$\mathcal{O}(rd^{w^*(\rho)}). \tag{4.1}$$

4.2.3 Set partitions

A *partition of a set S* is a collection $B_1, \ldots, B_k$ of nonempty, pairwise disjoint subsets of S such that $S = \bigcup_{i=1}^{k} B_i$. The sets B_i are called *blocks* of the partition.

Example 4.1. Consider set $S = \{1,2,3\}$. The collection of blocks $B_1 = \{1,3\}$, $B_2 = \{2\}$ is an example of a partition of S. Using a standard notation for set partitions it can be written down as $\{\{1,3\},\{2\}\}$. There are four other partitions of set S: $\{\{1,2,3\}\}$, $\{\{1\},\{2,3\}\}$, $\{\{1,2\},\{3\}\}$, and $\{\{1\},\{2\},\{3\}\}$.

Set partitions are intimately connected to equality. For any consistent set of equality assertions on variables, there exists one or more partitions in which the variables that are equal are in the same block, and the variables that are not equal are in different blocks.

If we consider a semantic mapping from variables to individuals in the world, the inverse of this mapping, where two variables that map to the same individual are in the same block, forms a partition of the variables.

Given a partition π, we denote by $\mathsf{C}(\pi)$ a set of equality assertions corresponding to π.

Example 4.2. Consider set of variables $\{B,C,D\}$. Equality assertions $B \neq C$, $B \neq D$, $C = D$ correspond to a partition $\{\{B\},\{C,D\}\}$. We have $\mathsf{C}(\{\{B\},\{C,D\}\}) = \{B \neq C, B \neq D, C = D\}$.

The number of partitions of the set of size n is equal to the n-th *Bell number* ϖ_n, named after Eric T. Bell, who wrote several papers on the subject [Bell, 1934a,b, 1938]. Bell numbers satisfy the following recurrence:

$$\begin{aligned} \varpi_0 &= 1, \\ \varpi_{n+1} &= \sum_{k=0}^{n} \varpi_k \binom{n}{k}. \end{aligned} \tag{4.2}$$

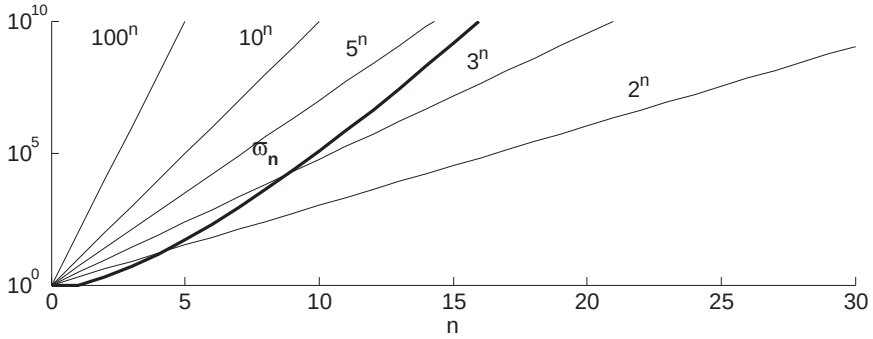


Figure 4.2: Comparison of ϖ_n and exponential functions.

According to Knuth [2005], Equation 4.2 was discovered by Toshiaki Honda in the early 1800s, and William A. Whitworth [Whitworth, 1878] first pointed out the connection between Bell numbers and set partitions. The first few Bell numbers are:

n	=	0	1	2	3	4	5	6	7	8	9	10	...
ϖ_n	=	1	1	2	5	15	52	203	877	4140	21147	115975	...

Bell numbers grow faster than any exponential function (see Lovász [2003]), but for small n's they stay much smaller than exponential functions with a moderate base (see Figure 4.2).

We end this section with a proof of a property of Bell numbers which we will use later on.

Proposition 4.1. Let $i_1, i_2, \ldots, i_k \geq 0$. Then $\varpi_{i_1}\varpi_{i_2}\ldots\varpi_{i_k} \leq \varpi_{i_1+i_2+\cdots+i_k}$.

Proof. The right side of the inequality represents the number of all partitions of the set of size $i_1+i_2+\cdots+i_k$.

Assume we have a set of size $i_1+i_2+\cdots+i_k$, whose elements are painted with k colors and for $1 \leq j \leq k$, there are i_j elements painted with the color j. The left side of the inequality represents the number of all partitions of this set, such that elements with different colors are always in different blocks. It is easy to see, that such a number is smaller or equal to the number of all partitions of the set of size $i_1+i_2+\cdots+i_k$. □

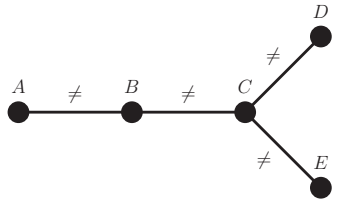


Figure 4.3: Constraint graph with tree structure.

4.3 Counting solutions to CSP instances with inequality constraints

For the rest of this chapter, we restrict our attention to CSP instances with only inequality constraints (either between a pair of variables or between a variable and constants) as only such constraints arise in lifted probabilistic inference. Moreover, for simplicity of the presentation but without loss of generality, we only consider instances for which the constraint graph consists of a single connected component. The number of solutions to CSP with a disconnected constraint graph is simply the product of the numbers of solutions for the connected components of the constraint graph. Note that, unlike the rest of this thesis, in this chapter we do not assume that variables from the same connected component of the constraint graph have the same domain.

Given two variables A and B and an inequality constraint between them, we do not need to consider individual values from their domains in order to calculate the number of solutions to such CSP. If we know the sizes of the variables' domains and the size of the intersection of the domains, we can calculate the number of solutions: $|A||B| - |A \cap B|$. We show below how this simple idea can be generalized to an arbitrary CSP with inequality constraints and develop a modified variable elimination algorithm that does not need to enumerate values from the domains.

4.3.1 Analysis of the problem

Let us start with a very simple constraint graph.

Example 4.3. Consider the constraint graph presented in Figure 4.3, where all variables have the same domain, the domain size is d, and there are no unary con-

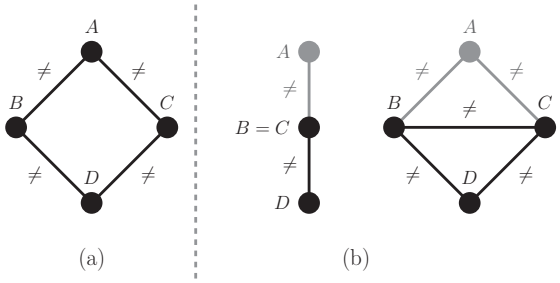


Figure 4.4: Constraint graph with a cycle (a). The two cases: $B = C$ and $B \neq C$ (b).

straints. The graph has a tree structure, which allows us to immediately solve the problem: we can assign the value to A in d ways, and are left with $d-1$ possible values for B, $d-1$ possible values for C and $d-1$ possible values for D and E. Hence, there are $d \cdot (d-1) \cdot (d-1) \cdot (d-1) \cdot (d-1)$ solutions to this CSP instance.

In the next example, we analyze a constraint graph with a cycle.

Example 4.4. Consider the constraint graph presented in Figure 4.4 (a). As in the previous example, all variables have the same domain, the domain size is d, and there are no unary constraints. The graph has a cycle, which makes the calculation more complicated. We can assign the value to A in d ways, and are left with $d-1$ possible values for B and $d-1$ possible values for C. For D we need to consider two cases: $B = C$ and $B \neq C$, as is shown in Figure 4.4 (b). In the $B = C$ case, D can take $d-1$ values, while in the $B \neq C$ case, D can have $d-2$ values. Hence, the number of solutions to this CSP instance is $d \cdot (d-1) \cdot ((d-1) + (d-2) \cdot (d-2))$. The two cases correspond to two partitions of a set $\{B,C\}$: a partition $\{\{B,C\}\}$ and a partition $\{\{B\},\{C\}\}$.

Let us compare our simple reasoning to the computation that would be performed by the #VE algorithm. Considered CSP can be represented with four factors: $f(A,B)$, $f(A,C)$, $f(B,D)$ and $f(C,D)$, each of size d^2. To eliminate A, the #VE algorithm multiplies factors $f(A,B)$ and $f(A,C)$. Next, it sums out A from the factor representing the product and obtains a factor on variables B and C of size d^2. The elimination process continues, but of interest to us is the fact, that

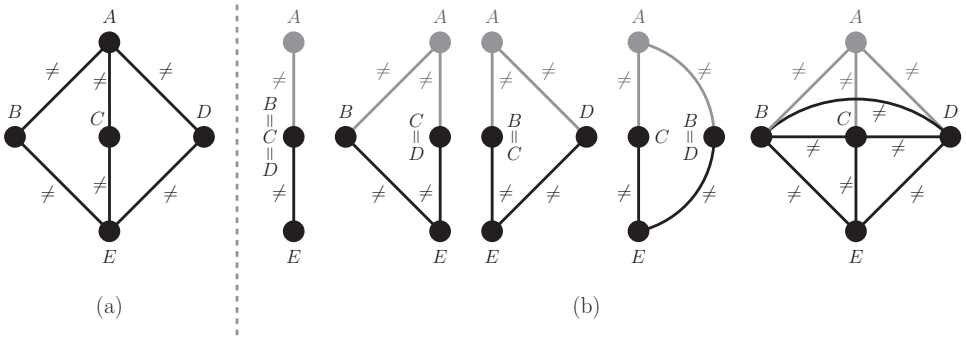


Figure 4.5: Constraint graph discussed in Example 4.5 (a). Cases corresponding to the respective partitions of the set $\{B,C,D\}$ (b).

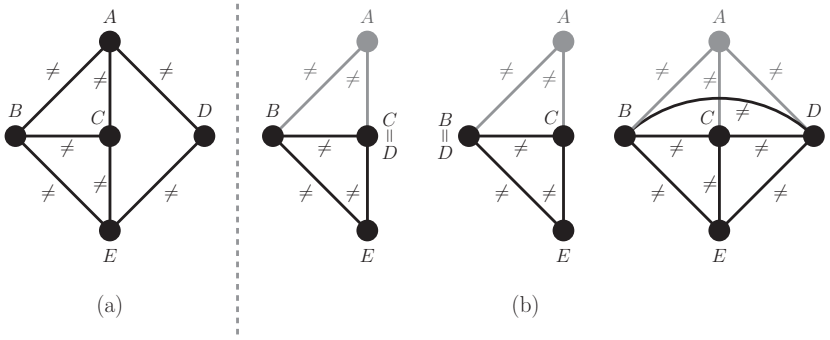


Figure 4.6: Constraint graph discussed in Example 4.5 (a). Cases corresponding to the respective partitions of the set $\{B,C,D\}$ that are consistent with inequality $B \neq C$ (b).

#VE computed d^2 assignments of values to variables B and C, while above we just needed to consider the two partitions: $\{\{B,C\}\}$ and $\{\{B\},\{C\}\}$.

We further examine the above observation in the next example.

Example 4.5. Consider the graph from Figure 4.5 (a). Again, assume that all variables have the same domain, the domain size is d, and there are no unary constraints. If we count possible assignments to variables in a way described in Example 4.4 following order $\rho = \langle A,B,C,D,E\rangle$, we need to consider $\varpi_3 = 5$ partitions of set $\{B,C,D\}$: $\{\{B,C,D\}\}$, $\{\{B\},\{C,D\}\}$, $\{\{B,C\},\{D\}\}$, $\{\{B,D\},\{C\}\}$, $\{\{B\},\{C\},\{D\}\}$. Each partition corresponds to a different case as to whether the

variables are equal or not; these cases are shown on Figure 4.5 (b). The #VE algorithm while following the elimination ordering ρ would create a factor on variables B, C, D of size $d^{w^*(\rho)} = d^3$.

If we add a constraint $B \neq C$ to the constraint graph (see Figure 4.6 (a)), we need to consider only partitions that are consistent with this constraint, that is, partitions in which B and C are in different blocks. There are three such partitions (see Figure 4.6 (b)). The #VE algorithm still creates a factor on B, C, D of size d^3.

Notice that what we have done in Example 4.5 is to consider at most ϖ_3 partitions of variables, rather than d^3 assignments of values to these variables. Since we do not care about empty partitions, we will never have to consider more partitions than there are assignments of values. As we mentioned in Section 4.2.3, for small n's (which in our case is equal to the induced width of a constraint graph $w^*(\rho)$, see Section 4.3.4.2) ϖ_n stays much smaller than exponential functions with a moderate base (in our case equal to the domain size d). In the problems induced during first-order probabilistic inference we consider, we do not expect $w^*(\rho)$ to be very large, but we are likely work with large domains; therefore, considering $\varpi_{w^*(\rho)}$ cases instead of $d^{w^*(\rho)}$ can be a big gain.

In practice, variables can have different domains or different values from their domains might be excluded by unary constraints. In such a situation, we can apply the above reasoning to any set of values that are indistinguishable as far as counting is concerned. For example, the intersection of all domains is a set of values for which we only need the size; there is no point in reasoning about each individual value separately. Similarly, the values from the domain of a variable that do not belong to the domain of any other variable can be grouped and treated together. All we need is to know how many there are.

Example 4.6. Consider again the constraint graph from Figure 4.4. Assume that all variables but B have the same domain $\mathsf{D}(A) = \mathsf{D}(C) = \mathsf{D}(D)$ of size d, and that B has a domain of size $d + d'$, where $|\mathsf{D}(B) \cap \mathsf{D}(A)| = d$. In the case where B has a value from $\mathsf{D}(B) \cap \mathsf{D}(A) = \mathsf{D}(A)$, as before, the number of solutions to this CSP instance is equal to $d \cdot (d-1) \cdot ((d-1) + (d-2) \cdot (d-2))$. In the case where B has a value from $\mathsf{D}(B) \setminus \mathsf{D}(A)$, where $|\mathsf{D}(B) \setminus \mathsf{D}(A)| = d'$, the number of solutions

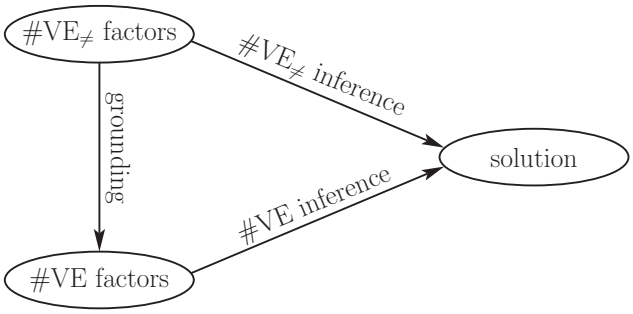


Figure 4.7: Relationship between #VE$_{\neq}$ and #VE.

is equal to $d \cdot d' \cdot (d-1) \cdot (d-1)$. The overall number of solutions is equal to the sum of these two quantities.

Example 4.7. If we were to extend Example 4.6 so that D has the same domain as B, we would then only need to consider how many more solutions would be in the final answer. In this case, we can consider the other values from the domain of D and how these add to the total count of models. We would like to do this locally when summing out variables, rather than globally, as implied in this example.

4.3.2 The #VE$_{\neq}$ algorithm

In this section we describe the #VE$_{\neq}$ algorithm for counting the number of solutions to CSPs with inequality constraints. The core of the #VE$_{\neq}$ algorithm is the same as the #VE algorithm (see Section 4.2.2), but #VE$_{\neq}$ is a lifted algorithm, that is, it reasons at a higher level of abstraction than does #VE. In particular, a #VE$_{\neq}$ factor (Section 4.3.2.2) is more complicated than the corresponding #VE factor, but one tuple in a #VE$_{\neq}$ factor represents many tuples in a #VE factor.

Assume we are given the CSP instance $\mathsf{P} = (\mathsf{X}, \mathsf{D}, \mathsf{C})$, $\mathsf{X} = \{X_1, \ldots, X_n\}$, $\mathsf{C} = \{C_1, \ldots, C_l, C_{l+1}, \ldots, C_r\}$ where $C_1, \ldots, C_l$ are unary constraints and $C_{l+1}, \ldots, C_r$ are binary constraints. We use $\widehat{\mathsf{D}}(X_j)$ to denote the domain $\mathsf{D}(X_j)$ without those values that are excluded by the unary constraints $C_1, \ldots, C_l$.

4.3.2.1 S-constants

Following the analysis presented in Section 4.3.1 we partition domains of variables into disjoint sets of values from these domains. We use *s-constants* to represent such sets. Each s-constant denotes a (non-empty) set of domain values. The sets of domain values associated with different s-constants are assumed to be disjoint. With each s-constant, we have the size of the set it denotes.

Instead of reasoning with individual domain values, we can reason with these s-constants. In the rest of this chapter, we will at times treat an s-constant as a set (as is done in normal mathematics); in this case, we mean the set the s-constant denotes. Note that the algorithm never deals with the sets that the s-constants denote (we don't assume that it has access to these sets).

Example 4.8. In Example 4.7 variables A and C have the same domain of size d, let $\mathsf{D}(A) = \mathsf{D}(C) = \{x_1, \ldots, x_d\}$. Variables B and D have the same domain of size $d + d'$, such that $\mathsf{D}(B) \cap \mathsf{D}(A) = \mathsf{D}(A)$, let $\mathsf{D}(B) = \mathsf{D}(D) = \{x_1, \ldots, x_d, x_{d+1}, \ldots, x_{d+d'}\}$. We need two s-constants to represent values from domains of these four variables: c_1 that represents values in domains of all four variables, $\{x_1, \ldots, x_d\}$, and c_2 that denotes the d' extra values in domains of B and D, $\{x_{d+1}, \ldots, x_{d+d'}\}$. For the purpose of our algorithm we can represent domains of the variables as follows: $\mathsf{D}(A) = \mathsf{D}(C) = c_1$ and $\mathsf{D}(B) = \mathsf{D}(D) = c_1 \cup c_2$. The only additional information we will require is that c_1 represents d values and c_2 represents d' values.

4.3.2.2 #VE$_{\neq}$ factors

A #VE$_{\neq}$ *factor* $\mathcal{F}$ on variables $Y_1, \ldots, Y_m$ is a function from a set of #VE$_{\neq}$ tuples into the natural numbers. A #VE$_{\neq}$ *tuple* $\langle c_1, \ldots, c_k, [\pi_{c_1} \ldots \pi_{c_k}] \rangle$ consists of:

- an s-constant c_i for each variable;
- a set of partitions that contains for each set of variables represented by the same s-constant c_i, a partition π_{c_i} of these variables.

In text we use square brackets to form a set of partitions to avoid confusion with curly brackets of partitions within the set. For the same reason we do not separate partitions with commas. We skip the brackets when displaying #VE$_{\neq}$ factors.

Example 4.9. Let us continue Examples 4.7 and 4.8. Constraints $A \neq B$ and $B \neq D$ can be represented by the following #VE$_{\neq}$ factors:

A	B	Partition(s)	#
c_1	c_1	$\{\{A,B\}\}$	0
c_1	c_1	$\{\{A\},\{B\}\}$	1
c_1	c_2	$\{\{A\}\}$ $\{\{B\}\}$	1

B	D	Partition(s)	#
c_1	c_1	$\{\{B,D\}\}$	0
c_1	c_1	$\{\{B\},\{D\}\}$	1
c_1	c_2	$\{\{B\}\}$ $\{\{D\}\}$	1
c_2	c_1	$\{\{B\}\}$ $\{\{D\}\}$	1
c_2	c_2	$\{\{B,D\}\}$	0
c_2	c_2	$\{\{B\},\{D\}\}$	1

The #VE$_{\neq}$ factor representing the constraint $A \neq B$, $\mathcal{F}_{A\neq B}$, contains three #VE$_{\neq}$ tuples. The first one, $\langle c_1, c_1, [\{\{A,B\}\}]\rangle$, represents the set of tuples from the Cartesian product of subsets of domains of A and B represented by c_1, namely $c_1 \times c_1 = \{x_1,\ldots,x_d\} \times \{x_1,\ldots,x_d\}$, that satisfy the constraint $A = B$, which is implicit in the partition $\{\{A,B\}\}$. Such tuples do not satisfy the constraint $A \neq B$ represented by the factor and are assigned value 0. The second #VE$_{\neq}$ tuple, $\langle c_1, c_1, [\{\{A\},\{B\}\}]\rangle$, represents the set of tuples from $c_1 \times c_1 = \{x_1,\ldots,x_d\} \times \{x_1,\ldots,x_d\}$ that satisfy the constraint represented by the partition $\{\{A\},\{B\}\}$, that is $A \neq B$. These tuples are assigned value 1. Finally, the third #VE$_{\neq}$ tuple, $\langle c_1, c_2, [\{\{A\}\}\ \{\{B\}\}]\rangle$, represents the set of tuples from the Cartesian product of subsets of domains of A and B represented by s-constants c_1 and c_2, $c_1 \times c_2 = \{x_1,\ldots,x_d\} \times \{x_{d+1},\ldots,x_{d+d'}\}$. These subsets are disjoint and each variable is in a partition by itself, meaning that there is no constraint encoded. All tuples from the product satisfy the constraint $A \neq B$ and are assigned value 1.

Suppose s-constant c_i denotes the set S_i. The number associated with the #VE$_{\neq}$ tuple $t = \langle c_{j_1},\ldots,c_{j_m},\Pi\rangle$ in a #VE$_{\neq}$ factor $\mathcal{F}$ will be the same as the number in the corresponding #VE factor on $Y_1,\ldots,Y_m$ associated with each tuple from $S_{j_1} \times \cdots \times S_{j_m}$ that satisfies the equality and inequality constraints implicit in the partitions Π (each of these tuples is associated with the same number). We call the corresponding #VE factor a *grounding* of the factor $\mathcal{F}$, and denote it as $\mathcal{G}(\mathcal{F})$. We denote the tuples from $\mathcal{G}(\mathcal{F})$ corresponding to t by $\mathcal{G}(t)$ and call them *ground tuples*.

Example 4.10. Consider the #VE$_{\neq}$ factor representing constraint $A \neq B$ from Example 4.9. Below we show its grounding:

A	B	#
x_1	x_1	0
$\vdots$	$\vdots$	$\vdots$
x_1	x_d	1
x_1	x_{d+1}	1
$\vdots$	$\vdots$	$\vdots$
x_1	$x_{d+d'}$	1
$\vdots$	$\vdots$	$\vdots$
x_d	x_1	1
$\vdots$	$\vdots$	$\vdots$
x_d	x_d	0
x_d	x_{d+1}	1
$\vdots$	$\vdots$	$\vdots$
x_d	$x_{d+d'}$	1

As with implementation of matrices, we can have either dense representations, for example, using 1-dimensional arrays where we can quickly index any value but need to store zeros, or sparse representations that allow us to avoid storing zeros or repeated structure but are slower when there are few zeros. The examples will show zeros when we think it makes it clearer, but an implementation should do whichever is more efficient. Appendix B shows how to implement the dense representation of factors using a hierarchy of 1-dimensional arrays, so that instead of storing both #VE$_{\neq}$ tuples and values in memory, we only need to store values.

To describe the #VE$_{\neq}$ algorithm, we need to modify multiplication and summing out operators so they can handle a richer representation of #VE$_{\neq}$ factors.

In the example below we show a CSP represented with s-constants and #VE$_{\neq}$ factors. We will use this CSP to illustrate operations on #VE$_{\neq}$ factors in Sections 4.3.2.3 and 4.3.2.4.

$\mathsf{D}(A) = \{x_1, x_2, x_3, x_4, x_5, x_6\}$
$\mathsf{D}(B) = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$
$\mathsf{D}(C) = \{x_1, x_2, x_3, x_4, x_5\}$

$\mathsf{C} = \{A \neq x_1, A \neq x_6, A \neq B, A \neq C$
$B \neq x_2, B \neq x_5, B \neq x_6, B \neq x_7,$
$C \neq x_1\}$

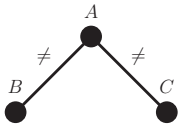


Figure 4.8: A CSP used in Examples 4.11–4.15.

Example 4.11. Consider the CSP presented in Figure 4.8 with $\widehat{\mathsf{D}}(A) = \{x_2, x_3, x_4, x_5\}$, $\widehat{\mathsf{D}}(B) = \{x_1, x_3, x_4, x_8\}$ and $\widehat{\mathsf{D}}(C) = \{x_2, x_3, x_4, x_5\}$. Let $S_0 = \{x_3, x_4\}$, $S_1 = \{x_1, x_8\}$, $S_2 = \{x_2, x_5\}$. Then $\widehat{\mathsf{D}}(A) = S_0 \cup S_2$, $\widehat{\mathsf{D}}(B) = S_0 \cup S_1$, and $\widehat{\mathsf{D}}(B) = S_0 \cup S_2$. Let s-constant c_i denote S_i. When counting the number of solutions, we are not concerned with values; all we need are the counts. We can thus represent this example using two factors, $\mathcal{F}_{A \neq B}$ and $\mathcal{F}_{A \neq C}$, and three numbers:

A	B	Partition(s)	#
$\mathbf{c_0}$	$\mathbf{c_0}$	$\{\{\mathbf{A}, \mathbf{B}\}\}$	**0**
c_0	c_0	$\{\{A\}, \{B\}\}$	1
c_0	c_1	$\{\{A\}\}$ $\{\{B\}\}$	1
c_2	c_0	$\{\{A\}\}$ $\{\{B\}\}$	1
c_2	c_1	$\{\{A\}\}$ $\{\{B\}\}$	1

A	C	Partition(s)	#
$\mathbf{c_0}$	$\mathbf{c_0}$	$\{\{\mathbf{A}, \mathbf{C}\}\}$	**0**
c_0	c_0	$\{\{A\}, \{C\}\}$	1
c_0	c_2	$\{\{A\}\}$ $\{\{C\}\}$	1
c_2	c_0	$\{\{A\}\}$ $\{\{C\}\}$	1
$\mathbf{c_2}$	$\mathbf{c_2}$	$\{\{\mathbf{A}, \mathbf{C}\}\}$	**0**
c_2	c_2	$\{\{A\}, \{C\}\}$	1

$size(c_0) = 2$
$size(c_1) = 2$
$size(c_2) = 2$

#VE$_{\neq}$ tuples in the above #VE$_{\neq}$ factors that are typeset in bold could be pruned as they are assigned value 0.

4.3.2.3 Multiplication

Example 4.12. Consider the factors presented in Example 4.11. The third #VE$_{\neq}$ tuple from the $\mathcal{F}_{A \neq B}$ factor, $\langle c_0, c_1, [\{\{A\}\}\ \{\{B\}\}]\rangle$, represents all tuples from $c_0 \times c_1$. The second #VE$_{\neq}$ tuple from the $\mathcal{F}_{A \neq C}$ factor, $\langle c_0, c_0, [\{\{A\}, \{C\}\}]\rangle$, represents all tuples from $c_0 \times c_0$ such that $A \neq C$. The product of these two #VE$_{\neq}$ tuples represents all tuples from $c_0 \times c_1 \times c_0$ such that $A \neq C$ and can be represented by a #VE$_{\neq}$ tuple $\langle c_0, c_1, c_0, [\{\{A\}\{C\}\}\ \{\{B\}\}]\rangle$.

The second #VE$_{\neq}$ tuple from the $\mathcal{F}_{A \neq B}$ factor, $\langle c_0, c_0, [\{\{A\}, \{B\}\}]\rangle$, represents all tuples from $c_0 \times c_0$ such that $A \neq B$. The product of this #VE$_{\neq}$ tuple with the

second #VE$_{\neq}$ tuple from the $\mathcal{F}_{A\neq C}$ factor represents all tuples from $c_0 \times c_0 \times c_0$ such that $A \neq B$ and $A \neq C$. This is not present in the form of a #VE$_{\neq}$ tuple in a product factor, as constraints $A \neq B$ and $A \neq C$ do not uniquely identify a partition of the set $\{A,B,C\}$. There are two cases: $B = C$ and $B \neq C$. The first case corresponds to the partition $\{\{A\},\{B,C\}\}$, and the second to the partition $\{\{A\},\{B\},\{C\}\}$. The resulting #VE$_{\neq}$ factor has one #VE$_{\neq}$ tuple for each of these two cases: $\langle c_0, c_0, c_0, [\{\{A\},\{B,C\}\}]\rangle$ and $\langle c_0, c_0, c_0, [\{\{A\},\{B\},\{C\}\}]\rangle$, respectively.

We can multiply #VE$_{\neq}$ factors as we do standard factors (see Equation 2.1), treating the s-constants as domain values, except for the case where the same s-constant is used for multiple variables in the product. In this case, we need to create new #VE$_{\neq}$ tuples for each partition of the variables that is consistent with the partitions of the #VE$_{\neq}$ tuples being multiplied (consistency is defined treating a partition as a set of equality and inequality statements). As a special case, if the partitions are inconsistent, no #VE$_{\neq}$ tuples are produced. We can also prune any #VE$_{\neq}$ tuple that has more blocks in the partition for a s-constant than there are values in the set represented by the s-constant. We denote a multiplication operator described above by $\odot^{\neq}$ and define it formally below.

Suppose $\mathcal{F}_1$ is a #VE$_{\neq}$ factor on variables $X_1,\ldots,X_i,Y_1,\ldots,Y_j$, and $\mathcal{F}_2$ is a #VE$_{\neq}$ factor on variables $Y_1,\ldots,Y_j,Z_1,\ldots,Z_l$, where sets $\{X_1,\ldots,X_i\}$, $\{Y_1,\ldots,Y_j\}$ and $\{Z_1,\ldots,Z_l\}$ are pairwise disjoint. The *product* of $\mathcal{F}_1$ and $\mathcal{F}_2$ is a #VE$_{\neq}$ factor $\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2$ on the union of the variables, namely $X_1,\ldots,X_i,Y_1,\ldots,Y_j,Z_1,\ldots,Z_l$, defined by:

$$(\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2)(\langle \overline{c_X}, \overline{c_Y}, \overline{c_Z}, \Pi\rangle) = \mathcal{F}_1(\langle \overline{c_X}, \overline{c_Y}, \Pi_1\rangle)\,\mathcal{F}_2(\langle \overline{c_Y}, \overline{c_Z}, \Pi_2\rangle)\,, \tag{4.3}$$

where

- $\overline{c_X}$, $\overline{c_Y}$, and $\overline{c_Z}$ represent s-constants corresponding to variables $X_1,\ldots,X_i$, $Y_1,\ldots,Y_j$, and $Z_1,\ldots,Z_l$, respectively;
- Π is a set of partitions, one partition per each subset of variables $X_1,\ldots,X_i,Y_1,\ldots,Y_j,Z_1,\ldots,Z_l$ assigned the same s-constant, Π_1 is a set of partitions, one partition per each subset of variables $X_1,\ldots,X_i,Y_1,\ldots,Y_j$ assigned the same

s-constant, and Π_2 is a set of partitions, one partition per each subset of variables $Y_1,\ldots,Y_j,Z_1,\ldots,Z_l$ assigned the same s-constant;

- Π is consistent with Π_1 and Π_2, that is

$$\bigcup_{\pi\in\Pi} \mathsf{C}(\pi) \supseteq \bigcup_{\pi_1\in\Pi_1} \mathsf{C}(\pi_1) \cup \bigcup_{\pi_2\in\Pi_2} \mathsf{C}(\pi_2). \tag{4.4}$$

If we do not prune any #VE$_{\neq}$ tuples, for each #VE$_{\neq}$ tuple from $\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2$, there exists exactly one #VE$_{\neq}$ tuple from $\mathcal{F}_1$ and exactly one #VE$_{\neq}$ tuple from $\mathcal{F}_2$ that satisfies condition (4.4). If we allow pruning, these #VE$_{\neq}$ tuples might not exist and in such case the corresponding #VE$_{\neq}$ tuple in the product $\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2$ is not created.

Example 4.13. Let us multiply the two #VE$_{\neq}$ factors from Example 4.11. Assume that we have pruned 0-valued #VE$_{\neq}$ tuples from the input factors. We have numbered the input #VE$_{\neq}$ tuples and shown where the resulting #VE$_{\neq}$ tuples came from:

	A	B	Partition(s)	#
[1]	c_0	c_0	$\{\{A\},\{B\}\}$	1
[2]	c_0	c_1	$\{\{A\}\}\ \{\{B\}\}$	1
[3]	c_2	c_0	$\{\{A\}\}\ \{\{B\}\}$	1
[4]	c_2	c_1	$\{\{A\}\}\ \{\{B\}\}$	1

	A	C	Partition(s)	#
[5]	c_0	c_0	$\{\{A\},\{C\}\}$	1
[6]	c_0	c_2	$\{\{A\}\}\ \{\{C\}\}$	1
[7]	c_2	c_0	$\{\{A\}\}\ \{\{C\}\}$	1
[8]	c_2	c_2	$\{\{A\},\{C\}\}$	1

$\xrightarrow{\odot^{\neq}}$

	A	B	C	Partition(s)	#
[1·5]	c_0	c_0	c_0	$\{\{A\},\{B,C\}\}$	1
[1·5]	$\mathbf{c_0}$	$\mathbf{c_0}$	$\mathbf{c_0}$	$\{\{\mathbf{A}\},\{\mathbf{B}\},\{\mathbf{C}\}\}$	**1**
[1·6]	c_0	c_0	c_2	$\{\{A\},\{B\}\}\ \{\{C\}\}$	1
[2·5]	c_0	c_1	c_0	$\{\{A\},\{C\}\}\ \{\{B\}\}$	1
[2·6]	c_0	c_1	c_2	$\{\{A\}\}\ \{\{B\}\}\ \{\{C\}\}$	1
[3·7]	c_2	c_0	c_0	$\{\{A\}\}\ \{\{B,C\}\}$	1
[3·7]	c_2	c_0	c_0	$\{\{A\}\}\ \{\{B\},\{C\}\}$	1
[3·8]	c_2	c_0	c_2	$\{\{A\},\{C\}\}\ \{\{B\}\}$	1
[4·7]	c_2	c_1	c_0	$\{\{A\}\}\ \{\{B\}\}\ \{\{C\}\}$	1
[4·8]	c_2	c_1	c_2	$\{\{A\},\{C\}\}\ \{\{B\}\}$	1

Note that a #VE$_{\neq}$ tuple $\langle c_0,c_0,c_2,[\{\{A,B\}\}\ \{\{C\}\}]\rangle$ is not present in the product factor. It is because a #VE$_{\neq}$ tuple $\langle c_0,c_0,[\{\{A,B\}\}]\rangle$ is not present in the $\mathcal{F}_{A\neq B}$ as it has been pruned.

Let us discuss the #VE$_{\neq}$ tuple products from Example 4.12 in context of Equation 4.3.

Consider a #VE$_{\neq}$ tuple $\langle c_0,c_1,c_0,\Pi\rangle$, $\Pi=[\{\{A\}\{C\}\}\ \{\{B\}\}]$ from the product factor. The value assigned to this #VE$_{\neq}$ tuple is the product of the value assigned to a #VE$_{\neq}$ tuple $\langle c_0,c_1,\Pi_1\rangle$ by $\mathcal{F}_{A\neq B}$ and the value assigned to a #VE$_{\neq}$

tuple $\langle c_0, c_0, \Pi_2 \rangle$ by $\mathcal{F}_{A \neq C}$, where $\Pi_1 = [\{\{A\}\ \{B\}\}]$ and $\Pi_2 = [\{\{A\}, \{C\}\}]$. Partitions Π_1 and Π_2 are the only partitions that are consistent with Π for #VE$_{\neq}$ tuples on $A = c_0, B = c_1$ from $\mathcal{F}_{A \neq B}$ and #VE$_{\neq}$ tuples on $A = c_0, C = c_0$ from $\mathcal{F}_{A \neq C}$, respectively.

Consider a #VE$_{\neq}$ tuple $\langle c_0, c_0, c_0, \Pi \rangle$, $\Pi = [\{\{A\}, \{B, C\}\}]$ from the product factor. The value assigned to this #VE$_{\neq}$ tuple is the product of the value assigned to a #VE$_{\neq}$ tuple $\langle c_0, c_0, \Pi_1 \rangle$ by $\mathcal{F}_{A \neq B}$ and the value assigned to a #VE$_{\neq}$ tuple $\langle c_0, c_0, \Pi_2 \rangle$ $\langle c_0, c_0, \Pi_2 \rangle$ by $\mathcal{F}_{A \neq C}$, where $\Pi_1 = [\{\{A\}, \{B\}\}]$ and $\Pi_2 = [\{\{A\}, \{C\}\}]$. Partitions Π_1 and Π_2 are the only partitions that are consistent with Π for #VE$_{\neq}$ tuples on $A = c_0, B = c_0$ from $\mathcal{F}_{A \neq B}$ and #VE$_{\neq}$ tuples on $A = c_0, C = c_0$ from $\mathcal{F}_{A \neq C}$, respectively.

Finally, consider a #VE$_{\neq}$ tuple $\langle c_0, c_0, c_0, \Pi \rangle$, $\Pi = [\{\{A\}, \{B\}, \{C\}\}]$ from the product factor. The value assigned to this #VE$_{\neq}$ tuple is the product of the values assigned to the same #VE$_{\neq}$ tuples as above, namely $\langle c_0, c_0, \Pi_1 \rangle$ and $\langle c_0, c_0, \Pi_2 \rangle$. Partitions Π_1 and Π_2 are the only partitions that are consistent with Π for #VE$_{\neq}$ tuples on $A = c_0, B = c_0$ from $\mathcal{F}_{A \neq B}$ and #VE$_{\neq}$ tuples on $A = c_0, C = c_0$ from $\mathcal{F}_{A \neq C}$, respectively.

Note that we can prune the second #VE$_{\neq}$ tuple from the product factor as $size(c_0) = 2$. There are two values in the set denoted by c_0, and if there are more than two blocks in the partition, it is impossible to assign variables to values. If we do not prune the second #VE$_{\neq}$ tuple, it will eventually get multiplied by zero (in this case when A is summed out, see Example 4.14 and Equation 4.5).

Theorem 4.2. Let f_1 and f_2 be two #VE$_{\neq}$ factors. Then

$$\mathcal{G}(f_1 \odot^{\neq} f_2) = \mathcal{G}(f_1) \odot \mathcal{G}(f_2).$$

Proof. Suppose $\mathcal{F}_1$ is a #VE$_{\neq}$ factor on variables $X_1, \ldots, X_i, Y_1, \ldots, Y_j$, and $\mathcal{F}_2$ is a #VE$_{\neq}$ factor on variables $Y_1, \ldots, Y_j, Z_1, \ldots, Z_l$, where sets $\{X_1, \ldots, X_i\}$, $\{Y_1, \ldots, Y_j\}$ and $\{Z_1, \ldots, Z_l\}$ are pairwise disjoint. #VE$_{\neq}$ tuples in $\mathcal{F}_1$ and $\mathcal{F}_2$ represent exactly ground tuples in $\mathcal{G}(\mathcal{F}_1)$ and $\mathcal{G}(\mathcal{F}_2)$, respectively. While computing the product $\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2$, the #VE$_{\neq}$ algorithm attempts to multiply all of the #VE$_{\neq}$ tuples from $\mathcal{F}_1$ by all of the #VE$_{\neq}$ tuples from $\mathcal{F}_2$.

We start by showing that $\mathcal{G}(\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2) \supseteq \mathcal{G}(\mathcal{F}_1) \odot \mathcal{G}(\mathcal{F}_2)$.

Suppose that $t' \in \mathcal{G}(\mathcal{F}_1) \odot \mathcal{G}(\mathcal{F}_2)$ where the value of t' is v. Then, by the definition of $\odot$, there exist tuples $t'_1 \in \mathcal{G}(\mathcal{F}_1)$ with value v_1 and $t'_2 \in \mathcal{G}(\mathcal{F}_2)$ with value v_2 such that $t' = t'_1 \odot t'_2$ and $v = v_1 \cdot v_2$.

By the definition of $\mathcal{G}()$, there exist #VE$_{\neq}$ tuples $t_1 \in \mathcal{F}_1$ with value v_1 and $t_2 \in \mathcal{F}_2$ with value v_2 such that $t'_1 \in \mathcal{G}(t_1)$ and $t'_2 \in \mathcal{G}(t_2)$. Tuple t'_1 is equal to t'_2 at positions of variables $Y_1, \ldots, Y_j$ (since they were multiplied together by the #VE algorithm) and so t_1 is equal to t_2 at positions of variables $Y_1, \ldots, Y_j$ (since s-constants in #VE$_{\neq}$ tuples denote disjoint sets of domain values). Also, partitions in t_1 are consistent with partitions in t_2; otherwise, partitions in one of the #VE$_{\neq}$ tuples t_1, t_2 would have variables Y_n and Y_m, $1 \leq n,m \leq j$, $n \neq m$ in one block while partitions from the other #VE$_{\neq}$ tuple have them in different blocks. This would imply that in one of the tuples t'_1, t'_2 elements at positions of variables Y_n and Y_m are the same, while in the other there are different elements at these positions, which we know is not the case. Thus t_1 and t_2 are multiplied together by the #VE$_{\neq}$ algorithm, and their product has the value $v_1 v_2 = v$ in $\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2$. It is easy to see that t' belongs to the grounding of the product of t_1 and t_2, hence $t' \in \mathcal{G}(\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2)$.

Now we will show that $\mathcal{G}(\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2) \subseteq \mathcal{G}(\mathcal{F}_1) \odot \mathcal{G}(\mathcal{F}_2)$.

Suppose $t' \in \mathcal{G}(\mathcal{F}_1 \odot^{\neq} \mathcal{F}_2)$, where the value of t' is v. Then by the definition of $\mathcal{G}()$, a #VE$_{\neq}$ tuple $t \in \mathcal{F}_1 \odot^{\neq} \mathcal{F}_2$ exists such that $t' \in \mathcal{G}(t)$. By the definition of $\odot^{\neq}$ there exists a #VE$_{\neq}$ tuple t_1 from $\mathcal{F}_1$ with value v_1 and a #VE$_{\neq}$ tuple t_2 from $\mathcal{F}_2$ with value v_2, such that $t \in t_1 \odot^{\neq} t_2$ (multiplication of two #VE$_{\neq}$ tuples may result in more than one #VE$_{\neq}$ tuple) and $v = v_1 v_2$.

Let t'_1 be the element of $\mathcal{G}(t_1)$ that is identical to t' at positions of variables $X_1, \ldots, X_i$, $Y_1, \ldots, Y_j$. This element must exist because t_1 is identical to t at the positions of variables $X_1, \ldots, X_i$, $Y_1, \ldots, Y_j$, and the partitions of t_1 are the subset of the partitions of t. In an analogous way, we can show that there is a tuple $t'_2 \in \mathcal{G}(t_2)$ that is identical to t' at the positions of variables $Y_1, \ldots, Y_j, Z_1, \ldots, Z_l$.

#VE$_{\neq}$ tuples t_1 and t_2 are equal at positions of variables $Y_1, \ldots, Y_j$, and partitions from t_1 are consistent with partitions from t_2; otherwise, they could not be multiplied. This means that the #VE algorithm would multiply each tuple from $\mathcal{G}(t_1)$ by each tuple from $\mathcal{G}(t_2)$. Thus, tuples t'_1 and t'_2 would be multiplied by the #VE algorithm. It is easy to see that $t'_1 \odot t'_2 = t'$, hence $t' \in \mathcal{G}(\mathcal{F}_1) \odot \mathcal{G}(\mathcal{F}_2)$. $\square$

4.3.2.4 Summing out

To motivate the summation operation in the #VE$_{\neq}$ algorithm, we review the analogous operator in the #VE algorithm: when summing out a variable X from a factor, each tuple contributes its count to the resulting tuple in the new factor. That is, each tuple $\langle X = x, \overline{Y} = \overline{y} \rangle$ with value v (where $\overline{Y}$ is the other variables), adds v to the resulting tuple $\langle \overline{Y} = \overline{y} \rangle$. The complication with #VE$_{\neq}$ is that each #VE$_{\neq}$ tuple represents many different ground tuples.

To sum out a variable X from a #VE$_{\neq}$ factor we go through each #VE$_{\neq}$ tuple in the factor, and determine which part of the resulting #VE$_{\neq}$ factor it contributes to and how much it contributes. Suppose the #VE$_{\neq}$ tuple under consideration is $\langle X = c_X, \overline{Y} = \overline{c_Y}, \Pi \rangle$ and its value is v. We want to determine the effective count of this #VE$_{\neq}$ tuple with respect to X.

The *effective count* of a #VE$_{\neq}$ tuple $\langle X = c_X, \overline{Y} = \overline{y_Y}, \Pi \rangle$ with respect to variable X is the number of ways X can be assigned to values represented by s-constant c_X given assignments of values to variables $\overline{Y}$. Consider a partition $\pi_X \in \Pi$ that contains X. If X is in the same block as another variable, the effective count is 1. If X is in a block by itself, the effective count is $size(c_X)$ minus the number of other blocks in π_X unless the number of other blocks is greater than $size(c_X)$, in which case the effective count is 0.

Example 4.14. Consider a #VE$_{\neq}$ tuple $\langle c_0, c_0, c_0, [\{\{A\}, \{B, C\}\}] \rangle$ from the product #VE$_{\neq}$ factor from Example 4.13. Recall that $size(c_0) = 2$. In the only partition in this #VE$_{\neq}$ tuple, variable A is in a block by itself and there is one more block. Therefore the effective count of this #VE$_{\neq}$ tuple with respect to A is $2 - 1 = 1$. Variable B is in the same block as C. The effective count of this #VE$_{\neq}$ tuple with respect to B is 1.

Consider another #VE$_{\neq}$ tuple, $\langle c_0, c_0, c_0, [\{\{A\}, \{B\}, \{C\}\}] \rangle$, from the same #VE$_{\neq}$ factor. In this #VE$_{\neq}$ tuple, variable A is in a block by itself and there are two more blocks. The effective count of this #VE$_{\neq}$ tuple with respect to A is $2 - 2 = 0$.

The #VE$_{\neq}$ tuple $\langle X = c_X, \overline{Y} = \overline{c_Y}, \Pi \rangle$, then, contributes v times the effective count of X to the #VE$_{\neq}$ tuple $\langle \overline{Y} = \overline{c_Y}, \Pi' \rangle$ from the resulting #VE$_{\neq}$ factor, where Π' is the same as Π but with X removed from the appropriate partition. We assume that Π' is simplified so that empty blocks are removed. We denote a summation operator described above by $\sum^{\neq}$ and define it formally below.

Suppose $\mathcal{F}$ is a #VE$_{\neq}$ factor on variables $X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n$. The *summing out* of variable X_i from $\mathcal{F}$, denoted as $\sum^{\neq}_{X_i} \mathcal{F}$, is the #VE$_{\neq}$ factor on variables $X_1, \ldots, X_{i-1}, X_i, X_{i+1}, \ldots, X_n$ such that:

$$\left(\sum\nolimits^{\neq}_{X_i} \mathcal{F}\right)\left(\langle \overline{c_1}, \overline{c_n}, \Pi' \rangle\right) = \sum_{c_i \in \mathsf{D}(X_i)} \left(\mathcal{F}(\langle \overline{c_1}, c_i, \overline{c_n}, \Pi \rangle) \cdot ec(\langle \overline{c_1}, c_i, \overline{c_n}, \Pi \rangle, X_i)\right), \quad (4.5)$$

where

- $\overline{c_1}$ and $\overline{c_n}$ represent s-constants corresponding to variables $X_1, \ldots, X_{i-1}$ and $X_{i+1}, \ldots, X_n$, respectively;
- Π' is a set of partitions obtained by removing variable X_i from the appropriate partition in Π;
- $ec(\langle \overline{c_1}, c_i, \overline{c_n}, \Pi \rangle, X_i)$ is the effective count of #VE$_{\neq}$ tuple $\langle \overline{c_1}, c_i, \overline{c_n}, \Pi \rangle$ with respect to variable X_i.

Example 4.15. Let us sum out A from the product factor from Example 4.13. We have numbered the input #VE$_{\neq}$ tuples and shown where the resulting #VE$_{\neq}$ tuples came from:

	A	B	C	Partition(s)	#
[1]	c_0	c_0	c_0	$\{\{A\},\{B,C\}\}$	1
[2]	c_0	c_0	c_2	$\{\{A\},\{B\}\}\ \{\{C\}\}$	1
[3]	c_0	c_1	c_0	$\{\{A\},\{C\}\}\ \{\{B\}\}$	1
[4]	c_0	c_1	c_2	$\{\{A\}\}\ \{\{B\}\}\ \{\{C\}\}$	1
[5]	c_2	c_0	c_0	$\{\{A\}\}\ \{\{B,C\}\}$	1
[6]	c_2	c_0	c_0	$\{\{A\}\}\ \{\{B\},\{C\}\}$	1
[7]	c_2	c_0	c_2	$\{\{A\},\{C\}\}\ \{\{B\}\}$	1
[8]	c_2	c_1	c_0	$\{\{A\}\}\ \{\{B\}\}\ \{\{C\}\}$	1
[9]	c_2	c_1	c_2	$\{\{A\},\{C\}\}\ \{\{B\}\}$	1

$\xrightarrow{\sum^{\neq}_{A}}$

	B	C	Partition(s)	#
[1+5]	c_0	c_0	$\{\{B,C\}\}$	1+2=3
[6]	c_0	c_0	$\{\{B\},\{C\}\}$	2
[2+7]	c_0	c_2	$\{\{B\}\}\ \{\{C\}\}$	1+1=2
[3+8]	c_1	c_0	$\{\{B\}\}\ \{\{C\}\}$	1+2=3
[4+9]	c_1	c_2	$\{\{B\}\}\ \{\{C\}\}$	2+1=3

The #VE$_{\neq}$ tuple labeled [1] provides a $(size(c_0) - 1) \cdot 1 = 1$ contribution to the #VE$_{\neq}$ tuple labeled $[1+5]$. The #VE$_{\neq}$ tuple labeled [5] provides $size(c_2) \cdot 1 = 2$

contribution to the same #VE$_{\neq}$ tuple. Thus the #VE$_{\neq}$ tuple labeled $[1+5]$ has a value of 3 in the resulting #VE$_{\neq}$ factor shown above.

Notice that if the size of the sets associated with the s-constants was bigger, there would have been one extra #VE$_{\neq}$ tuple resulting from Example 4.13, namely $\langle c_0, c_0, c_0, [\{\{A\},\{B\},\{C\}\}]\rangle$, but otherwise we would just be multiplying and adding bigger numbers.

Theorem 4.3. Let $\mathcal{F}$ be a #VE$_{\neq}$ factor and X be a variable. Then

$$\mathcal{G}(\sum\nolimits_X^{\neq} \mathcal{F}) = \sum_X \mathcal{G}(\mathcal{F}).$$

Proof. Suppose $\mathcal{F}$ is a #VE$_{\neq}$ factor on variables $Y_1,\ldots,X,\ldots,Y_n$ (if $\mathcal{F}$ is not a factor on X, then the theorem is trivially true).

We start by showing, that $\mathcal{G}(\sum_X^{\neq} \mathcal{F}) \supseteq \sum_X \mathcal{G}(\mathcal{F})$.

Suppose that $t' = \langle y_1,\ldots,y_n\rangle \in \sum_X \mathcal{G}(\mathcal{F})$ and value of t' is v. Then, by the definition of $\sum$ there exist tuples $t'_1 = \langle y_1,\ldots,x_1,\ldots,y_n\rangle, \ldots, t'_m = \langle y_1,\ldots,x_m,\ldots,y_n\rangle \in \mathcal{G}(\mathcal{F})$, with values $v_1,\ldots,v_m$ such that $v = v_1 + \cdots + v_m$. Values $x_1,\ldots,x_m$ belong to sets of domains values denoted by s-constants $c_1,\ldots,c_j$, where $j \leq m$.

By the definition of $\mathcal{G}()$ for each t'_i, $1 \leq i \leq m$ there exists a #VE$_{\neq}$ tuple $t_k \in \mathcal{F}$, $1 \leq k \leq j$ with s-constant c_k at the position of variable X such that $t'_i \in \mathcal{G}(t_k)$. Tuples $t_1,\ldots,t_j$ are identical at all positions except for the position of variable X (because corresponding ground tuples are identical at those positions). Also, all of their partitions not involving variable X are identical; otherwise, partitions in one of the #VE$_{\neq}$ tuples $t_1,\ldots,t_j$ would have variables Y_p and Y_r, $1 \leq p,r \leq n$, $p \neq r$ in one block while partitions from the other #VE$_{\neq}$ tuple would have them in different blocks. This scenario would imply that in one of the tuples $t'_1,\ldots,t'_m$ elements at positions of variables Y_p and Y_r are the same, while in the other there are different elements at these positions, which we know is not the case.

The above conclusion means that the #VE$_{\neq}$ algorithm would add together #VE$_{\neq}$ tuples $t_1,\ldots,t_j$ during summing out of X. Such summation would yield a #VE$_{\neq}$ tuple t with the same s-constants at positions of variables $Y_1,\ldots,Y_n$ as tuples $t_1,\ldots,t_j$, and with overall contribution to the value of t from the s-constants $c_1,\ldots,c_j$ equal to v. It is easy to see that $t' \in \mathcal{G}(t)$, hence $t' \in \mathcal{G}(\sum_X^{\neq} \mathcal{F})$.

Now we will show that $\mathcal{G}(\sum_X^{\neq} \mathcal{F}) \subseteq \sum_X \mathcal{G}(\mathcal{F})$.

Consider a tuple $t' \in \mathcal{G}(\sum_X^{\neq} \mathcal{F})$ where the value of t' is v. By the definition of $\mathcal{G}()$, there exists a #VE$_{\neq}$ tuple $t = \langle c_{y_1}, \ldots, c_{y_n}, \Pi \rangle \in \sum_X^{\neq} \mathcal{F}$ with value v such that $t' \in t$. Then, by the definition of $\sum^{\neq}$, there exist tuples $t_1 = \langle c_{y_1}, \ldots, c_{x_1}, \ldots, c_{y_n}, \Pi_1 \rangle$, $\ldots, t_j = \langle c_{y_1}, \ldots, c_{x_j}, \ldots, c_{y_n}, \Pi_j \rangle \in \mathcal{F}$ (for partitions $\Pi_i, 1 \leq i \leq j$, if we remove the partition containing X, we obtain partitions Π) with values $v_1, \ldots, v_j$, with t obtained by adding together $t_1, \ldots, t_j$. Let us denote the effective count of t_i with respect to X by $ec(t_i, X)$, $1 \leq i \leq j$. Then we have $v = \sum_{i=1}^{j} v_i \cdot ec(t_i, X)$.

Tuple t' corresponds to the sum of $ec(t_1, X)$ tuples from $\mathcal{G}(t_1)$ with value v_1, $ec(t_2, X)$ tuples from $\mathcal{G}(t_2)$ with value v_2, ..., $ec(t_j, X)$ tuples from $\mathcal{G}(t_j)$ with value v_j. All of these ground tuples were identical to t' at positions of variables $Y_1, \ldots, Y_n$; no other tuples in $\mathcal{G}(\mathcal{F})$ had this property. The #VE algorithm would add all of those tuples and obtain t' as a result, hence $t' \in \sum_X \mathcal{G}(\mathcal{F})$. □

4.3.2.5 The algorithm

The #VE$_{\neq}$ algorithm is presented in Figure 4.9. It reassembles the #VE algorithm (see Section 4.2.2 and Figure 4.1 on page 110) as well as the VE algorithm for inference in belief networks (see Section 2.3.2 and Figure 2.2 on page 10).

As in the #VE algorithm for #CSP, we maintain the following invariant: at each step of the computation, the product of all factors that exist at this step associates with its tuples the number of consistent extensions to the previously eliminated variables. If, given a CSP instance $\mathsf{P} = (\mathsf{X}, \mathsf{D}, \mathsf{C})$, we use the procedure $\#\mathsf{CSP\_VE}_{\neq}(\mathsf{P}, \mathsf{E}, \mathsf{H})$ presented in Figure 4.9 to eliminate all variables ($\mathsf{E} = \mathsf{X}$), we obtain a factor on the empty set of variables, which is simply a number equal to the number of solutions to the input CSP instance. As in the case of the #VE algorithm, if we decide to eliminate only some of the variables ($\mathsf{E} \subset \mathsf{X}$), we end up with a factor giving us the number of solutions for each combination of values of variables $\mathsf{X} \setminus \mathsf{E}$.

A procedure for constructing a #VE$_{\neq}$ factor-based representation of a CSP instance is shown in Figure 4.10. The procedure does not create 0-valued tuples. It also assumes that s-constants and sizes of the corresponding disjoint sets of domain values are given as the input data. In Section 4.3.4.1, we comment on the complex-

```
[00]  procedure #CSP_VE≠(P, E, H)
[01]    input: CSP instance P = (X, D, C) with only inequality constraints
[02]                          domains in D consist of s disjoint sets S_1, ..., S_s
[03]                          sets are represented through s s-constants c_1, ..., c_s
[04]                          s-constants have counts size(c_1), ..., size(c_s),
[05]           set of variables to eliminate E ⊆ X,
[06]           elimination ordering heuristic H;
[07]    output: factor representing the solution;
[08]    set F := initialize_VE≠(P);
[09]    while there is a factor in F involving a variable from E do
[10]      select variable X_i ∈ E according to H;
[11]      set F := eliminate_VE≠(X_i, F, size(c_1), ..., size(c_s));
[12]      set E := E \ X_i;
[13]    end
[14]    return ⊙^≠_{F∈F} F;
[15]  end

[16]  procedure eliminate_VE≠(X_i, F, size(c_1), ..., size(c_s))
[17]    input: variable to be eliminated X_i,
[18]           set of factors F,
[19]           counts size(c_1), ..., size(c_s);
[20]    output: set of factors F with X_i summed out;
[21]    partition F = {F_1, ..., F_m} into {F_1, ..., F_l} that do not contain X_i and
[22]                                       {F_{l+1}, ..., F_m} that do contain X_i;
[23]    return {F_1, ..., F_l, Σ^≠_{X_i} F_{l+1} ⊙^≠ ... ⊙^≠ F_m};
[24]  end
```

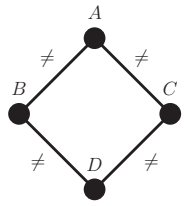
Figure 4.9: #VE$_{\neq}$ algorithm for #CSP with inequality constraints.

ity of constructing these data from the most basic, array-based representation of the input CSP instance. In Appendix C we show an example which illustrates how it is computed in our implementation of the lifted probabilistic inference algorithm.

In the next section, we apply our #VE$_{\neq}$ algorithm to a problem with large domains.

```
[00]  procedure initialize_VE≠(P)
[01]    input: CSP instance P = (X, D, C);
[02]    output: representation of P as a set of factors F;
[03]    set F := ∅;
[04]    for every binary inequality constraint C_i in C, where S(C_i) = {X_u, X_v} do
[05]      create factor F_i on X_u, X_v:
[06]        if S_i ⊂ D̂(X_u), S_j ⊂ D̂(X_v) and S_i ≠ S_j
[07]          create tuple ⟨c_i, c_j, [{{X_u}} {{X_v}}]⟩ with value 1;
[08]        if S_i ⊂ D̂(X_u), S_i ⊂ D̂(X_v) and size(c_i) > 1
[09]          create tuple ⟨c_i, c_i, [{{X_u}, {X_v}}]⟩ with value 1;
[10]      set F := F ∪ {F_i};
[11]    end
[12]    return F;
[13]  end
```

Figure 4.10: Initialization procedure for the #VE$_{\neq}$ algorithm.



$$\mathsf{D}(A) = \{x_1, x_2, \ldots, x_{10000}\}$$
$$\mathsf{D}(B) = \mathsf{D}(C) = \mathsf{D}(D) = \{x_1, x_2, \ldots, x_{5000}\}$$

$$\begin{aligned}\mathsf{C} = \{&A \neq x_1, A \neq x_3, A \neq x_4, A \neq x_5, A \neq B, A \neq C,\\ &B \neq x_1, B \neq x_2, B \neq x_3, B \neq x_5, B \neq D,\\ &C \neq x_1, C \neq x_3, C \neq x_5, C \neq D,\\ &D \neq x_1, D \neq x_2, D \neq x_3, D \neq x_5\}.\end{aligned}$$

Figure 4.11: A CSP used in Section 4.3.3.

4.3.3 Example computation

Consider the CSP instance presented in Figure 4.11. We want to compute the number of models for different values of D using our algorithm, with elimination ordering $\rho = \langle A, B, C \rangle$.

After processing all unary constraints, we obtain: $\widehat{\mathsf{D}}(A) = \{x_2, x_6, \ldots, x_{10000}\}$, $\widehat{\mathsf{D}}(C) = \{x_2, x_4, x_6, \ldots, x_{5000}\}$, and $\widehat{\mathsf{D}}(B) = \widehat{\mathsf{D}}(D) = \{x_4, x_6, \ldots, x_{5000}\}$.

Let $S_0 = \{x_2\}$, $S_1 = \{x_4\}$, $S_2 = \{x_6, \ldots, x_{5000}\}$, and $S_3 = \{x_{5001}, \ldots, x_{10000}\}$. Then $\widehat{\mathsf{D}}(A) = S_0 \cup S_2 \cup S_3$, $\widehat{\mathsf{D}}(B) = S_1 \cup S_2$, $\widehat{\mathsf{D}}(C) = S_0 \cup S_1 \cup S_2$ and $\widehat{\mathsf{D}}(D) = S_1 \cup S_2$. Let s-constant c_i denote S_i, $i = 0, 1, 2, 3$. The CSP instance can be described using four #VE$_{\neq}$ factors and four numbers shown below. In what follows, #VE$_{\neq}$ tuples that are typeset in bold are pruned, either because they are assigned value 0, or because the effective count of one of the s-constants in the #VE$_{\neq}$ tuple is 0.

(i)

A	B	Partition(s)	#
c_0	c_1	$\{\{A\}\}\{\{B\}\}$	1
c_0	c_2	$\{\{A\}\}\{\{B\}\}$	1
c_2	c_1	$\{\{A\}\}\{\{B\}\}$	1
$\mathbf{c_2}$	$\mathbf{c_2}$	$\{\{\mathbf{A},\mathbf{B}\}\}$	**0**
c_2	c_2	$\{\{A\},\{B\}\}$	1
c_3	c_1	$\{\{A\}\}\{\{B\}\}$	1
c_3	c_2	$\{\{A\}\}\{\{B\}\}$	1

(ii)

A	C	Partition(s)	#
$\mathbf{c_0}$	$\mathbf{c_0}$	$\{\{\mathbf{A},\mathbf{C}\}\}$	**0**
$\mathbf{c_0}$	$\mathbf{c_0}$	$\{\{\mathbf{A}\},\{\mathbf{C}\}\}$	**1**
c_0	c_1	$\{\{A\}\}\{\{C\}\}$	1
c_0	c_2	$\{\{A\}\}\{\{C\}\}$	1
c_2	c_0	$\{\{A\}\}\{\{C\}\}$	1
c_2	c_1	$\{\{A\}\}\{\{C\}\}$	1
$\mathbf{c_2}$	$\mathbf{c_2}$	$\{\{\mathbf{A},\mathbf{C}\}\}$	**0**
c_2	c_2	$\{\{A\},\{C\}\}$	1
c_3	c_0	$\{\{A\}\}\{\{C\}\}$	1
c_3	c_1	$\{\{A\}\}\{\{C\}\}$	1
c_3	c_2	$\{\{A\}\}\{\{C\}\}$	1

(iii)

B	D	Partition(s)	#
$\mathbf{c_1}$	$\mathbf{c_1}$	$\{\{\mathbf{B},\mathbf{D}\}\}$	**0**
$\mathbf{c_1}$	$\mathbf{c_1}$	$\{\{\mathbf{B}\},\{\mathbf{D}\}\}$	**1**
c_1	c_2	$\{\{B\}\}\{\{D\}\}$	1
c_2	c_1	$\{\{B\}\}\{\{D\}\}$	1
$\mathbf{c_2}$	$\mathbf{c_2}$	$\{\{\mathbf{B},\mathbf{D}\}\}$	**0**
c_2	c_2	$\{\{B\},\{D\}\}$	1

(iv)

C	D	Partition(s)	#
c_0	c_1	$\{\{C\}\}\{\{D\}\}$	1
c_0	c_2	$\{\{C\}\}\{\{D\}\}$	1
$\mathbf{c_1}$	$\mathbf{c_1}$	$\{\{\mathbf{C},\mathbf{D}\}\}$	**0**
$\mathbf{c_1}$	$\mathbf{c_1}$	$\{\{\mathbf{C}\},\{\mathbf{D}\}\}$	**1**
c_1	c_2	$\{\{C\}\}\{\{D\}\}$	1
c_2	c_1	$\{\{C\}\}\{\{D\}\}$	1
$\mathbf{c_2}$	$\mathbf{c_2}$	$\{\{\mathbf{C},\mathbf{D}\}\}$	**0**
c_2	c_2	$\{\{C\},\{D\}\}$	1

$size(c_0) = 1$
$size(c_1) = 1$
$size(c_2) = 4995$
$size(c_3) = 5000$

To eliminate A, we multiply factors (i) and (ii), and sum out A from their product. As a result we obtain factor (v):

A	B	C	Partition(s)	#
c_0	c_1	c_1	$\{\{A\}\}\{\{B,C\}\}$	1
$\mathbf{c_0}$	$\mathbf{c_1}$	$\mathbf{c_1}$	$\{\{\mathbf{A}\}\}\{\{\mathbf{B}\},\{\mathbf{C}\}\}$	**1**
c_0	c_1	c_2	$\{\{A\}\}\{\{B\}\}\{\{C\}\}$	1
c_0	c_2	c_1	$\{\{A\}\}\{\{B\}\}\{\{C\}\}$	1
c_0	c_2	c_2	$\{\{A\}\}\{\{B,C\}\}$	1
c_0	c_2	c_2	$\{\{A\}\}\{\{B\},\{C\}\}$	1
c_2	c_1	c_0	$\{\{A\}\}\{\{B\}\}\{\{C\}\}$	1
c_2	c_1	c_1	$\{\{A\}\}\{\{B,C\}\}$	1
$\mathbf{c_2}$	$\mathbf{c_1}$	$\mathbf{c_1}$	$\{\{\mathbf{A}\}\}\{\{\mathbf{B}\},\{\mathbf{C}\}\}$	**1**
c_2	c_1	c_2	$\{\{A\},\{C\}\}\{\{B\}\}$	1
c_2	c_2	c_0	$\{\{A\},\{B\}\}\{\{C\}\}$	1
c_2	c_2	c_1	$\{\{A\},\{B\}\}\{\{C\}\}$	1
c_2	c_2	c_2	$\{\{A\},\{B,C\}\}$	1
c_2	c_2	c_2	$\{\{A\},\{B\},\{C\}\}$	1
c_3	c_1	c_0	$\{\{A\}\}\{\{B\}\}\{\{C\}\}$	1
c_3	c_1	c_1	$\{\{A\}\}\{\{B,C\}\}$	1
$\mathbf{c_3}$	$\mathbf{c_1}$	$\mathbf{c_1}$	$\{\{\mathbf{A}\}\}\{\{\mathbf{B}\},\{\mathbf{C}\}\}$	**1**
c_3	c_1	c_2	$\{\{A\}\}\{\{B\}\}\{\{C\}\}$	1
c_3	c_2	c_0	$\{\{A\}\}\{\{B\}\}\{\{C\}\}$	1
c_3	c_2	c_1	$\{\{A\}\}\{\{B\}\}\{\{C\}\}$	1
c_3	c_2	c_2	$\{\{A\}\}\{\{B,C\}\}$	1
c_3	c_2	c_2	$\{\{A\}\}\{\{B\},\{C\}\}$	1

$\xrightarrow{\sum_A^{\neq}}$

(v)

B	C	Partition(s)	#
c_1	c_0	$\{\{B\}\}\{\{C\}\}$	9995
c_1	c_1	$\{\{B,C\}\}$	9996
c_1	c_2	$\{\{B\}\}\{\{C\}\}$	9995
c_2	c_0	$\{\{B\}\}\{\{C\}\}$	9994
c_2	c_1	$\{\{B\}\}\{\{C\}\}$	9995
c_2	c_2	$\{\{B,C\}\}$	9995
c_2	c_2	$\{\{B\},\{C\}\}$	9994

To eliminate B, we multiply factors (iii) and (v), and sum out B from their product. As a result we obtain factor (vi):

B	C	D	Partition(s)	#
c_1	c_0	c_2	$\{\{B\}\}\{\{C\}\}\{\{D\}\}$	9995
c_1	c_1	c_2	$\{\{BC\}\}\{\{D\}\}$	9996
c_1	c_2	c_2	$\{\{B\}\}\{\{C,D\}\}$	9995
c_1	c_2	c_2	$\{\{B\}\}\{\{C\},\{D\}\}$	9995
c_2	c_0	c_1	$\{\{B\}\}\{\{C\}\}\{\{D\}\}$	9994
c_2	c_0	c_2	$\{\{B\},\{D\}\}\{\{C\}\}$	9994
c_2	c_1	c_1	$\{\{B\},\{\{C,D\}\}$	9995
$\mathbf{c_2}$	$\mathbf{c_1}$	$\mathbf{c_1}$	$\{\{\mathbf{B}\},\{\{\mathbf{C}\},\{\mathbf{D}\}\}$	**9995**
c_2	c_1	c_2	$\{\{B\},\{D\}\}\{\{C\}\}$	9995
c_2	c_2	c_1	$\{\{B,C\}\}\{\{D\}\}$	9995
c_2	c_2	c_1	$\{\{B\},\{C\}\}\{\{D\}\}$	9994
c_2	c_2	c_2	$\{\{B\},\{C,D\}\}$	9994
c_2	c_2	c_2	$\{\{B,C\},\{D\}\}$	9995
c_2	c_2	c_2	$\{\{B\},\{C\},\{D\}\}$	9994

$\xrightarrow{\sum_B^{\neq}}$

(vi)

C	D	Partition(s)	#
c_0	c_1	$\{\{C\}\}\{\{D\}\}$	49920030
c_0	c_2	$\{\{C\}\}\{\{D\}\}$	49920031
c_1	c_1	$\{\{C,D\}\}$	49925025
c_1	c_2	$\{\{C\}\}\{\{D\}\}$	49925026
c_2	c_1	$\{\{C\}\}\{\{D\}\}$	49920031
c_2	c_2	$\{\{C,D\}\}$	49920031
c_2	c_2	$\{\{C\},\{D\}\}$	49920032

Finally, to eliminate C, we multiply factors (iv) and (vi), and sum out C from their product. As a result we obtain factor (vii):

C	D	Partition(s)	#
c_0	c_1	$\{\{C\}\}\{\{D\}\}$	49920030
c_0	c_2	$\{\{C\}\}\{\{D\}\}$	49920031
c_1	c_2	$\{\{C\}\}\{\{D\}\}$	49925026
c_2	c_1	$\{\{C\}\}\{\{D\}\}$	49920031
c_2	c_2	$\{\{C\},\{D\}\}$	49920032

$\xrightarrow{\Sigma_C^{\neq}}$ (vii)

D	Partition(s)	#
c_1	$\{\{D\}\}$	249400474875
c_2	$\{\{D\}\}$	249400484865

To obtain the final result, we would normally multiply all the remaining factors, but for this example we are left only with factor (vii), so it provides our solution: for $D = x_4$, the given CSP instance has 249 400 474 875 solutions, and for each value x_i from $D = \{x_6, x_7, \ldots, x_{5000}\}$, it has 249 400 484 865 solutions.

Eliminating D from factor (vii) would give us the overall number of solutions:

$$|\mathsf{P}| = 249\,400\,474\,875 + 249\,400\,484\,865 \cdot 4995 = 1\,246\,004\,822\,375\,550\,.$$

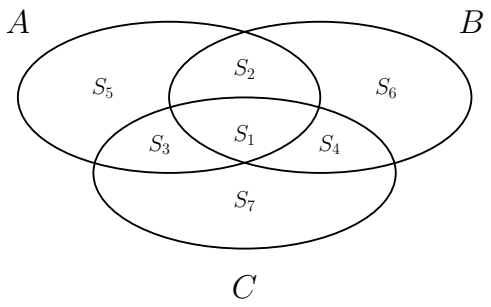
4.3.4 Complexity of the algorithm

In Section 4.3.4.1, we discuss the complexity of constructing s-constants and sizes of the corresponding disjoint sets of domain values. In Section 4.3.4.2, we analyze the complexity of inference in #VE$_{\neq}$.

4.3.4.1 Preprocessing

The complexity of the preprocessing phase depends highly on the representation of the input problem instance. One can expect that if the domains of the variables in a CSP instance are very large, then they will be represented using functions and relations. Such representation might allow for a very efficient construction of disjoint subsets without enumerating the values of any domain. (see Appendix C). To obtain a fair comparison of the preprocessing costs for #VE and #VE$_{\neq}$ we analyze below the most basic, array-based representation, which does not favor the #VE$_{\neq}$ algorithm.

Assume that a CSP instance has n variables with maximum domain size d_{max}, r binary constraints and some number of unary constraints. Let U be a union of the domains of all n variables without values excluded by unary constraints. We will



$$A = S_1 \cup S_2 \cup S_3 \cup S_5$$

$$B = S_1 \cup S_2 \cup S_4 \cup S_6$$

$$C = S_1 \cup S_3 \cup S_4 \cup S_7$$

Figure 4.12: Domains of three variables. We need at most seven disjoint subsets, $S_1, S_2, \ldots, S_7$, to specify domains of the variables.

consider a representation, where binary constraints are stored as a list of r tuples and values of the domains and unary constraints are stored in the arrays.

We need to process unary constraints and we need to divide set U into disjoint subsets so that domains of variables can be represented using s-constants corresponding to these subsets. In the worst case we need to split U into $2^n - 1$ such subsets, one subset of values per each non-empty subset of the set of variables (see Figure 4.12).

In order to do the above task efficiently, we sort the n arrays storing domains and the n arrays storing unary constraints (time complexity $\mathcal{O}(n\, d_{max} \log(d_{max}))$). The choice of the ordering is irrelevant as long as it is a strict total ordering. With each array, we associate a pointer, initially set to the first element of the sorted array. We will refer to the elements pointed at as to current elements. We also create a data structure of size $2^n - 1$ for storing counts associated with each potential subset of U[1].

Next, we proceed in a manner reassembling the merging algorithm. We choose the smallest current element x from n domain arrays. We check which of other current elements of domain arrays are equal to x, which gives as a set of variables whose domains contain x. We check which of current elements of unary constraints arrays are equal to x, which gives as a set of variables for which x is excluded from their domains. This information allows us to update the count for the respective

[1] A sparse representation should be considered as one can expect many of the counts to be 0.

disjoint set. After the count is updated, we move forward the pointers associated with the elements and constraints equal to x.

We repeat this step for each of at most $n\,d_{max}$ values in U. The time complexity of this operation is $\mathcal{O}(n\,d_{max}(n+\log(2^n-1))) = \mathcal{O}(n^2 d_{max})$. The space complexity is $\mathcal{O}(n\,d_{max}+2^n)$.

Given the disjoint subsets and their counts, we can create #VE$_{\neq}$ factors to represent all r binary constraints. There are at most 2^{n-1} s-constants corresponding to the disjoint subsets with non-zero counts. These s-constants are used to represent domains of variables. At the same time the domain of each variable is represented by at most d_{max} s-constants, as we do not use s-constants to represent empty subsets. Therefore the domain of each variable is represented by at most $\min(\{2^{n-1}, d_{max}\})$ s-constants.

Each #VE$_{\neq}$ factor has at most $\min(\{2^{n-1}, d_{max}\})^2 + \min(\{2^{n-1}, d_{max}\})$ #VE$_{\neq}$ tuples. This is because there are $\min(\{2^{n-1}, d_{max}\})^2$ possible pairs of s-constants and for $\min(\{2^{n-1}, d_{max}\})$ pairs of identical s-constants, there are two possible partitions of associated two variables: one in which variables are in the same block and one in which they are in different blocks[2]. Thus, the time and space complexity of creating the #VE$_{\neq}$factors is $\mathcal{O}(r\,(\min(\{2^{n-1}, d_{max}\})^2 + \min(\{2^{n-1}, d_{max}\})2^{2n-2}))$.

The time complexity of the whole preprocessing phase for the array-based representation is

$$\mathcal{O}(n\,d_{max}(\log(d_{max})+n) + r\,(\min(\{2^{n-1}, d_{max}\})^2 + \min(\{2^{n-1}, d_{max}\}))),$$

and the space complexity is

$$\mathcal{O}(r + n\,d_{max} + 2^n + r\,(\min(\{2^{n-1}, d_{max}\})^2 + \min(\{2^{n-1}, d_{max}\}))\,.$$

For the same CSP instance, a naive #VE implementation creates the representation in time $\mathcal{O}(r d_{max}^2)$. However, most VE implementations first sort domains of the variables (as this allows for a 1-dimensional representation of factors as well as

[2]If we do not represent zero-valued #VE$_{\neq}$ tuples in the #VE$_{\neq}$ factors, the maximum size of the #VE$_{\neq}$ factor becomes $\min(\{2^{n-1}, d_{max}\})^2$ as for pairs of identical s-constants we then only represent these cases where variables are in different blocks of the partition.

speeding up the processing of constraints). In such cases, the time and space complexities are respectively $\mathcal{O}(nd_{max}\log(d_{max}) + rd_{max}^2)$ and $\mathcal{O}(r + nd_{max} + rd_{max}^2)$.

Therefore the cost of the preprocessing phase for #VE$_{\neq}$ does not have to be greater than the cost for #VE. In many cases, #VE$_{\neq}$ constructs a network much faster and is less likely to run out of memory (see Section 4.3.5).

4.3.4.2 Inference

Assume that #VE$_{\neq}$ factors given as the input data to the #VE$_{\neq}$ algorithm involve s s-constants. As in the case of the standard #VE algorithm for #CSP, the time and space complexities of the inference phase of the #VE$_{\neq}$ algorithm are determined by the size of the biggest #VE$_{\neq}$ factor, which depends on the induced tree width $w^*(\rho)$ of the associated constraint graph.

For the elimination ordering ρ, the biggest #VE$_{\neq}$ factor represents a function on $w^*(\rho)+1$ variables. In the worst case, the domain of each variable in the biggest factor is represented by all s s-constants, and the factor may need to represent all $s^{w^*(\rho)+1}$ possible combinations of those s-constants.

For each combination, there are possibly several #VE$_{\neq}$ tuples in the factor. For a combination containing s-constants $c_1, c_2, \ldots, c_k$, if each s-constant occurs in the combination $n_{c_1}, n_{c_2}, \ldots, n_{c_k}$ times, respectively (where $n_{c_1} + n_{c_2} + \cdots + n_{c_k} = w^*(\rho) + 1$), then there may be up to $\varpi_{n_{c_1}}\varpi_{n_{c_2}}\ldots\varpi_{n_{c_k}}$ #VE$_{\neq}$ tuples in the factor. Proposition 4.1 tells us that this number varies between 0 and $\varpi_{w^*(\rho)+1}$.

Thus, the size of the biggest #VE$_{\neq}$ factor can be at most:

$$s^{w^*(\rho)+1}\varpi_{w^*(\rho)+1}.$$

The total number of #VE$_{\neq}$ factors processed during the execution of the algorithm is bounded by $2r$, where r is the number of constraints. Therefore the space complexity of our algorithm is equal to

$$\mathcal{O}\left(rs^{w^*(\rho)+1}\varpi_{w^*(\rho)+1}\right). \tag{4.6}$$

The time complexity of the algorithm is equal to the space complexity times some polynomial cost of manipulating partitions $P(w^*(\rho)+1)$. For both space and time

complexities, the only direct domain-size dependency is the actual length of the numbers we store, multiply, sum and output.

In comparison to the space complexity of the #VE algorithm (see Equation 4.1), the component $d^{w^*(\rho)+1}$ (where d is the size of the largest domain) is possibly greatly reduced to $s^{w^*(\rho)+1}$, but we have an additional, possibly large, component $\varpi_{w^*(\rho)+1}$. The plot from Figure 4.2 suggests that for moderate values of $w^*(\rho)$ and large domains #VE$_{\neq}$ should achieve much lower space complexity than #VE. A comparison of time complexities of both algorithm leads to the same conclusion. There, #VE$_{\neq}$ has an additional, polynomial overhead related to the cost of manipulating partitions.

4.3.5 Empirical evaluation

We implemented in Java both the #VE algorithm[3] by Dechter [2003] presented in Section 4.2.2 and the #VE$_{\neq}$ algorithm[4] introduced in Section 4.3.2, and compared their performances. We wrote the code using J2SDK 1.4.2.06 and ran it under JRE 1.5.0.01. We performed the experiments on an Intel Pentium IV 3.20GHz machine with 1GB of RAM was made available to the Java Virtual Machine. For accurate timing, we used the method described in [Gørtz, 2000].

The purpose of our tests was to answer the following questions. How does more complex preprocessing affects #VE$_{\neq}$ performance compared to #VE? How do rich data structures used by #VE$_{\neq}$ affect it performance compared to #VE? In the previous section we computed the asymptotic complexities of the preprocessing and inference phases of both algorithms. Even though the #VE$_{\neq}$ algorithm performs better asymptotically as the domain size increases, we do not know answers to these questions because we do not know the constants associated with these complexities. The constants for the #VE$_{\neq}$ algorithm could be so large as to make it impractical.

To estimate the constants associated with each inference method, we performed tests on random CSP instances. We sampled instances with 5 to 50 variables. Each of possible binary inequality constraints was chosen independently (as it would be

[3] http://people.cs.ubc.ca/~kisynski/code/ve_hcsp/

[4] http://people.cs.ubc.ca/~kisynski/code/ve_in_hcsp/

done using standard generation models A and C, see Achlioptas et al. [1997]) with probability varying from 0.1 to 0.5.

We only considered instances that theoretically could be solved by the #VE algorithm given 1GB of memory. As a result, the induced width varied from 2 to 32 and the domain size varied from 2 to 2^{16} (all variables in one instance having the same domain).

We generated two sets of CSPs, each containing 103 instances. In the first set each of possible unary constrains was chosen independently with the probability 0.1 (which resulted in big differences between the domains of variables after unary constraints were processed). In the second set the probability was equal to 0.01 (so domains were more similar).

We considered four elimination-ordering heuristics – max-cardinality, min-degree, min-factor and min-fill (see Kjærulff [1990] for an overview of these heuristics). The last three achieved similar performance, superior to the max-cardinality scheme. We report results achieved with the min-degree heuristic, as it is the simplest and the fastest to compute.

To allow for the variation in run time due to garbage collection and other covariates, all computations were performed five times. We report mean run time and, where necessary, standard error. In all of our experiments values of the domains of the variables, unary constraints and binary constraints in input CSP instances were stored in the arrays in an arbitrary order. We analyzed the amount of time #VE and #VE$_{\neq}$ spent on constructing their internal representations given array-based descriptions of CSPs and the amount of time they spent doing inference.

In Figure 4.13 we present counts of the outcomes of the experiment. The #VE$_{\neq}$ algorithm never failed during the preprocessing phase. During the inference phase, it failed only 3 times for instances for which the #VE algorithm succeeded. #VE$_{\neq}$ was faster than #VE for most instances during both preprocessing and inference.

In Figure 4.14 we further summarize the results. For each set of instances we show the total time the algorithms spent on solving all 103 instances. If an algorithm failed to finish a computation for an instance, we included the time it spent on the computation until failure. For the instances with 0.1 probability of a value being excluded by a unary constraint, #VE was not only slower, but also failed to compute the result for 29 instances, while #VE$_{\neq}$ failed for only 11 instances. In

$\mathcal{P}(u)$	Cases			Counts	
	#VE fails	#VE$_{\neq}$ fails	#VE faster than #VE$_{\neq}$	Preprocessing	Inference
0.1	Yes	Yes	N/A	0	9
	Yes	No	N/A	8	12
	No	Yes	N/A	0	2
	No	No	Yes	25	21
			No	70	51
0.01	Yes	Yes	N/A	0	8
	Yes	No	N/A	8	13
	No	Yes	N/A	0	1
	No	No	Yes	21	15
			No	74	58

Figure 4.13: Summary of experiments. $\mathcal{P}(u)$ - probability of a value being excluded by a unary constraint.

$\mathcal{P}(u)$	Algor.	Preprocessing			Inference			Total		
		time [s]	s_e [s]	#f	time [s]	s_e [s]	#f	time [s]	s_e [s]	#f
0.10	#VE	21.85	0.16	8	99 358.84	3 475.86	21	99 380.69	3 475.79	29
	#VE$_{\neq}$	0.60	0.01	0	49 175.77	781.37	11	49 176.36	781.37	11
0.01	#VE	21.02	0.11	8	77 749.75	2 321.95	21	77 770.78	2 321.93	29
	#VE$_{\neq}$	0.45	0.01	0	31 260.52	2 133.07	9	31 260.97	2 133.08	9

Figure 4.14: Summary of experiments. $\mathcal{P}(u)$ - probability of a value being excluded by a unary constraint; #f - number of times a particular algorithm failed due to lack of memory.

the second set, the probability of a value being excluded by a unary constraint was equal to 0.01; for those instances, #VE$_{\neq}$ also achieved much better performance than #VE did. The experiments showed that more complicated preprocessing in the case of #VE$_{\neq}$ does not significantly influence performance, as the total run time is dominated by the inference phase. The experiments also showed that #VE$_{\neq}$ is capable of solving more instances than #VE can.

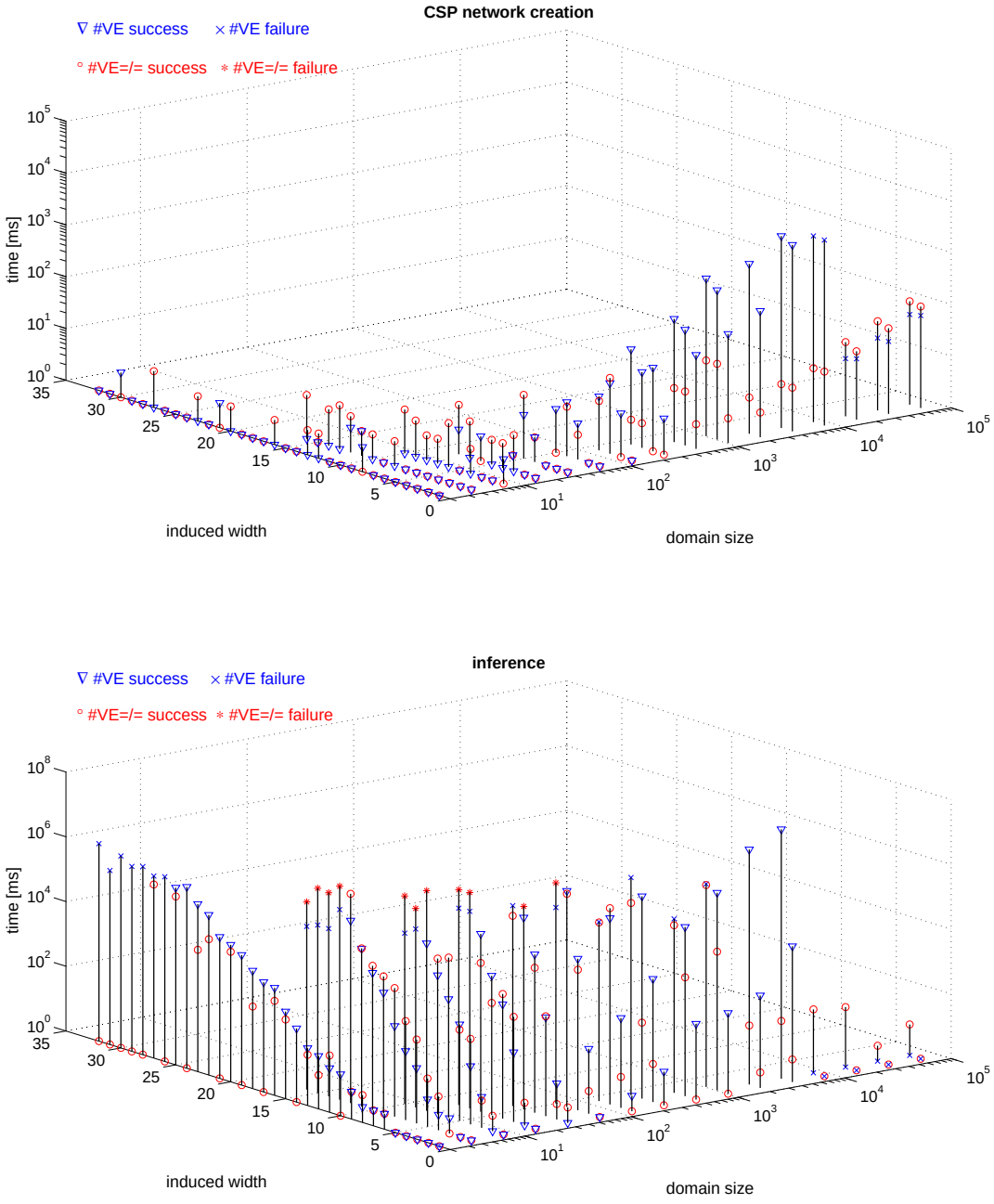


Figure 4.15: Results of experiments on CSP instances with the probability of a value being excluded by a unary constraint equal to 0.1.

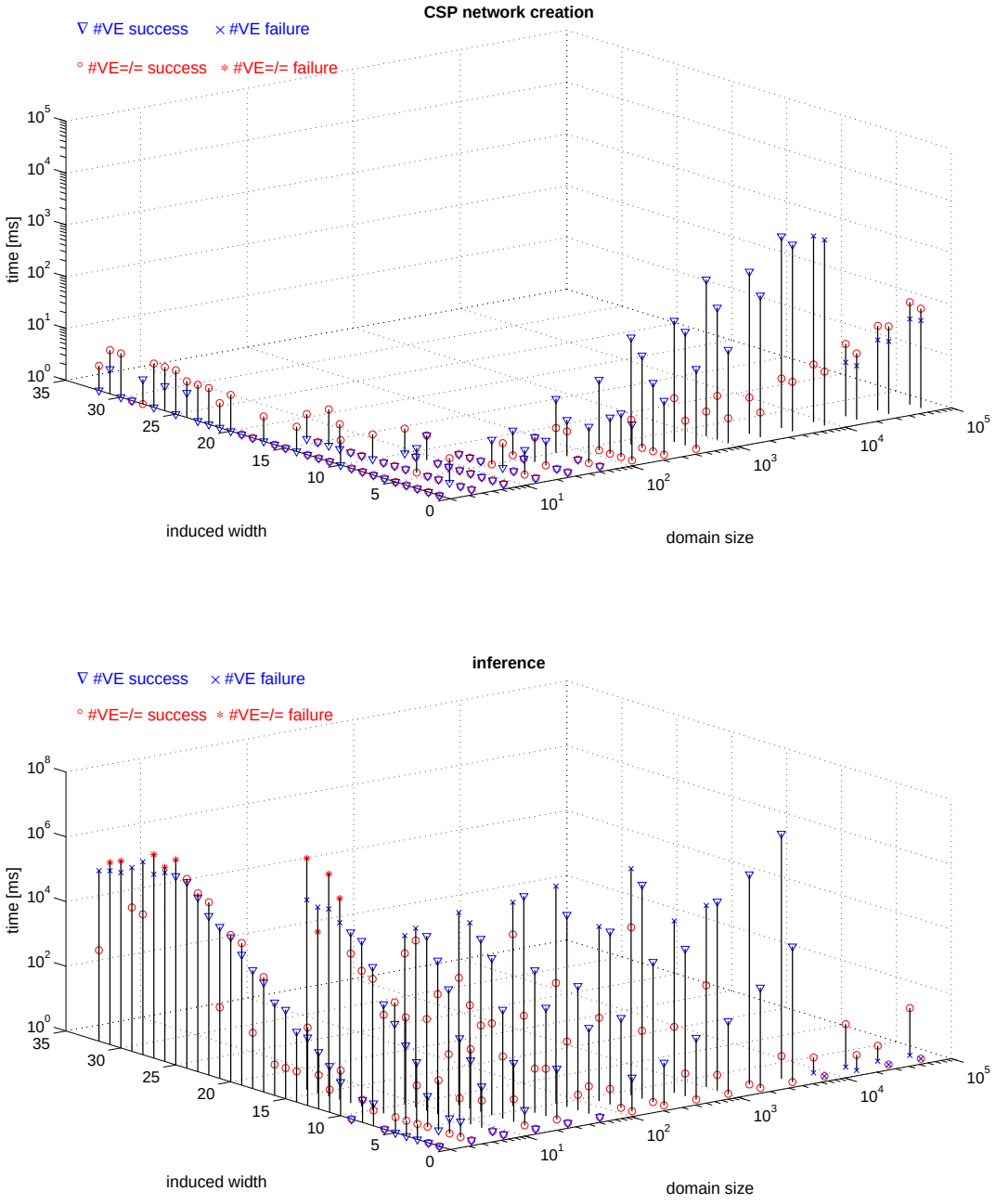


Figure 4.16: Results of experiments on CSP instances with the probability of a value being excluded by a unary constraint equal to 0.01.

Figure 4.15 visualizes the detailed results for the first set of instances, and Figure 4.16, for the second one. In both plots, each vertical line corresponds to one test instance.

It is important to remember that for all instances for which #VE$_{\neq}$ succeeded, we could keep increasing the domain size and the #VE$_{\neq}$ algorithm would still be able to compute the answer, while the #VE algorithm would sooner or later run out of memory.

4.4 Conclusions

We approached #CSP for a practical reason: we needed an efficient, lifted algorithm to solve #CSP induced during lifted probabilistic inference practical. Such CSP instances involve only inequality constraints (either between a pair of variables or between a variable and a constant) and domains of variables might be very large. The first property turned out to be sufficient to allow for the development of an efficient, lifted method based on the variable elimination framework. The time and space complexities of our algorithm do not depend directly on the domain size. Moreover, because our algorithm is a lifted one, it can inter-operate with lifted probabilistic inference engine in a natural way (see Appendix C and Appendix D).

There has recently been work on coloring problems by Angelsmark and Thapper [2006]; Björklund et al. [2009]. This work does not solve our problems as the inputs and outputs of these algorithms are not lifted. However, it is possible that our algorithm could be applied to their problems, which remains an intriguing possibility.

Chapter 5

Constraint Processing in Lifted Inference

So I said to the Gym instructor "Can you teach me to do the splits?"
He said "How flexible are you?". I said "I can't make Tuesdays".
— Tim Vine

5.1 Introduction

In this chapter we analyze lifted inference from the perspective of constraint processing and, through this viewpoint, we analyze and compare existing approaches to constraint processing and expose their advantages and limitations. Our theoretical results show that the wrong choice of constraint processing method can lead to exponential increase in computational complexity. Our empirical tests confirm the importance of constraint processing in lifted inference.

In Section 5.2 we give an overview of constraint processing during lifted probabilistic inference. In Section 5.3 we compare the *splitting as needed* approach of Poole [2003] with the *shattering* approach of de Salvo Braz et al. [2007]. In Section 5.4 we compare two approaches to solving #CSPs encountered during lifted inference: the use of a specialized #CSP solver [de Salvo Braz et al., 2007; Poole, 2003] and the requirement that the parfactors used during inference be in *normal*

form [Milch et al., 2008], which simplifies #CSP solving, but requires additional splitting.

5.2 Overview of constraint processing in lifted inference

Specification of a first-order probabilistic model consists of probabilistic statements that may involve inequalities between a logical variable (that represents an individual in a population) and a constant (an individual from the population) as well as between two logical variables.

Due to prior knowledge or observations, we may know that some individual(s) should be treated differently from the rest of the population. In that case, statements about the rest of the population require the constraint that the logical variable is not one of the exceptional individuals. For example, if we know that everyone is Joe's friend, for the parameterized random variable $friends(X,Y)$, we may treat the cases $Y = joe$ and $Y \neq joe$ as separate cases. Example 2.8 shows constraints induced by prior knowledge. There are also different cases where two logical variables are equal or not.

Example 5.1. We want to specify a conditional probability describing the likelihood that two people are friends, provided they like each other. It is natural to have two parameterized random variables: $likes(X,Y)$ and $friends(X,Y)$, where X and Y are logical variables with the same population representing a group of people. Obviously, the case of whether someone likes themself ($X = Y$) should be treated differently from the case of whether someone likes someone else ($X \neq Y$).

Inequality constraints can be introduced by observing. For example, if we observed that Joe and Peter are friends, we need to treat $friends(joe, peter)$ separately from $friends(X,Y)$ for $X \neq joe$ and $Y \neq peter$. Section 2.5.2.7 contains another example of constrains introduced by observing.

Inequality constraints can also be introduced during inference. For example, if we have a probabilistic statement that contains the parameterized random variables $p(X)$ and $p(Y)$, we need to treat the $X = Y$ and $X \neq Y$ cases separately: in the first case, there is one random variable in each grounding; in the second case, there are two random variables in each grounding.

As pointed out in Poole [2003], when performing probabilistic reasoning within such models we need to take into account the population sizes of logical variables. For each probabilistic statement (an original one or one created during inference) we need to know how many random variables are represented by parameterized random variables within that statement. Poole [2003] gave example, where the probability that someone is guilty of a crime depends on how many other people could have committed the crime (i.e., the population size). The probability that a system will have a component with a fault depends on how many components there are. Thus, for exact inference we need to count the number of solutions to the CSP associated with the individuals and the inequality constraints, that is we need to solve the #CSP associated with the individuals and the inequality constraints.

In Section 2.5.2 we gave an overview of the C-FOVE algorithm [Milch et al., 2008]. In this section we look at it again, but this time we focus on constraint processing outlined above that is performed during lifted inference with C-FOVE.

Let us recall the notation from Section 2.5.2. Φ denotes a set of parfactors and $\mathbf{Q}$ denotes queried random variables, that is, a subset of the set of all random variables represented by parameterized random variables present in parfactors in Φ. Given Φ and $\mathbf{Q}$, the C-FOVE algorithm computes the marginal $\mathcal{J}_{\mathbf{Q}}(\Phi)$ by summing out random variables from $\mathbf{Q}$, where possible in a lifted manner. Evidence is handled by adding to Φ additional parfactors on observed random variables.

As we discussed in the previous section, parfactors in Φ can contain inequality constraints and parfactors representing observations are likely to explicitly name particular individuals. This makes parameterized random variables sharing the same functor represent different sets of random variables.

Example 5.2. Consider the first-order probabilistic model and its grounding presented in Figure 5.1. Let A and B be logical variables typed with a population $\mathcal{D}(A) = \mathcal{D}(B) = \{x_1, \ldots, x_n\}$. Let g and h be functors with range $\{false, true\}$. Assume we do not have any specific knowledge about ground instances of $g(A)$, but we have some specific knowledge about $h(A,B)$ for the case where $A = x_1$ and for the case where $A \neq x_1$ and $A = B$. We would represent the model using the

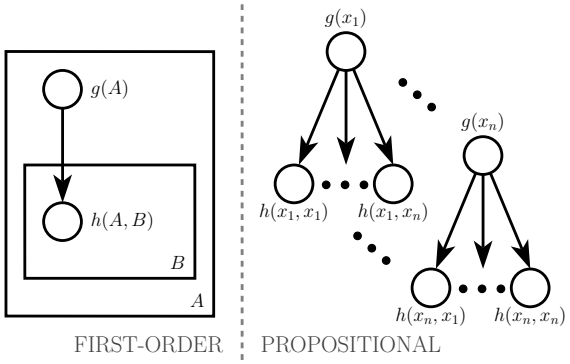


Figure 5.1: A first-order model from Example 5.2.

following parfactors:

$$\begin{aligned}
\Phi_0 = \{\, &\langle \emptyset, \{g(A)\}, \mathcal{F}_0 \rangle, && [0]\\
&\langle \emptyset, \{g(x_1), h(x_1, B)\}, \mathcal{F}_1 \rangle, && [1]\\
&\langle \{A \neq x_1\}, \{g(A), h(A,A)\}, \mathcal{F}_2 \rangle, && [2]\\
&\langle \{A \neq x_1, A \neq B\}, \{g(A), h(A,B)\}, \mathcal{F}_3 \rangle \}, && [3]
\end{aligned}$$

where $\mathcal{F}_0$ is a factor from $range(g)$ to the reals and $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ are factors from $range(g) \times range(h)$ to the reals, be a set of parfactors, such that $\mathcal{J}(\Phi_0)$ represents a joint probability distribution over random variables present in the model.

The sets of random variables represented by $g(A)$ and $g(x_1)$ in parfactor [0] and parfactor [1] are not disjoint and are not identical. The same applies to $g(A)$ in parfactor [0] and parfactors [2] and [3].

We will use the set of parfactors Φ_0 in a series of examples to follow. We will not compute a particular marginal, just highlight constraint processing involved in operations that can be performed within Φ_0.

5.2.1 Splitting and expanding

Before a ground instance of a parameterized random variable can be summed out, a number of conditions must be satisfied. One is that a ground instance of a parameterized random variable can be summed out from a parfactor in Φ only if there are

no other parfactors in Φ involving this ground instance (condition (S1) of Proposition 2.1 and condition (SC1) of Proposition 2.2). To satisfy this condition, the inference procedure may need to multiply parfactors prior to summing out.

Parfactor multiplication has a condition of its own: two parfactors $\langle \mathcal{C}_1, \mathcal{V}_1, \mathcal{F}_1 \rangle$ and $\langle \mathcal{C}_2, \mathcal{V}_2, \mathcal{F}_2 \rangle$ can be multiplied only if for each parameterized random variable from $\mathcal{V}_1$ and for each parameterized random variable from $\mathcal{V}_2$, the sets of random variables represented by these two parameterized random variables in respective parfactors are identical or disjoint (condition (M1) in Proposition 2.3). This condition is trivially satisfied for parameterized random variables with different functors.

In Section 2.5.2.3 we introduced parfactor splitting and expanding a counting formula, the two operations that are used during lifted inference to enable parfactor multiplication. Poole [2003] proposed a scheme in which splitting and expanding are performed as needed during inference when two parfactors are about to be multiplied and the precondition for multiplication is not satisfied. We call this approach *splitting as needed*. An alternative, called *shattering*, was proposed by de Salvo Braz et al. [2007]. The shattering operation performs all the splits and expansions that are required to ensure that for any two parameterized random variables present in parfactors, the sets of random variables represented by them are either identical or disjoint. We compare splitting as needed and shattering in Section 5.3.

Shattering is used in the C-FOVE algorithm of Milch et al. [2008]. Shattering is performed as the first operation of the inference and might be also necessary in the middle of inference if propositionalization or full expansion of a counting formula is performed. The C-FOVE algorithm also requires all parfactors to be in normal form, and the shattering operation might perform additional splits and expansions to bring all parfactors to normal form. An example computation presented in Section 2.5.2.7 includes a shattering operation. In Appendix E we show how to compute the same query in more efficient manner using splitting as needed.

The following example illustrates how to use splitting to make the sets of random variables represented by two parameterized random variables in different parfactors be identical or disjoint and how to use splitting to convert a parfactor to normal form.

Example 5.3. Let us continue from Example 5.2. The sets of random variables represented by $g(A)$ and $g(x_1)$ in parfactor [0] and parfactor [1] are not disjoint and are not identical: $g(A)$ and $g(x_1)$ unify with MGU $\{\{A/x_1\}\}$ (see Section 2.5.2.5 for an overview of the role of unification in lifted probabilistic inference). We split parfactor [0] on a substitution $\{A/x_1\}$, creating parfactors [4] and [5]:

$$\begin{aligned}
\Phi_1 = \{\, &\langle \emptyset, \{g(x_1), h(x_1, B)\}, \mathcal{F}_1 \rangle, && [1]\\
&\langle \{A \neq x_1\}, \{g(A), h(A, A)\}, \mathcal{F}_2 \rangle, && [2]\\
&\langle \{A \neq x_1, A \neq B\}, \{g(A), h(A, B)\}, \mathcal{F}_3 \rangle, && [3]\\
&\langle \emptyset, \{g(x_1)\}, \mathcal{F}_0 \rangle, && [4]\\
&\langle \{A \neq x_1\}, \{g(A)\}, \mathcal{F}_0 \rangle \}. && [5]
\end{aligned}$$

We have $\mathcal{J}(\Phi_1) = \mathcal{J}(\Phi_0)$. Sets of random variables represented by parameterized random variables present in Φ_1 are disjoint or identical, but parfactor [3] is not in normal form as $\mathcal{E}_A^{\mathcal{C}} \setminus \{B\} = \{x_1\} \neq \emptyset = \mathcal{E}_B^{\mathcal{C}} \setminus \{A\}$, where $\mathcal{C} = \{A \neq x_1, A \neq B\}$. To bring parfactor [3] to normal-from, we split on a substitution $\{B/x_1\}$, creating parfactors [6] and [7]:

$$\begin{aligned}
\Phi_2 = \{\, &\langle \emptyset, \{g(x_1), h(x_1, B)\}, \mathcal{F}_1 \rangle, && [1]\\
&\langle \{A \neq x_1\}, \{g(A), h(A, A)\}, \mathcal{F}_2 \rangle, && [2]\\
&\langle \emptyset, \{g(x_1)\}, \mathcal{F}_0 \rangle, && [4]\\
&\langle \{A \neq x_1\}, \{g(A)\}, \mathcal{F}_0 \rangle, && [5]\\
&\langle \{A \neq x_1\}, \{g(A), h(A, x_1)\}, \mathcal{F}_3 \rangle && [6]\\
&\langle \{A \neq x_1, A \neq B, B \neq x_1\}, \{g(A), h(A, B)\}, \mathcal{F}_3 \rangle \}. && [7]
\end{aligned}$$

We have $\mathcal{J}(\Phi_2) = \mathcal{J}(\Phi_1)$ and all parfactors in Φ_2 are in normal form.

5.2.2 Multiplication

Once the preconditions for parfactor multiplication are satisfied, multiplication can be performed in a lifted manner as described in Section 2.5.2.2. The lifted inference procedure needs to know how many factors each parfactor involved in the multiplication represents and how many factors their product will represent. These

numbers can be different because the product parfactor might involve more logical variables than a parfactor participating in the multiplication.

Given a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$, the number of factors it represents is equal to the number of solutions to the constraint satisfaction problem (CSP, see Section 4.2.1) formed by logical variables from $\mathcal{V}$ and constraints from $\mathcal{C}$, which means that we need to solve the associated #CSP. We compare existing approaches to this problem in Section 5.4.

Example 5.4. Assume that we want to multiply parfactors [4] and [1] from Φ_2 (Example 5.3). Parfactor [4], $\langle \emptyset, \{g(x_1)\}, \mathcal{F}_0 \rangle$, represents 1 factor while parfactor [1], $\langle \emptyset, \{g(x_1), h(x_1, B)\}, \mathcal{F}_1 \rangle$, represents n factors. Their product, a parfactor $\langle \emptyset, \{g(x_1), h(x_1, B)\}, \mathcal{F}_8 \rangle$, where $\mathcal{F}_8$ is a factor from $range(g) \times range(h)$ to the reals, represents n factors. We need to bring values of the factor $\mathcal{F}_0$ to the power $\frac{1}{n}$ (see Proposition 2.3) when computing $\mathcal{F}_8$: $\mathcal{F}_8 = \mathcal{F}_0^{\frac{1}{n}} \odot \mathcal{F}_3$. Let

$$
\begin{aligned}
\Phi_3 = \{ & \langle \{A \neq x_1\}, \{g(A), h(A, A)\}, \mathcal{F}_2 \rangle, && [2] \\
& \langle \{A \neq x_1\}, \{g(A)\}, \mathcal{F}_0 \rangle, && [5] \\
& \langle \{A \neq x_1\}, \{g(A), h(A, x_1)\}, \mathcal{F}_3 \rangle && [6] \\
& \langle \{A \neq x_1, A \neq B, B \neq x_1\}, \{g(A), h(A, B)\}, \mathcal{F}_3 \rangle, && [7] \\
& \langle \emptyset, \{g(x_1), h(x_1, B)\}, \mathcal{F}_8 \rangle \}. && [8]
\end{aligned}
$$

We have $\mathcal{J}(\Phi_3) = \mathcal{J}(\Phi_2)$ and all parfactors in Φ_3 are in normal form.

5.2.3 Summing out

During lifted summing out, a parameterized random variable is summed out from a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F}_{\mathcal{F}} \rangle$, which means that a random variable is eliminated from each factor represented by the parfactor in one inference step. Lifted inference will perform summing out only once on the factor $\mathcal{F}$. If some logical variables only appear in the parameterized variable that is being eliminated, the resulting parfactor will represent fewer factors than the original one. As in the case of parfactor multiplication, the inference procedure needs to compensate for this difference. Values of the factor component in the resulting parfactor are brought to the power r, where the number r tells us how many times fewer factors the result of summing

out represents compared to the original parfactor. This is described formally in Equations 2.3 and 2.5 from Propositions 2.1 and 2.2 respectively.

The number r is equal to the size of the set $(\mathcal{D}(X_1) \times \cdots \times \mathcal{D}(X_k)) : \mathcal{C}$, where $X_1, \ldots, X_k$ are logical variables that will disappear from the parfactor. If the parfactor is in normal form, the exponent r does not depend on values of logical variables remaining in the parfactor after summing out. In such case, computation reduces to solving #CSP and is not different from computation performed during parfactor multiplication.

Example 5.5. Assume that we want to sum out $ground(h(A,B)) : \{A \neq x_1, A \neq B, B \neq x_1\}$ from parfactor [7] from Φ_3 (Example 5.4) i.e., we want to sum out all ground instances of $h(A,B)$ that satisfy the constraints in [7]. The logical variable B will not appear in the resulting parfactor. We have $|\mathcal{D}(B) : \{A \neq x_1, A \neq B, B \neq x_1\}| = n-2$. Let $\mathcal{F}_9$ be a factor from $range(g)$ to the reals, $\mathcal{F}_9 = (\sum_{h(A,B)} \mathcal{F}_3)^{n-2}$. As a result of summation we obtain a parfactor $\langle \{A \neq x_1\}, \{g(A)\}, \mathcal{F}_9 \rangle$. Let

$$
\begin{aligned}
\Phi_4 = \{ & \langle \{A \neq x_1\}, \{g(A), h(A,A)\}, \mathcal{F}_2 \rangle, && [2] \\
& \langle \{A \neq x_1\}, \{g(A)\}, \mathcal{F}_0 \rangle, && [5] \\
& \langle \{A \neq x_1\}, \{g(A), h(A,x_1)\}, \mathcal{F}_3 \rangle && [6] \\
& \langle \emptyset, \{g(x_1), h(x_1,B)\}, \mathcal{F}_8 \rangle, && [8] \\
& \langle \{A \neq x_1\}, \{g(A)\}, \mathcal{F}_9 \rangle \}. && [9]
\end{aligned}
$$

We have $\mathcal{J}(\Phi_4) = \sum_{ground(h(A,B)):\{A \neq x_1, A \neq B, B \neq x_1\}} \mathcal{J}(\Phi_3)$.

If the parfactor is not in normal form, the exponent r may depend on values of remaining logical variables and it is necessary to compute all the sizes of the set $\mathcal{X}$ conditioned on values of logical variables remaining in the parfactor. In such case, the summing out operation creates multiple parfactors.

Example 5.6. Consider an alternative joint probability distribution over random variables present in the model from Figure 5.1 to the distribution presented in the Example 5.2. Assume we do not have any specific knowledge about ground instances of $g(A)$, but we have some specific knowledge about $h(A,B)$ for the case where $B = x_1$ and for the case where $A = B$. We would represent the model using

the following parfactors:

$$\begin{aligned}
\Phi_i = \{\, & \langle \emptyset, \{g(A)\}, \mathcal{F}_i \rangle, && [i] \\
& \langle \emptyset, \{g(A), h(A,x_1)\}, \mathcal{F}_{ii} \rangle, && [ii] \\
& \langle \{A \neq x_1\}, \{g(A), h(A,A)\}, \mathcal{F}_{iii} \rangle, && [iii] \\
& \langle \{A \neq B, B \neq x_1\}, \{g(A), h(A,B)\}, \mathcal{F}_{iv} \rangle \}, && [iv]
\end{aligned}$$

where $\mathcal{F}_i$ is a factor from $range(h)$ to the reals and $\mathcal{F}_1$, $\mathcal{F}_2$, and $\mathcal{F}_3$ are factors from $range(g) \times range(h)$ to the reals.

The sets of random variables represented by $g(A)$ in parfactors $[i]$ and $[ii]$ and parfactors $[iii]$ and $[iv]$ are not disjoint and are not identical. The sets of random variables represented by $h(A,x_1)$, $h(A,A)$ and $h(A,B)$ in parfactors from Φ_i are disjoint. Parfactor $[iv]$ is not in normal form, as $\mathcal{E}_A^{\{A \neq B, B \neq x_1\}} \setminus \{B\} = \emptyset \neq \{x_1\} = \mathcal{E}_B^{\{A \neq B, B \neq x_1\}} \setminus \{A\}$. Assume that we want to sum out $ground(h(A,B)) : \{A \neq B, B \neq x_1\}$ from Φ_i i.e., we want to sum out all ground instances of $h(A,B)$ that satisfy the constraints in $[iv]$. Conditions (S1) and (S2) of Proposition 2.1 are satisfied. The requirement that parfactor $[iv]$ is in normal-form is not satisfied. Nevertheless, lifted summation can be performed. The logical variable B will disappear from $[iv]$. We have

$$|\mathcal{D}(B) : \{A \neq B, B \neq x_1\}| = \begin{cases} n-1, & \text{if } A = x_1; \\ n-2, & \text{if } A \neq x_1. \end{cases} \tag{5.1}$$

Let $\mathcal{F}_v$ be a factor from $range(g)$ to the reals, $\mathcal{F}_v = \sum_{h(A,B)} \mathcal{F}_{iv}$. We obtain two parfactors as the result of summation: a parfactor $\langle \emptyset, \{g(x_1)\}, {\mathcal{F}_v}^{n-1} \rangle$ and a parfactor $\langle \{A \neq x_1\}, \{g(A)\}, {\mathcal{F}_v}^{n-2} \rangle$. Let

$$\begin{aligned}
\Phi_{ii} = \{\, & \langle \emptyset, \{g(A)\}, \mathcal{F}_i \rangle, && [i] \\
& \langle \emptyset, \{g(A), h(A,x_1)\}, \mathcal{F}_{ii} \rangle, && [ii] \\
& \langle \{A \neq x_1\}, \{g(A), h(A,A)\}, \mathcal{F}_{iii} \rangle, && [iii] \\
& \langle \emptyset, \{g(x_1)\}, {\mathcal{F}_v}^{n-1} \rangle, && [v] \\
& \langle \{A \neq x_1\}, \{g(A)\}, {\mathcal{F}_v}^{n-2} \rangle \}, && [vi]
\end{aligned}$$

We have $\mathcal{J}(\Phi_{ii}) = \sum_{ground(h(A,B)):\{A \neq B, B \neq x_1\}} \mathcal{J}(\Phi_i)$.

The above example shows that normal-form requirement is not necessary, but if it is not satisfied, the solution of #CSP encountered during summing out is conditioned values of logical variables remaining in the parfactor. In Section 5.4 we compare two approaches to solving #CSPs during lifted probabilistic inference: one based on a #CSP solver [de Salvo Braz et al., 2007; Poole, 2003], which can compute the expression (5.1), and the other normal-form based [Milch et al., 2008]. Similar observations could be made for a summing out a parameterized random variable from an aggregation parfactor (see Section 3.4.2.3).

The above overview of lifted inference, together with simple examples, shows that constraint processing is an integral, important part of lifted probabilistic inference. It also points out that there are two choices regarding constraint processing during lifted inference: a choice between splitting as needed and shattering and a choice between using a #CSP solver and enforcing normal form. These choices are independent, and shattering can be used with a #CSP solver [de Salvo Braz et al., 2007] or with normal form parfactors [Milch et al., 2008]. Splitting as needed can be used together with a #CSP solver [Poole, 2003], one could also imagine using splitting as needed but requiring normal-form for parfactors. The rest of this chapter is devoted to the comparison of these different strategies.

5.3 Splitting as needed vs. shattering

In this section we compare the splitting as needed approach and the shattering approach. For clarity, we do not analyze here splits related to converting constraints sets to normal form.

Shattering simplifies design and implementation of lifted inference procedures. Given a set of parfactors Φ, shattering performs all splits and expansions on substitutions that are obtained from unification and analysis of constraints (see Section 2.5.2.5). After shattering, all sets of random variables represented by parameterized random variables present in Φ are pairwise identical or disjoint. Lifted inference will not perform any additional splits or expansions, unless propositionalization or full expansion is performed.

Splitting as needed will not perform splits or expansions on other substitutions than those used during shattering (unless propositionalization or full expansion is

performed). In fact, it might perform much fewer splits and expansions as we demonstrate in the example below.

Example 5.7. Consider the following set of parfactors:

$$\begin{aligned}
\Phi = \{&\langle \emptyset, \{g_Q(), g_1(X_1, X_2, \ldots, X_k), g_2(X_2, X_3, \ldots, X_k), \ldots, g_k(X_k))\}, \mathcal{F}_0\rangle, && [0]\\
&\langle \emptyset, \{g_1(a, X_2, \ldots, X_k)\}, \mathcal{F}_1\rangle, && [1]\\
&\langle \emptyset, \{g_2(a, X_3, \ldots, X_k)\}, \mathcal{F}_2\rangle, && [2]\\
&\ldots,\\
&\langle \emptyset, \{g_{k-1}(a, X_k)\}, \mathcal{F}_{k-1}\rangle, && [k-1]\\
&\langle \emptyset, \{g_k(a)\}, \mathcal{F}_k\rangle, && [k]
\end{aligned}$$

and let $\mathbf{Q} = \mathit{ground}(g_Q())$. Assume we want to compute the marginal $\mathcal{J}_{\mathbf{Q}}(\Phi)$. Lifted inference with shattering will create exponentially more parfactors (in the number of logical variables in a parfactor) than lifted inference with splitting as needed.

For $i = 1, \ldots, k$, a set of random variables represented by a parameterized random variable $g_i(X_i, \ldots, X_k)$ in a parfactor $[0]$ is a proper superset of a set of random variables represented by a parameterized random variable $g_i(a, X_{i+1}, \ldots, X_k)$ in a parfactor $[i]$. Therefore lifted inference with shattering needs to perform several splits. Since the order of splits during shattering does not matter here, assume that the first operation is a split of the parfactor $[0]$ on a substitution $\{X_1/a\}$ which creates a parfactor

$$\langle \emptyset, \{g_Q(), g_1(a, X_2, \ldots, X_k), g_2(X_2, X_3, \ldots, X_k)\}, \mathcal{F}_0\rangle \qquad [k+1]$$

and a residual parfactor

$$\langle \{X_1 \neq a\}, \{g_Q(), g_1(X_1, X_2, \ldots, X_k), g_2(X_2, X_3, \ldots, X_k), \ldots, g_k(X_k))\}, \mathcal{F}_0\rangle\,. \quad [k+2]$$

In both newly created parfactors, for $i = 2, \ldots, k$, a set of random variables represented by a parameterized random variable $g_i(X_i, \ldots, X_k)$ is a proper superset of a set of random variables represented by a parameterized random variable $g_i(a, X_{i+1}, \ldots, X_k)$ in a parfactor $[i]$ and shattering proceeds with further splits of

both parfactors. Assume that in the next step parfactors $[k+1]$ and $[k+2]$ are split on a substitution $\{X_2/a\}$. The splits result in four new parfactors. The result of the split of the parfactor $[k+1]$ on $\{X_2/a\}$ contains a parameterized random variable $g_1(a,a,\ldots,X_k)$ and a parfactor $[1]$ needs to be split on a substitution $\{X_2/a\}$. The shattering process continues following a scheme described above. It terminates after $2^{k+1}-k-2$ splits and results in $2^{k+1}-1$ parfactors (each original parfactor $[i]$, $i=0,\ldots,k$, is shattered into 2^{k-i} parfactors). Assume that after initial shattering, lifted inference proceeds with the optimal elimination ordering $g_1,\ldots,g_k$ (this elimination ordering does not introduce counting formulas; whereas other orderings do). To compute the marginal $\mathcal{J}_{g_{\mathbf{Q}}()}(\Phi)$, 2^k lifted multiplications and $2^{k+1}-2$ lifted summations are performed.

Consider lifted inference with splitting as needed. Assume lifted inference follows an elimination ordering $g_1,\ldots,g_k$. A set of random variables represented by a parameterized random variable $g_1(X_1,\ldots,X_k)$ in a parfactor $[0]$ is a proper superset of a set of random variables represented by a parameterized random variable $g_1(a,X_2,\ldots,X_k)$ in a parfactor $[1]$ and the parfactor $[0]$ is split on a substitution $\{X_1/a\}$. The split results in parfactors identical to the parfactors $[k+1]$ and $[k+2]$ from the description of shattering above. The parfactor $[k+1]$ is multiplied by the parfactor $[1]$ and all ground instances of $g_1(a,X_1,\ldots,X_k)$ are summed out from their product while all ground instances of $g_1(X_1,X_2,\ldots,X_k)$ (subject to a constraint $X_1 \neq a$) are summed out from the parfactor $[k+2]$. The summations create two parfactors:

$$\langle \emptyset, \{g_Q(), g_2(X_2,X_3,\ldots,X_k),\ldots,g_k(X_k))\}, \mathcal{F}_{\mathcal{F}_{k+3}} \rangle, \qquad [k+3]$$
$$\langle \emptyset, \{g_Q(), g_2(X_2,X_3,\ldots,X_k),\ldots,g_k(X_k))\}, \mathcal{F}_{\mathcal{F}_{k+4}} \rangle. \qquad [k+4]$$

Ground instances of g_2 are eliminated next. Parfactors $[k+3]$ and $[k+4]$ are split on a substitution $\{X_2/a\}$, the results of the splits and a parfactor $[2]$ are multiplied together and the residual parfactors are multiplied together. Then, all ground instances of $g_2(a,X_3,\ldots,X_k)$ are summed out from the first product while all ground instances of $g_2(X_2,\ldots,X_k)$ (subject to a constraint $X_2 \neq a$) are summed out from the second product. The elimination of $g_3,\ldots,g_k$ looks the same as for g_2. In total,

$2k-1$ splits, $3k-2$ lifted multiplications and $2k$ lifted summations are performed. At any moment, the maximum number of parfactors is $k+3$.

The above example shows that shattering approach is sometimes much worse than splitting as needed.

Even though shattering creates a lot of additional parfactors, parfactors resulting from the same split share the same factor component. If factors are implemented as an immutable data structure, then they can be shared between different parfactors inside the implementation. This might help reduce overhead of shattering. We demonstrate this in the following example.

Example 5.8. Consider the following set of parfactors:

$$\begin{aligned}
\Phi = \{\, & \langle \emptyset, \{g_Q(), g_1(a)\}, \mathcal{F}_0 \rangle, && [0] \\
& \langle \{X \neq a\}, \{g_Q(), g_1(X)\}, \mathcal{F}_1 \rangle, && [1] \\
& \langle \emptyset, \{g_1(X), g_2(X)\}, \mathcal{F}_2 \rangle, && [2] \\
& \langle \emptyset, \{g_2(X), g_3(X)\}, \mathcal{F}_3 \rangle, && [3] \\
& \ldots, \\
& \langle \emptyset, \{g_{k-1}(X), g_k(X)\}, \mathcal{F}_k \rangle, && [k] \\
& \langle \emptyset, \{g_k(X)\}, \mathcal{F}_{k+1} \rangle \} && [k+1].
\end{aligned}$$

Assume that all functors have the range size 10, $\mathbf{Q} = \mathit{ground}(g_Q())$ and that we want to compute the marginal $\mathcal{J}_{\mathbf{Q}}(\Phi)$.

Because of the presence of parameterized random variable $g_1(a)$ in parfactor $[0]$, lifted inference with shattering splits parfactor $[2]$ on substitution $\{X/a\}$, which creates a parfactor $\langle \emptyset, \{g_1(a), g_2(a)\}, \mathcal{F}_2 \rangle$ and a residual parfactor. This in turn causes a split of parfactor $[3]$ on a substitution $\{X/a\}$. Splits propagate until the last split of parfactor $[k+1]$ on a substitution $\{X/a\}$, to a total of k splits, which create k additional parfactors. Afterward probabilistic inference proceeds with $2k+1$ multiplications and $k+1$ summations regardless of the elimination ordering.

Lifted inference with splitting as needed with elimination ordering $g_1, g_2, \ldots, g_k$ would perform exactly the same splits, multiplications, and summations as lifted

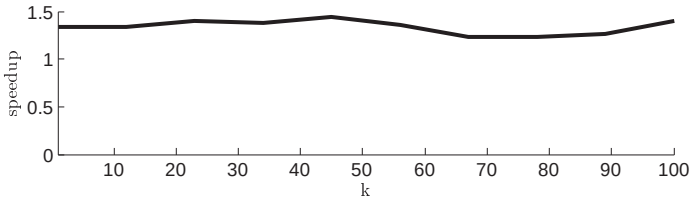


Figure 5.2: Speedup of splitting as needed over shattering for Example 5.8.

inference with shattering. Lifted inference with splitting as needed and ordering $g_k, g_{k-1}, \ldots, g_1$ performs just 1 split, $k+2$ multiplication,s and $k+1$ summations.

As we said in the introduction to this example, even though shattering creates a lot of additional parfactors, parfactors resulting from the same split share the same factor component, so overhead of shattering might be smaller than expected. We used a Java implementation of C-FOVE with factors implemented as an immutable data structure shared between different parfactors to verify this point. Tests were performed on an Intel Core 2 Duo 2.66GHz processor with 1GB of memory made available to the JVM. Figure 5.2 shows the results of the experiment where we varied k from 1 to 100. We report an average speedup of splitting as needed with elimination ordering $g_k, g_{k-1}, \ldots, g_1$ over shattering over 10 runs. Even though lifted inference with shattering created nearly twice as many parfactors, it used virtually the same amount of memory as lifted inference with splitting. It was slower because it performed more arithmetic operations, but it was not slower by a factor of two.

While minimizing the number of splits might result in faster inference, the size of factor components created during inference has a decisive influence on inference's speed. Consider the following example.

Example 5.9.

$$
\begin{aligned}
\Phi = \{\, & \langle \emptyset, \{g_Q(), g_1(X)\}, \mathcal{F}_0 \rangle, && [0] \\
& \langle \emptyset, \{g_1(X), g_2(X)\}, \mathcal{F}_1 \rangle, && [1] \\
& \ldots, \\
& \langle \emptyset, \{g_{k-2}(X), g_{k-1}(X)\}, \mathcal{F}_{k-2} \rangle, && [k-2] \\
& \langle \emptyset, \{g_{k-1}(X), g_k(X)\}, \mathcal{F}_{k-1} \rangle, && [k-1] \\
& \langle \emptyset, \{g_k(a)\}, \mathcal{F}_k \rangle, && [k] \\
& \langle \{X \neq a\}, \{g_k(X)\}, \mathcal{F}_{k+1} \rangle \} && [k+1]
\end{aligned}
$$

Assume that functors have the range size 10, $\mathbf{Q} = \mathit{ground}(g_Q())$ and that we want to compute the marginal $\mathcal{J}_{\mathbf{Q}}(\Phi)$.

Because of the presence of $g_k(a)$ in a parfactor $[k]$, lifted inference ordering with shattering splits a parfactor $[k-1]$ on a substitution $\{X/a\}$, which creates a parfactor $\langle \emptyset, \{g_{k-1}(a), g_k(a)\}, \mathcal{F}_{k-1} \rangle$ (and a residual parfactor). This in turn causes a split of a parfactor $[k-2]$ on a substitution $\{X/a\}$. Splits propagate until the last split of a parfactor $[0]$ on a substitution $\{X/a\}$, to a total of k splits, which create k additional parfactors. Afterward probabilistic inference proceeds with $2k$ multiplications and $2k$ summations regardless of the elimination ordering. The maximum size among factors created during inference is 100. Lifted inference with splitting as needed and elimination ordering $g_k, g_{k-1}, \ldots, g_1$ would achieve identical performance.

Lifted inference with splitting as needed and elimination ordering $g_1, g_2, \ldots, g_k$ first performs $k-1$ multiplications and summations and is left with three parfactors: $\langle \emptyset, \{g_Q(), g_k(X)\}, \mathcal{F} \rangle$ and parfactors $[k]$ and $[k+1]$. Then it splits the first parfactor on a substitution $\{X/a\}$. The result of a split is multiplied by the parfactor $[k]$ and the residual is multiplied by the parfactor $[k-1]$. Two additional summations eliminate ground instances of g_k from obtained products. The final result is obtained by multiplying together two resulting parfactors. Inference performed 1 split, $k+2$ multiplications and $k+1$ summations. The maximum size among factors created during inference is 1000.

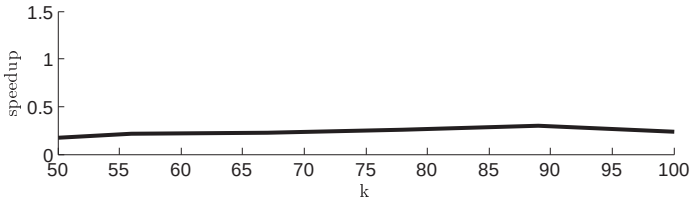


Figure 5.3: Splitting as needed vs. shattering speedup for Example 5.9.

We compared the shattering approach to splitting as needed with elimination ordering $g_1, g_2, \ldots, g_k$ under the same conditions as in Example 5.8. Figure 5.3 shows the results of the experiment where we varied k from 50 to 100. Even though lifted inference with the splitting as needed approach with elimination ordering $g_1, g_2, \ldots, g_k$ performs approximately only half of operations on parfactors compared to the shattering approach, it is slower because it manipulates much larger factors. Thus, it performed many more arithmetic operations. An elimination ordering heuristic should be able to detect that the above elimination ordering is undesired.

From the above examples we can see that with a good elimination ordering heuristic, the splitting as needed approach can be faster than the shattering approach. It is worth pointing out though, that splitting as needed complicates the design of an elimination ordering heuristic. With the shattering approach it is easy to compute the total size of parfactors that would be created by a potential inference step (see Section 2.5.2.6). With the splitting as needed approach, for each step under consideration we first need to determine the potentially necessary splits.

5.4 Normal form parfactors vs. #CSP solver

In Section 5.2 we pointed out, that during lifted probabilistic inference, we may encounter arbitrarily complicated #CSPs with inequality constraints because of the model's complexity, as well as the fact that the inference itself introduces many new inequality constraints. First-order probabilistic models usually involve logical variables with large populations, which means that we need to be able to solve #CSPs with large domains. Moreover, one run of a first-order probabilistic infer-

ence algorithm might involve solving many #CSPs, so an efficient algorithm for solving #CSP is required. We introduced such algorithm in Chapter 5.

Milch et al. [2008] avoid the need to use a constraint solver by requiring all parfactors to be in normal form (Section 2.5.1.1). Below we present two propositions which give insights into structure of constraints in normal-form parfactors.

Proposition 5.1. Let $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ be a parfactor in normal form. Then each connected component of the constraint graph corresponding to CSP formed by logical variables from $\mathcal{V}$ and constraints from $\mathcal{C}$ is fully connected.

Proof. The proposition is trivially true for components with one or two logical variables. Let us consider a connected component with more than two logical variables. Suppose, contrary to our claim, that there are two logical variables X and Y with no edge between them. Since the component is connected, there exists a path $X, Z_1, Z_2 ..., Z_m, Y$. As $\mathcal{C}$ is in normal form, $\mathcal{E}^{\mathcal{C}}_{Z_i} \setminus \{Z_{i+1}\} = \mathcal{E}^{\mathcal{C}}_{Z_{i+1}} \setminus \{Z_i\}$, $i = 1, \ldots, m-1$ and $\mathcal{E}^{\mathcal{C}}_{Z_m} \setminus \{Y\} = \mathcal{E}^{\mathcal{C}}_{Y} \setminus \{Z_m\}$. We have $X \in \mathcal{E}^{\mathcal{C}}_{Z_1}$, and consequently $X \in \mathcal{E}^{\mathcal{C}}_{Y}$. This contradicts our assumption that there is no edge between X and Y. □

Proposition 5.2. Let $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ be a parfactor in normal form. Let P be the CSP formed by logical variables from $\mathcal{V}$ and constraints from $\mathcal{C}$. Then for logical variables in the same connected component of the constraint graph corresponding to P sets of constants excluded by unary constraints are identical.

Proof. The proposition is trivially true for components with one logical variable. Let us consider a connected component with more than one logical variable. Suppose, contrary to our claim, that there are two logical variables X and Y and a constant c such that $c \in \mathcal{E}^{\mathcal{C}}_{X}$ and $c \notin \mathcal{E}^{\mathcal{C}}_{Y}$. Since the parfactor is in normal-form from Proposition 5.1 we know that the component is fully connected and $(X \neq Y) \in \mathcal{C}$. Therefore, because the parfactor is in normal-form, we have $\mathcal{E}^{\mathcal{C}}_{X} \setminus \{Y\} = \mathcal{E}^{\mathcal{C}}_{Y} \setminus \{X\}$. This contradicts our assumption that $c \in \mathcal{E}^{\mathcal{C}}_{X}$ and $c \notin \mathcal{E}^{\mathcal{C}}_{Y}$. □

The following corollary is a direct consequence of Propositions 5.1 and 5.2 and the definition of a normal-form parfactor from Section 2.5.1.1.

Corollary 5.3. A parfactor is in normal form $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ if and only if each connected component of the constraint graph corresponding to CSP formed by logical

variables from $\mathcal{V}$ and constraints from $\mathcal{C}$ is fully connected and each logical variable within the same connected component is subject to the same unary constraints.

The above characteristic of normal-form parfactors is the basis for a straightforward computation of the number of factors represented by a normal-form parfactor. We describe it in the following proposition.

Proposition 5.4. Let $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$ be a normal-form parfactor. Assume that a constraint graph corresponding to CSP formed by logical variables present in $\mathcal{V}$ and constraints from $\mathcal{C}$ consist of one connected component. Let m be the number of logical variables present in $\mathcal{V}$, n be the size of population of these logical variables, and l be the number of different constants present in $\mathcal{C}$, that is $l = |\{c : \exists X \in \mathcal{V}\ (X \neq c) \in \mathcal{C}\}|$. Then

$$|\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle| = \frac{(n-l)!}{(n-l-m)!}.$$

Proof. From the assumption that the underlying constraint graph consists of a single connected component we know that the graph has m nodes, one node for each logical variable present in $\mathcal{V}$. Since the parfactor is in normal-form, by Proposition 5.2 for each logical variable the set of constants excluded by unary constraints in $\mathcal{C}$ is identical and has size l. From Proposition 5.1 we know that the constraint graph is fully connected. Let us traverse the constraint graph in an arbitrary order and construct all possible ground substitutions to all logical variables in $\mathcal{V}$ that satisfy the constraints in $\mathcal{C}$. The first logical variable can have $n-l$ values, the second one can have $n-l-1$ values, and the process continues until we reach the last, m-th, logical variable, which can have $n-l-m+1$ values. Therefore the number of ground substitutions is as follows:

$$(n-l)(n-l-1)\ldots(n-l-m+1) = \frac{(n-l)!}{(n-l-m)!}.$$

□

The above proposition can be easily generalized to the case where there are multiple connected components.

While Proposition 5.4 shows that for normal-form parfactors solving #CSP is straightforward, the normal form has also negative consequences. The requirement

```
[00]  procedure convert(⟨C,V,F⟩)
[01]    input: parfactor ⟨C,V,F⟩;
[02]    output: set of normal-form parfactors Ψ created from ⟨C,V,F⟩ through splitting;
[03]    add ⟨C,V,F⟩ to the queue;
[04]    set Ψ := {};
[05]    while queue not empty do
[06]      remove parfactor g = ⟨C_i,V_i,F_i⟩ from the queue;
[07]      for each logical variable X from C_i do
[08]        for each logical variable Y from the same component of C_i as X do
[09]          if (X ≠ Y) ∉ C_i
[10]            split g on {X/Y};
[11]            add the result to the queue;
[12]            set g to the residual;
[13]          for each unary constraint Y ≠ t from C_i do
[14]            if (X ≠ t) ∉ C_i
[15]              split g on {X/t};
[16]              add the result to the queue;
[17]              set g to the residual;
[18]          end
[19]        end
[20]      end
[21]      add g to Ψ;
[22]    end
[23]    return Ψ;
[24]  end
```

Figure 5.4: Algorithm for converting a parfactor to a set of normal-form parfactors.

that all parfactors be in normal from is enforced by splitting parfactors that are not in normal form on appropriate substitutions. Milch et al. [2008], who introduced the concept of normal-form parfactors, do not provide any details how to do this conversion. Nevertheless, Corollary 5.3 provides us with a clear recipe for a conversion algorithm: perform all the splits required to make each connected component of the associated constraint graph be fully connected and perform all the splits required to make logical variables in the same connected component be subject to the same unary constraints; repeat the process for all by-product parfactors created by the above splits. The algorithm is presented in Figure 5.4. It is clear, that a conversion of a single parfactor to normal form can create several parfactors. We illustrate this point with two examples provided below.

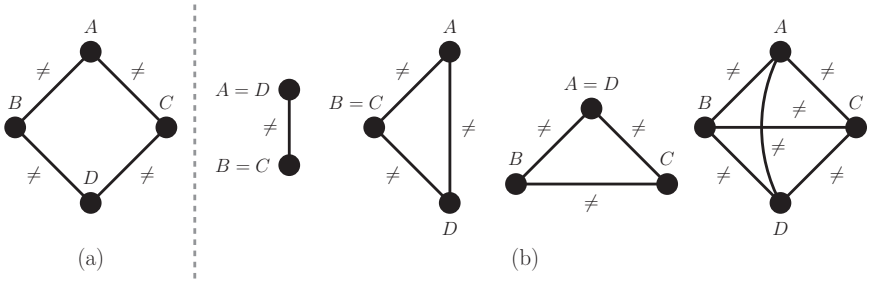


Figure 5.5: A simple constraint graph from Example 5.10 (a) and constraint graphs obtained through a conversion to normal form (b).

Example 5.10. Consider a parfactor $\langle \mathcal{C}, \mathcal{V}, \mathcal{F} \rangle$, where $\mathcal{C} = \{A \neq B, A \neq C, B \neq D, C \neq D\}$ and there are no other logical variables present in $\mathcal{V}$. The corresponding constraint graph is presented in Figure 5.5 (a). The parfactor is not in normal form, for example $\mathcal{E}_A^{\mathcal{C}} \setminus \{B\} \neq \mathcal{E}_B^{\mathcal{C}} \setminus \{A\}$. To convert the parfactor to a set of parfactors in normal form we need to perform splits on substitutions $\{A/D\}$ and $\{B/C\}$ in arbitrary order. After we split on the first substitution, we split both resulting parfactors on the second substitution. Thus we perform three splits and obtain four parfactors. Constraint graphs corresponding to these parfactors are presented Figure 5.5 (b).

Note that it is the same constraint graph as the one discussed in Example 4.4, where we needed to consider only two cases to get the number of solutions.

If the underlying graph is sparse, conversion might be very expensive as we show in the example below.

Example 5.11. Consider a parfactor

$$\langle \{X_0 \neq a, X_0 \neq X_1, \ldots, X_0 \neq X_k\}, \{g_0(X_0), g_1(X_1), \ldots, g_n(X_k)\}, \mathcal{F} \rangle,$$

where $\mathcal{D}(X_0) = \mathcal{D}(X_1) = \cdots = \mathcal{D}(X_k)$. Let $\mathcal{C}$ denote a set of constraints from this parfactor. We have $\mathcal{E}_{X_0}^{\mathcal{C}} = \{a, X_1, \ldots, X_k\}$ and $\mathcal{E}_{X_i}^{\mathcal{C}} = \{X_0\}$, $i = 1, \ldots, k$. The parfactor is not in normal form because $\mathcal{E}_{X_0}^{\mathcal{C}} \setminus \{X_i\} \neq \mathcal{E}_{X_i}^{\mathcal{C}} \setminus \{X_0\}$, $i = 1, \ldots, k$. As a result the size of the set $\mathcal{D}(X_0) : \mathcal{C}$ depends on other logical variables in the parfactor. For instance, it differs for $X_1 = a$ and $X_1 \neq a$ or for $X_1 = X_2$ and $X_1 \neq X_2$. A conversion of the considered parfactor to set of parfactors in normal form involves $2^k - 1$ splits on substitutions of the form $\{X_i/a\}$, $1 \leq i \leq k$ and $\sum_{i=2}^{k} \binom{k}{i} (\varpi_i - 1)$ splits on

substitutions of the form $\{X_i/X_j\}$, $1 \leq i,j \leq k$, where $\varpi_i - 1$ is i-th Bell number (see Section 4.2.3). The conversion creates $\sum_{i=0}^{k} \binom{k}{i} \varpi_i$ parfactors in normal form. In Example 5.13 we analyze how this conversion affects parfactor multiplication compared to the use of a #CSP solver.

From the above example we can see that the cost of converting a parfactor to normal form can be worse than exponential. Moreover, converting parfactors to normal form may be very inefficient when analyzed in context of parfactor multiplication (see Section 5.4.1) or summing out a parameterized random variable from a parfactor (see Section 5.4.2). Our empirical tests (see Section 5.4.3) confirm this observation.

While parfactors that involve counting formulas must be in normal form (see Section 2.5.1.1), this is not necessary for parfactors without counting formulas. An alternative to converting all parfactors to normal form is to use a #CSP solver. We presented an algorithm for solving #CSPs encountered during lifted probabilistic inference in Chapter 4. Details of the interaction between the solver and a probabilistic inference engine depend on the implementation of the solver, in particular on data structures used. In Appendix C we explain how to represent #CSPs encountered during lifted probabilistic inference using these data structures. In Appendix D we explain how to translate answers from the #CSP solver to data structures used by a probabilistic inference engine. In the example below we present a simple interaction with the solver.

Example 5.12. In Example 5.6 we need to know the number $|\mathcal{D}(B) : \{A \neq B, B \neq x_1\}|$, where $\mathcal{D}(A) = \mathcal{D}(B)$ and $|\mathcal{D}(A)| = n$. Let s-constant $\mathbf{a}_1$ denote set $\{x_1\}$ and s-constant $\mathbf{a}_2$ denote set $\mathcal{D}(A) \setminus \{x_1\}$ (s-constants were introduced in Section 4.3.2.1). The following $\#\mathrm{VE}_{\neq}$ factor has value 1 for substitutions to logical variables A,B that are solutions to the above CSP and 0 otherwise:

A	B	Partition(s)	
$\mathbf{a}_1$	$\mathbf{a}_2$	$\{\{A\}\}\ \{\{B\}\}$	1
$\mathbf{a}_2$	$\mathbf{a}_2$	$\{\{A,B\}\}$	0
$\mathbf{a}_2$	$\mathbf{a}_2$	$\{\{A\},\{B\}\}$	1

.

The #CSP solver eliminates B from the above factor and returns:

A	Partition(s)	
$\mathbf{a}_1$	$\{\{A\}\}$	$n-1$
$\mathbf{a}_2$	$\{\{A\}\}$	$n-2$

.

The two cases used in Example 5.6 are obtained through analysis of the resulting factor and knowledge that $\mathbf{a}_1$ denotes set $\{x_1\}$ and $\mathbf{a}_2$ denotes set $\mathcal{D}(A)\setminus\{x_1\}$. We can infer that $|\mathcal{D}(B) : \{A \neq B, B \neq x_1\}|$ equals $n-1$ if $A = x_1$ and $n-2$ if $A \neq x_1$.

5.4.1 Multiplication

In the example below we demonstrate how the normal form requirement might lead to a lot of, otherwise unnecessary, parfactor multiplications.

Example 5.13. Assume we would like to multiply the parfactor from Example 5.11 by a parfactor $pf = \langle \emptyset, \{g_1(X_1)\}, \mathcal{F}_1 \rangle$. First, let us consider how it is done with a #CSP solver. A #CSP solver computes the number of factors the parfactor from Example 5.11 represents, $(|\mathcal{D}(X_0)|-1)^{k+1}$. Next the solver computes the number of factors represented by the parfactor pf, which is trivially $|\mathcal{D}(X_1)|$. A correction is applied to values of the factor $\mathcal{F}_1$ to compensate for the difference between these two numbers. Finally the two parfactors are multiplied. The whole operation involved two calls to a #CSP solver, one correction and one parfactor multiplication. Now, let us see how it can be done without the use of #CSP solver. The first parfactor is converted to a set Φ of $\sum_{i=0}^{k} \binom{k}{i} \varpi_i$ parfactors in normal form, as presented in Example 5.11. Some of the parfactors in Φ contain a parameterized random variable $g_1(a)$, the rest contains a parameterized random variable $g_1(X)$ and a constraint $X_1 \neq a$, so the parfactor pf needs to be split on a substitution $\{X_1/a\}$. The split results in a parfactor $\langle \emptyset, \{g_1(a)\}, \mathcal{F}_1 \rangle$ and a residual $\langle \{X_1 \neq a\}, \{g_1(X_1)\}, \mathcal{F}_1 \rangle$. Next, each parfactor from Φ is multiplied by either the result of the split or the residual. Thus $\sum_{i=0}^{k} \binom{k}{i} \varpi_i$ parfactor multiplications need to be performed and most of these multiplication require a correction prior to the actual parfactor multiplication.

There is an opportunity for some optimization, as factor components of parfactors multiplications for different corrections could be cached and reused instead of being recomputed. Still, even with such a caching mechanism, multiple parfactor

multiplications would be performed compared to just one multiplication when a #CSP solver is used.

5.4.2 Summing out

Examples 5.6 and 5.12 demonstrate how summing out a parameterized variable from a parfactor that is not in normal form can be done with a help of a #CSP solver. In the example below we show how this operation would look if we convert the parfactor to a set of parfactors in normal form (which does not require a #CSP solver).

Example 5.14. Assume that we want to sum out $ground(h(A,B)) : \{A \neq B, B \neq x_1\}$ from the parfactor $\langle \{A \neq B, B \neq x_1\}, \{g(A), h(A,B)\}, \mathcal{F}_{iv} \rangle$ from the Example 5.6. First, we convert the parfactor to a set of parfactors in normal form by splitting on a substitutions $\{A/x_1\}$. We obtain two parfactors in normal form:

$$\langle \{B \neq x_1\}, \{g(x_1), h(x_1,B)\}, \mathcal{F}_{iv} \rangle,$$

which represents $n-1$ factors, and

$$\langle \{A \neq x_1, A \neq B, B \neq x_1\}, \{g(A), h(A,B)\}, \mathcal{F}_{iv} \rangle,$$

which represents $(n-1)(n-2)$ factors. Next we sum out $ground(h(x_1,B)) : \{B \neq x_1\}$ from the first parfactor and $ground(h(A,B)) : \{A \neq x_1, A \neq B, B \neq x_1\}$ from the second parfactor. In both cases a correction will be necessary, as B will no longer be among logical variables present in the resulting parfactors and these parfactors will represent fewer factors than the original parfactors.

In general, as illustrated by Examples 5.6, 5.12 and 5.14, a lifted inference procedure that enforces conversion to normal form and a lifted inference that uses a #CSP solver create the same number of parfactors. The difference is, that the first approach computes a factor component for the resulting parfactors once and then applies a different correction for each resulting parfactor based on the answer from the #CSP solver. The second approach computes the factor component multiple times, once for each resulting parfactor, but does not use a #CSP solver. As these factor components (before applying a correction) are identical, redundant

computations could be eliminated by caching. We successfully adopted a caching mechanism in our empirical test (Section 5.4.3), but expect it to be less effective for larger problems.

As in the case of splitting as needed, it might be difficult to design an efficient elimination ordering heuristic that would work with a #CSP solver. This is because we do not known in advance how many parfactors will be obtained as a result of summing out. We need to run a #CSP solver to obtain this information.

5.4.3 Experiment

In this experiment we summed out a parameterized random variable from a parfactor. We compared summing out with a help of a #CSP solver presented in Chapter 4 (#CSP-SUM) to summing out achieved by converting a parfactor to a set of parfactors in normal form and summing out a parameterized random variable from each obtained parfactor without a #CSP solver. (We cached factor components as suggested in Section 5.4.2).

We randomly generated sets of parfactors. There were up to 5 parameterized random variables in each parfactor with range sizes varying from 2 to 10. There were up to 10 logical variables present in each parfactor. Logical variables were typed with the same population. We varied the size of this population from 5 to 1000 to verify how well #CSP solver scaled for larger populations. The probability of presence of a binary constraint between a logical variable and another logical variable varied from 0.05 to 0.25 and each logical variable on average participated in 3 unary constraints. The above settings resulted in simple parfactors being generated. For each population size we generated 100 parfactors and reported a cumulative time.

We used Java implementations of tested lifted inference methods. Tests were performed on an Intel Core 2 Duo 2.66GHz processor with 1GB of memory made available to the JVM. The results are presented in Figure 5.6. For small population sizes, summing out with the help of a #CSP solver (#CSP-SUM) performs worse than summing out by converting to normal form (CONV-NFM-SUM). #CSP-SUM prevails for larger populations. For the second approach, we also report time excluding (NFM-SUM) conversion to normal form. The difference between CONV-

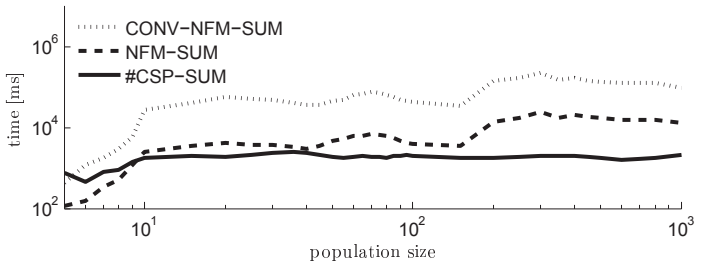


Figure 5.6: Summing out with and without a #CSP solver.

NFS-SUM and NFM-SUM shows the significant cost of conversion to normal form.

5.5 Conclusions

In this chapter we analyzed the impact of constraint processing on the efficiency of lifted inference and explained why we cannot ignore its role in lifted inference. We showed that a choice of constraint processing strategy has a big impact on efficiency of lifted inference. In particular, we discovered that shattering [de Salvo Braz et al., 2007] is never better—and sometimes worse—than splitting as needed [Poole, 2003], and that the conversion of parfactors to normal form [Milch et al., 2008] is an expensive alternative to using a specialized #CSP solver. Although in this chapter we focused on exact lifted inference, our results are potentially applicable to the approximate lifted inference algorithms which use parfactors, as even approximate methods might require the exact number of factors represented by a parfactor.

Chapter 6

Conclusions

Prediction is very difficult, especially about the future. — Niels Bohr

6.1 Summary

This thesis concerns exact lifted probabilistic inference in first-order probabilistic models. In particular, we focused on two problems: lifted aggregation in directed first-order probabilistic models (Chapter 3) and constraint processing during lifted probabilistic inference (Chapters 4 and 5).

Aggregation occurs naturally in non-trivial directed first-order models. Lifted inference algorithms focused on undirected first-order models and lacked data structures suitable for representing aggregation. We introduced a new data structure, aggregation parfactors, and defined operations on aggregation parfactors that allow them to be part of lifted inference algorithms. We demonstrated usefulness and effectiveness of our solution using a model from the social networks domain.

First-order probabilistic models involve inequality constraints on logical variables. Additional inequality constraints are introduced during inference. Constraints allow models to capture properties of particular individuals in a modeled domain. Lifted probabilistic inference has to process these constraints. One task is to count solutions to constraint satisfaction problems induced during inference. These CSPs might involve logical variables with large domains, but constraints are restricted to unary and binary inequality constraints. In Chapter 4 we presented

a lifted algorithm for counting solutions to such CSPs. The computational complexity of our algorithm does not directly depend on the domain sizes of logical variables. Because our algorithm is lifted, it inter-operates with a lifted probabilistic inference engine in a natural and efficient manner.

To date, constraint processing in lifted probabilistic inference has not received much attention in the literature and various researchers made seemingly arbitrary decisions about constraint processing strategies. In Chapter 5 we compared different approaches to constraint processing during lifted probabilistic inference. Our theoretical and empirical results stress the great importance of informed constraint processing.

6.2 Future work

There are many opportunities for future research in the area of lifted probabilistic inference. Below we outline open problems directly related to this thesis.

While it is difficult to design an elimination ordering heuristic that works well with the splitting as needed approach and a #CSP solver, it is worth pursuing.

In this thesis we used parfactors to describe first-order probabilistic models. Parfactors are data structures and hence not best suited for specifying models by humans. Moreover, sets of parfactors cannot directly represent all models that can be defined using first-order probabilistic modeling languages. For example, only simple ICL programs can be directly translated to sets of parfactors. Nevertheless, an inference procedure for ICL, could use the C-FOVE algorithm as a subroutine. Development of such procedure is an interesting open problem.

The framework presented in this thesis allows us only to reason about particular individuals from the population of a logical variable and about the rest of this population. A natural extension is to allow reasoning about sets of individuals belonging to some population. The calculus of parfactors would not be greatly affected by such generalization. The most important changes would be required in the unification process. Some changes would also be necessary in constraint processing, but not in the #CSP solver presented in Chapter 4.

The representational power of parfactors could be increased by allowing parfactors to contain other types of constraints, not just inequality constraints. For ex-

ample, 'less-than' constraint could be useful when used together with logical variables that naturally exhibit linear order, like space dimensions and time. Increased expressiveness comes at the higher cost of constraint processing. Balancing this tradeoff is a challenging task.

Finally, variable elimination-based lifted inference algorithms require lifted data structures for storing the intermediate results of a computation. If an appropriate lifted data structure is not available, such algorithms use propositionalization and store the results in factors. We believe that a search-based lifted inference algorithm could avoid unnecessary propositionalization.

Bibliography

D. Achlioptas, L. M. Kirousis, E. Kranakis, D. Krizanc, M. S. O. Molloy, and Y. C. Stamatiou [1997]. Random constraint satisfaction: A more accurate picture. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP 1997)*, 107–120. → pp. 139

O. Angelsmark and P. Jonsson [2003]. Improved algorithms for counting solutions in constraint satisfaction problems. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP 2003)*, 81–95. → pp. 107

O. Angelsmark, P. Jonsson, S. Linusson, and J. Thapper [2002]. Determining the number of solutions to binary CSP instances. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, 327–347. → pp. 107

O. Angelsmark and J. Thapper [2006]. Partitioning based algorithms for some colouring problem. In *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2005, Uppsala, Sweden, June 20-22, 2005, Revised Selected and Invited Papers*, volume 3978 of *Lecture Notes in Computer Science*, 44–58. Springer. → pp. 108, 143

K. R. Apt and M. Bezem [1991]. Acyclic programs. *New Generation Computing*, 9(3–4):335–363. → pp. 17

S. Arnborg, D. G. Corneil, and A. Proskurowski [1987]. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284. → pp. 12

R. J. Bayardo Jr. and J. D. Pehoushek [2000]. Counting models using connected components. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000)*, 157–162. → pp. 106

E. T. Bell [1934a]. Exponential numbers. *The American Mathematical Monthly*, 41(7):411–419. → pp. 111

E. T. Bell [1934b]. Exponential polynomials. *The Annals of Mathematics, 2nd Ser.*, 35(2):258–277. → pp. 111

E. T. Bell [1938]. The iterated exponential integers. *The Annals of Mathematics, 2nd Ser.*, 39(3):539–557. → pp. 111

U. Bertelè and F. Brioschi [1972]. *Nonserial Dynamic Programming*, volume 91 of *Mathematics in Science and Engineering*. Academic Press. → pp. 9

N. Biggs [1993]. *Algebraic Graph Theory*. Cambridge University Press, 2nd edition. → pp. 107, 108

E. Birnbaum and E. L. Lozinskii [1999]. The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10:457–477. → pp. 106

A. Björklund, T. Husfeldt, and M. Koivisto [2009]. Set partitioning via inclusion-exclusion. *SIAM Journal on Computing*, 39(2):546–563. → pp. 107, 108, 143

J. S. Breese [1992]. Construction of belief and decision networks. *Computational Intelligence*, 8(4):624–647. → pp. 2, 12, 21

A. A. Bulatov and V. Dalmau [2003]. Towards a dichotomy theorem for the counting constraint satisfaction problem. In *Proceedings of 44rd IEEE Symposium on Foundations of Computer Science (FOCS'03)*, 562–571. → pp. 106

W. L. Buntine [1994]. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225. → pp. 20

P. Carbonetto, J. Kisyński, M. Chiang, and D. Poole [2009]. Learning a contingently acyclic, probabilistic relational model of a social network. Technical Report TR-2009-08, The University of British Columbia, Department of Computer Science. → pp. 101, 103

M. Chavira, A. Darwiche, and M. Jaeger [2006]. Compiling relational Bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42(1–2):4–20. → pp. 21

V. Dahllöf, P. Jonsson, and M. Wahlström [2002]. Counting satisfying assignments in 2-SAT and 3-SAT. In *Proceedings of the 8th Annual International Computing and Combinatorics Conference (COCOON 2002)*, 535–543. → pp. 106, 107

V. Dahllöf, P. Jonsson, and M. Wahlström [2005]. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1-3):265 – 291. → pp. 106, 107

A. Darwiche [2009]. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press. → pp. 54

L. De Raedt, P. Frasconi, K. Kersting, and S. H. Muggleton, eds. [2008]. *Probabilistic Inductive Logic Programming*, volume 4911 of *Lecture Notes in Computer Science*. Springer. → pp. 2, 55

R. de Salvo Braz, E. Amir, and D. Roth [2005]. Lifted first-order probabilistic inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 1319–1125. → pp. 3, 21

R. de Salvo Braz, E. Amir, and D. Roth [2006]. MPE and partial inversion in lifted probabilistic variable elimination. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 2006)*, 1124–1130. → pp. 3, 21

R. de Salvo Braz, E. Amir, and D. Roth [2007]. Lifted first-order probabilistic inference. In L. Getoor and B. Taskar, eds., *Introduction to Statistical Relational Learning*, chapter 15, 433–450. MIT Press. → pp. ii, 3, 4, 15, 21, 22, 25, 31, 38, 66, 144, 148, 153, 168

R. de Salvo Braz, S. Natarajan, H. Bui, J. Shavlik, and S. Russell [2009]. Anytime lifted belief propagation. In *Proceedings of the International Workshop on Statistical Relational Learning (SRL 2009)*. → pp. 55

R. Dechter [1999]. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1):41–85. → pp. 9

R. Dechter [2003]. *Constraint Processing*. Morgan Kaufmann Publishers. → pp. ix, 106, 107, 108, 109, 110, 138

R. Dechter, K. Kask, and R. Mateescu [2002]. Iterative join-graph propagation. In *Proceedings of the 18th Annual Conference on Uncertainty in Artificial Intelligence (UAI 2002)*, 128–136. → pp. 107

R. Dechter and J. Pearl [1987]. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1):1–38. → pp. 107

F. J. Díez [1993]. Parameter adjustment in Bayes networks. The generalized noisy OR-gate. In *Proceedings of the 9th Annual Conference on Uncertainty in Artificial Intelligence (UAI 1993)*, 99–105. → pp. 59

F. J. Díez and S. F. Galán [2003]. Efficient computation for the noisy MAX. *International Journal of Intelligent Systems*, 18(2):165–177. → pp. 69, 70, 98

O. Dubois [1991]. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81(1):49–64. → pp. 106

L. Getoor and B. Taskar, eds. [2007]. *Introduction to Statistical Relational Learning*. Adaptive Computation and Machine Learning. MIT Press. → pp. 2, 55

J. Gørtz [2000]. Java tip 92: Use the JVM profiler interface for accurate timing. http://www.javaworld.com/javaworld/javatips/jw-javatip92.html. → pp. 138

R. Gupta, A. A. Diwan, and S. Sarawagi [2007]. Efficient inference with cardinality-based clique potentials. In *Proceedings of the 24th Annual International Conference on Machine Learning (ICML 2007)*, 329–336. → pp. 15

M. Horsch and D. Poole [1990]. A dynamic approach to probabilistic inference using Bayesian networks. In *Proceedings of the 6th Annual Conference on Uncertainty in AI (UAI 1990)*, 155–161. → pp. 2, 12

M. C. Horsch and W. S. Havens [2000]. Probabilistic arc consistency: A connection between constraint reasoning and probabilistic reasoning. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI 2000)*, 282–290. → pp. 107

M. Jaeger [2002]. Relational Bayesian networks: a survey. *Electronic Transactions in Artificial Intelligence*, 6. → pp. 60

K. Kask, R. Dechter, and V. Gogate [2004a]. Counting-based look-ahead schemes for constraint satisfaction. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, 317–331. → pp. 107

K. Kask, R. Dechter, and V. Gogate [2004b]. New look-ahead schemes for constraint satisfaction. In *The 8th International Symposium on Artificial Intelligence and Mathematics (AI&M 2004)*. → pp. 107

K. Kersting, B. Ahmadi, and S. Natarajan [2009]. Counting belief propagation. In *Proceedings of the 25th Annual Conference on Uncertainty in Artificial Intelligence (UAI 2009)*, 277–284. → pp. 55

J. Kisyński and D. Poole [2009a]. Constraint processing in lifted probabilistic inference. In *Proceedings of the 25th Annual Conference on Uncertainty in Artificial Intelligence (UAI 2009)*, 293–302. → pp. 5

J. Kisyński and D. Poole [2009b]. Lifted aggregation in directed first-order probabilistic models. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1922–1929. → pp. 5

U. Kjærulff [1990]. Triangulation of graphs - algorithms giving small total state space. Technical Report R90-09, Aalborg University, Department of Mathematics and Computer Science, Aalborg, Denmark. → pp. 12, 139

D. E. Knuth [2005]. *The Art of Computer Programming, Volume 4: Combinatorial Algorithms, Fascicle 3: Generating All Combinations and Partitions*. Addison-Wesley. → pp. 112, 184

D. Koller and N. Friedman [2009]. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive Computation and Machine Learning. MIT Press. → pp. 2, 54

D. Koller and A. Pfeffer [1997]. Object-oriented Bayesian networks. In *Proceedings of the 13th Annual Conference on Uncertainty in AI (UAI 1997)*, 302–313. → pp. 21

J. W. Lloyd [1987]. *Foundations of Logic Programming*. Springer, 2nd edition. → pp. 17

L. Lovász [2003]. *Combinatorial Problems and Exercises*. North-Holland, 2nd edition. → pp. 112

A. K. Mackworth [1977]. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118. → pp. 40

A. Meisels, S. E. Shimony, and G. Solotorevsky [2000]. Bayes networks for estimating the number of solutions to a CSP. *Annals of Mathematics and Artificial Intelligence*, 28(1-4):169–186. → pp. 106

B. Milch [2006]. *Probabilistic Models with Unknown Objects*. Ph.D. thesis, University of California, Berkeley, Computer Science Division. → pp. 55

B. Milch, L. S. Zettlemoyer, K. Kersting, M. Haimes, and L. P. Kaelbling [2008]. Lifted probabilistic inference with counting formulas. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*, 1062–1068. → pp. ii, 3, 4, 15, 21, 22, 25, 26, 31, 33, 35, 36, 37, 38, 46, 50, 66, 145, 146, 148, 153, 160, 162, 168

U. Montanari [1974]. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7(2):95–132. → pp. 106

C. H. Papadimitriou [1994]. *Computational complexity*. Addison Wesley. → pp. 106

J. Pearl [1986]. Fusion, propagation and structuring in belief networks. *Artificial Intelligence*, 29(3):241–288. → pp. 59

J. Pearl [1988]. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann. → pp. 6

M. Pearson and L. Michell [2000]. Smoke Rings: social network analysis of friendship groups, smoking and drug-taking. *Drugs: education, prevention and policy*, 7:21–37. → pp. 103

G. Pesant [2005]. Counting solutions of CSPs: A structural approach. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 260–265. → pp. 107

A. Pfeffer and D. Koller [2000]. Semantics and inference for recursive probability models. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000)*, 538–544. → pp. 21

Piṅgala [200 B.C.]. *Chandah-sûtra*. → pp. 88

D. Poole [1993]. Probabilistic Horn abduction and Bayesian networks. *Artificial Intelligence*, 64(1):81–129. → pp. 16

D. Poole [1997]. The Independent Choice Logic for modeling multiple agents under uncertainty. *Artificial Intelligence*, 94(1–2):7–56. → pp. 16

D. Poole [2000]. Abducting through negation as failure: Stable models with the Independent Choice Logic. *Journal of Logic Programming*, 44:5–35. → pp. 16, 18

D. Poole [2003]. First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, 985–991. → pp. ii, 2, 3, 4, 21, 22, 25, 38, 43, 144, 146, 148, 153, 168

D. Poole [2008]. The Independent Choice Logic and beyond. In L. De Raedt, P. Frasconi, K. Kersting, and S. H. Muggleton, eds., *Probabilistic Inductive Logic Programming*, volume 4911 of *Lecture Notes in Computer Science*, 222–243. Springer. → pp. 18, 101

D. Poole and A. Mackworth [2010]. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press. → pp. 1, 55

P. Refalo [2004]. Impact-based search strategies for constraint programming. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, 557–571. → pp. 107

D. Roth [1996]. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302. → pp. 106

S. J. Russell and P. Norvig [2009]. *Artificial Intelligence - A modern approach*. Prentice Hall, 3rd edition. → pp. 1, 55

P. Savicky and J. Vomlel [2007]. Exploiting tensor rank-one decomposition in probabilistic inference. *Kybernetika*, 43(5):747–764. → pp. 70

P. Sen, A. Deshpande, and L. Getoor [2009]. Bisimulation-based approximate lifted inference. In *Proceedings of the 25th Annual Conference on Uncertainty in Artificial Intelligence (UAI 2009)*, 496–505. → pp. 55

P. Singla and P. Domingos [2008]. Lifted first-order belief propagation. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, 1094–1099. → pp. 5, 55

L. Sterling and E. Shapiro [1994]. *The Art of Prolog*. The MIT Press, 2nd edition. → pp. 38, 42

N. Taghipour, W. Meert, J. Struyf, and H. Blockeel [2009]. First-order Bayes-Ball for CP-Logic. In *Proceedings of the International Workshop on Statistical Relational Learning (SRL 2009)*. → pp. 48

L. G. Valiant [1979]. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421. → pp. 106

W. A. Whitworth [1878]. *Choice and Chance: An Elementary Treatise on Permutations, Combinations, and Probability, with 300 Exercises (3rd, revised and enlarged edition)*. Cambridge: Deighton Bell. → pp. 112

N. L. Zhang and D. Poole [1994]. A simple approach to Bayesian network computations. In *Proceedings of the 10th Biennial Canadian Artificial Intelligence Conference (AI 1994)*, 171–178. → pp. 7, 9

N. L. Zhang and D. Poole [1996]. Exploiting causal independence in Bayesian network inference. *Journal of Artificial Intelligence Research*, 5:301–328. → pp. 58

W. Zhang [1996]. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155(1):277–288. → pp. 106

Appendix A

1-dimensional Representation of VE Factors

Appendix. Hmm. A-Anything else?
— Lt. Father Francis John Patrick Mulcahy (M*A*S*H)

In this appendix, we present a 1-dimensional representation of factors. It is based on David Poole's Java implementation of VE for belief networks. Besides saving memory when we want a dense representation, this representation allows for efficient implementation of the summing-out operator.

Let us consider factor f on variables $X_1, X_2, \ldots, X_n$, where each variable X_i has a domain $\{x_0^i, x_1^i, \ldots, x_{d_i-1}^i\}$ ordered according to some total ordering $<_i$. Assume we are given a total ordering $<_X$ of the variables. The ordering $<_X$ is arbitrary. This situation induces a lexicographic ordering of the tuples of elements from the domains of the variables so that factors can be implemented as 1-dimensional arrays.

Assume that we have $X_1 <_X X_2 <_X \cdots <_X X_n$ and $x_0^i <_i x_1^i <_i \cdots <_i x_{d_i-1}^i$, for $i = 1, 2, \ldots, n$. Ordering $<_T$ of the tuples in f is defined as follows:

$$(x_{i_1}^1, x_{i_2}^2, \ldots, x_{i_n}^n) <_T (x_{j_1}^1, x_{j_2}^2, \ldots, x_{j_n}^n) \iff$$
$$\exists_{l \in \{1,2,\ldots,n\}} \left((x_{i_l}^l <_l x_{j_l}^l) \wedge \forall_{k \in \{1,2,\ldots,l-1\}} (x_{i_k}^k =_k x_{j_k}^k) \right).$$

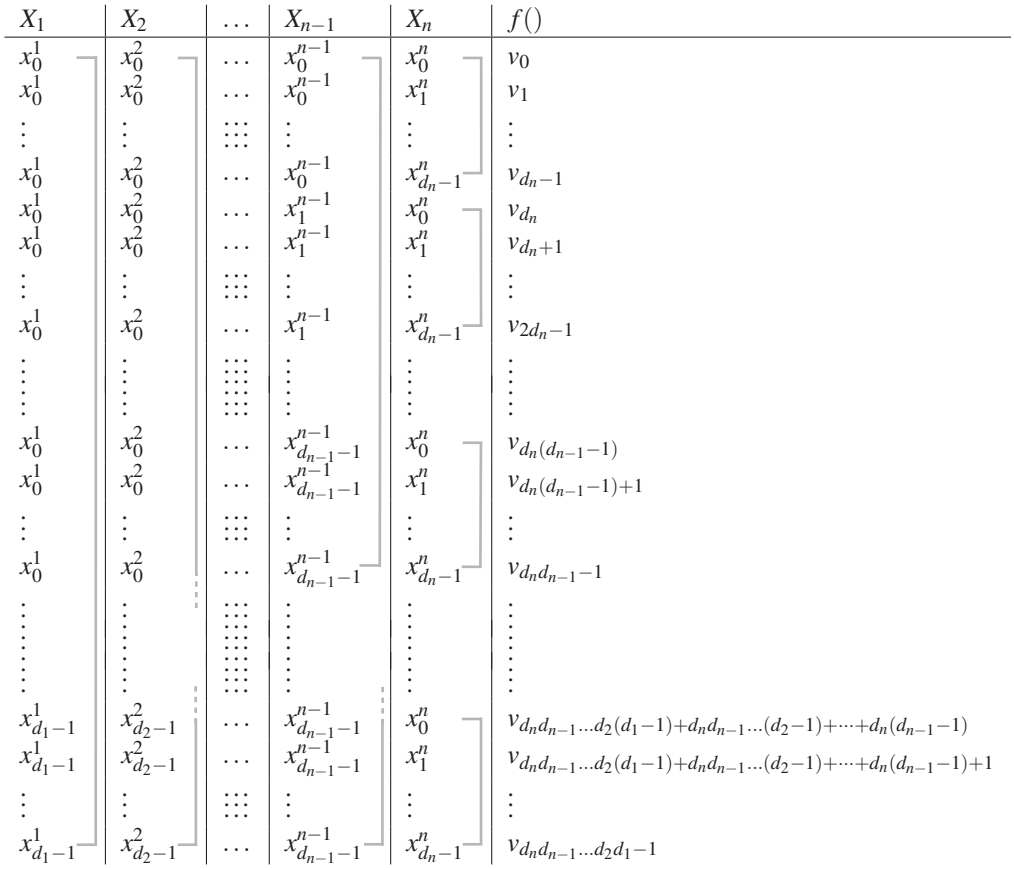


Figure A.1: Structure of a VE factor.

Figure A.1 shows the structure of factor f. The factor consists of $d_1d_2\ldots d_n$ tuples ordered according to $<_T$ with values $v_0, v_1, \ldots, v_{d_1d_2\ldots d_n}$. Consider tuple $(x_{i_1}^1, x_{i_2}^2, \ldots, x_{i_n}^n)$ from the factor f. Given indexes $i_1, i_2, \ldots, i_n$ we can compute the tuple's index (and thus, the index of its value) in f:

$$(x_{i_1}^1, x_{i_2}^2, \ldots, x_{i_n}^n) \rightarrow v_{((\ldots(i_1d_2+i_2)d_3+\ldots)d_{n-1}+i_{n-1})d_n+i_n}.$$

Given the index of the tuple (the index of its value) in f, we can recover the tuple's elements with the procedure indexToTuple(f,t) presented in Table A.2. Therefore, we do not need to store tuples in factors but only their values, which saves memory.

```
[00]  procedure indexToTuple(f,t)
[01]    input: factor f on variables X_1,X_2,...,X_n,
[02]           index t of the tuple;
[03]    output: tuple with index t in factor f;
[04]    for j := n downto 2 do
[05]      set i_j := t mod d_j;
[06]      set t := t div d_j;
[07]    end
[08]    set i_1 := t;
[09]    return (x^1_{i_1}, x^2_{i_2}, ..., x^n_{i_n});
[10]  end
```

Figure A.2: Procedure indexToTuple(f,t).

Appendix B

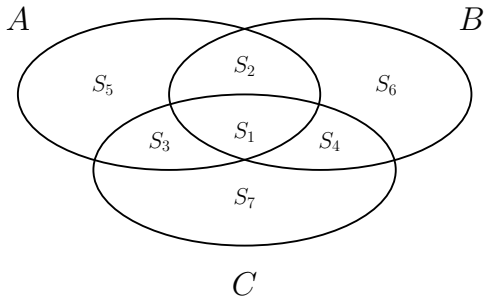
Hierarchical Representation of #VE$_{\neq}$ Factors

I'm afraid I have a bad appendix.
— Maj. Margaret 'Hot Lips' O'Houlihan (M*A*S*H)

In this appendix, we show how to implement the dense representation of #VE$_{\neq}$ factors as a hierarchy of 1-dimensional arrays, so that instead of storing both #VE$_{\neq}$ tuples and values in memory, we only need to store values.

For simplicity, we will describe our representation using a factor on three variables as an example. Consider Figure B.1, which illustrates domains of the three variables A, B and C. We need at most seven disjoint subsets, $S_1, S_2, \ldots, S_7$, to specify domains of the variables. In our example, we assume that all seven subsets are not empty; thus, the domain of each variable consists of four subsets. In the #VE$_{\neq}$ factor we represent subsets using seven s-constants, $c_1, c_2, \ldots, c_7$, as described in Section 4.3.2.2.

We represent a #VE$_{\neq}$ factor on A, B and C as a hierarchy of factors. The hierarchy consists of two levels. At the top level, there is a factor representing all combinations of s-constants describing the domain of each variable, and which maps them via pointers to factors from the bottom level of the hierarchy (see Figure B.2. If we introduce a total ordering among s-constants, then we can represent



$$A = S_1 \cup S_2 \cup S_3 \cup S_5$$

$$B = S_1 \cup S_2 \cup S_4 \cup S_6$$

$$C = S_1 \cup S_3 \cup S_4 \cup S_7$$

Figure B.1: Domains of three variables represented with disjoint subsets.

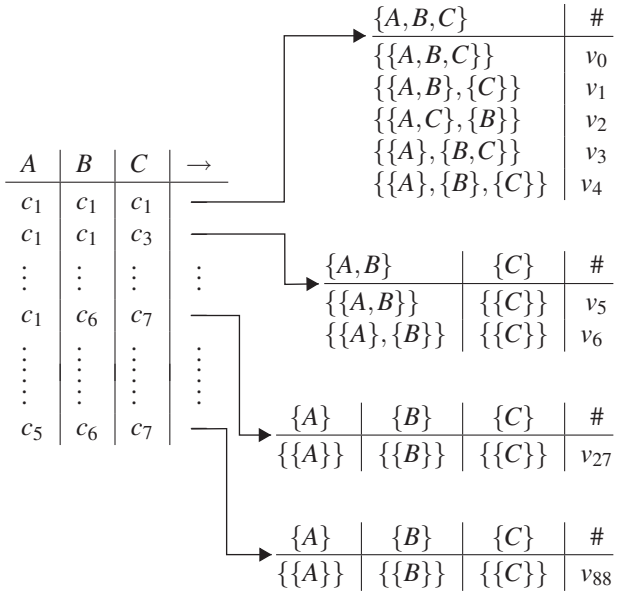


Figure B.2: Structure of a #VE$_{\neq}$ factor.

the factor from the top level as a 1-dimensional array, as described in Appendix A. The disadvantage of such dense representation is that we have to store all of the pointers in the array, even if they are *null*.

At the bottom level, there are factors mapping partitions of the variables A, B and C to the values of the #VE$_{\neq}$ factor. A combination of s-constants pointing to a factor from the bottom level determines the structure of the factor. For each subset of the variables with the same s-constant, we have some number of partitions.

For example, for the tuple of s-constants $\langle c1, c1, c2 \rangle$, we have two partitions of variables A and B (namely $\{\{A, B\}\}$ and $\{\{A\}, \{B\}\}$ and one partition of variable C ($\{\{C\}\}$). In the factor from the bottom level, we represent all of their combinations and map them to the appropriate values of the $\#\mathrm{VE}_{\neq}$ factor (see Figure B.2). If we use a standard lexicographic ordering of partitions (see [Knuth, 2005]), we can represent the factors from the bottom level as 1-dimensional arrays (as described in Appendix A) and a list of partitions for each subset of variables. The disadvantage of such dense representation is that we have to store values for all combinations of partitions in the array, even if they are 0 (which can happen sometimes after summing out).

The combinatorial properties of set partitions allow us to efficiently recover the partition structure from the position of the partition in the ordering, and to compute the position of the partition in the ordering given its structure. Therefore, we do not need to store lists of partitions associated with factors from the bottom level of the hierarchy, but only lists of their indexes in the ordering. Java methods that implement necessary operations on set partitions and Bell numbers are part of the Bell package[1]. As it is mainly a programming exercise, we do not describe those methods.

It is important to mention that the hierarchical, dense representation is not the only possibility. An empirical evaluation is necessary to decide whether it is better than a flat, sparse representation or a hierarchical, sparse representation. Similarly, one could use partitions themselves instead of indexes of set partitions.

[1] http://people.cs.ubc.ca/~kisynski/code/bell/

Appendix C

From Parfactors to #VE$_{\neq}$ Factors

Well, I figured I would go after the appendix while I'm in the area.
— Maj. Frank Marion 'Ferret Face' Burns (M*A*S*H)

In this appendix, we show how to represent a CSP induced by a parfactor in terms of #VE$_{\neq}$ factors. We use the dense representation of #VE$_{\neq}$ factors described in Appendix B.

For simplicity we will analyze a CSP consisting of a single connected component with three variables. Consider the following parfactor:

$$\langle \mathsf{C}, \{f(A,B,C), g(A,C), g(x_2,x_9)\}, \mathcal{F} \rangle, \qquad [0]$$

where $\mathsf{C} = \{A \neq x_2, B \neq x_3, B \neq x_7, C \neq x_2, A \neq B, B \neq C\}$, $\mathcal{D}(A) = \mathcal{D}(B) = \mathcal{D}(C)$ and $|\mathcal{D}(A)| = 10$.

Constraints in the parfactor form a CSP instance $\mathsf{P} = (\{A,B,C\}, \mathsf{D}, \mathsf{C})$, where $\mathsf{D}(A) = \mathsf{D}(B) = \mathsf{D}(C)$. We know that $\mathsf{D}(A)$ contains ten elements. We know only those elements of $\mathsf{D}(A)$, which are explicitly present in the parfactor: x_2, x_3, x_7, x_9.

Let us represent the instance P using #VE$_{\neq}$ factors. First, we exclude elements participating in unary constraints from the domains of the variables A, B, and C and obtain pruned domains $\widehat{\mathsf{D}}(A)$, $\widehat{\mathsf{D}}(B)$, and $\widehat{\mathsf{D}}(C)$. Then, we partition pruned domains into disjoint sets of elements, and represent these sets as s-constants (see

Section 4.3.2.1). There are three variables, so we will need at most $2^3 - 1 = 7$ s-constants (see Figure B.1).

In our implementation we create a sparse data structure that maps potential s-constants to the sets of elements they represent. We pick an arbitrary total ordering $<$ of $\mathsf{D}(A)$, sort unary constraints for each variable according to $<$, and process the elements known to us in order of $<$. This allows us to generate sets of indistinguishable individuals represented by s-constants through a single sweep of unary constraints and elements that are known to us. Each set of elements represented by an s-constant can be defined by listing all of its elements or by listing all elements from the domain that do not belong to it. For each set we choose a more compact representation.

For simplicity, let as assume that $x_2 < x_3 < x_7 < x_9$. Element x_2 is excluded from the domain of A and C, therefore it is represented by the s-constant S_6. Element x_3 is excluded from the domain of B, therefore it is represented by the s-constant S_3. Element x_7 is also represented by S_3. Element x_9 belongs to the domains of all variables and is represented by the s-constant S_1. The remaining six elements belong to the domains of all variables and are also represented by S_1. S_1 represents seven elements, S_3 represents two elements, and S_6 represents one element. We have $\widehat{\mathsf{D}}(A) = S_1 \cup S_3$, $\widehat{\mathsf{D}}(B) = S_1 \cup S_6$, and $\widehat{\mathsf{D}}(C) = S_1 \cup S_3$.

Although the #VE$_{\neq}$ algorithm only needs to know the sizes of the sets denoted by each s-constant, we need to know the elements of these sets so that we can interpret answers returned by the #VE$_{\neq}$ algorithm. The s-constant S_1 represents the set of elements $\{x \in \mathsf{D}(A) | A \neq x_2 \wedge A \neq x_3 \wedge A \neq x_7\}$, S_3 represents the set $\{x_3, x_7\}$ and S_6 represents the set $\{x_2\}$. The length of the descriptions of each of these sets is independent of the size of $\mathsf{D}(A)$.

Once we process unary constraints and obtain s-constants, we can use the procedure from Figure 4.10 to construct the following representation of the CSP instance:

A	B	Partition(s)	#
S_1	S_1	$\{\{A,B\}\}$	0
S_1	S_1	$\{\{A\},\{B\}\}$	1
S_1	S_6	$\{\{A\}\}$ $\{\{B\}\}$	1
S_3	S_1	$\{\{A\}\}$ $\{\{B\}\}$	1
S_3	S_6	$\{\{A\}\}$ $\{\{B\}\}$	1

B	C	Partition(s)	#
S_1	S_1	$\{\{B,C\}\}$	0
S_1	S_1	$\{\{B\},\{C\}\}$	1
S_1	S_3	$\{\{B\}\}$ $\{\{C\}\}$	1
S_6	S_1	$\{\{B\}\}$ $\{\{C\}\}$	1
S_6	S_3	$\{\{B\}\}$ $\{\{C\}\}$	1

The first #VE$_{\neq}$ factor represents the binary constraint $A \neq B$ and the second #VE$_{\neq}$ factor represents the binary constraint $B \neq C$.

If the domains of logical variables present in the parfactor were larger, we would perform exactly the same computation, only the size of the set represented by the s-constant S_1 would be larger.

Appendix D

From #VE$_{\neq}$ Factors to Parfactors

That's right up my alley, I wrote the book on the appendix. I even wrote the appendix, but they took that out.
— Capt. Benjamin Franklin 'Hawkeye' Pierce (M*A*S*H)

In this appendix, we show how an answer from the #VE$_{\neq}$ algorithm is processed by a lifted inference procedure.

Consider the following parfactor from Appendix C:

$$\langle \mathcal{C}, \{f(A,B,C), g(A,C), g(x_2,x_9)\}, \mathcal{F} \rangle, \qquad [0]$$

where $\mathcal{C} = \{A \neq x_2, B \neq x_3, B \neq x_7, C \neq x_2, A \neq B, B \neq C\}$, $\mathcal{D}(A) = \mathcal{D}(B) = \mathcal{D}(C)$ and $|\mathcal{D}(A)| = 10$.

Assume that a lifted inference procedure is about to sum out the parameterized random variable $f(A,B,C)$ from the parfactor [0]. The logical variable B appears only in $f(A,B,C)$ and will be eliminated from the parfactor. Therefore the resulting parfactor will represent fewer factors than the parfactor [0] and the lifted probabilistic inference procedure needs to compensate for this difference. To do so, it needs to compute the size of the set $\mathcal{D}(B) : \mathcal{C}$, which tells how many times fewer factors the resulting parfactor will represent. The parfactor [0] is not in normal form and it is necessary to compute all the sizes of the $\mathcal{D}(B) : \mathcal{C}$ set conditioned on values of logical variables remaining in the resulting parfactor, namely A and C. In our im-

plementation, the lifted inference procedure uses the #VE$_{\neq}$ algorithm to perform this task.

First, we represent the CSP instance formed by constraints in the parfactor [0]. This is described in Appendix C. Then, the #VE$_{\neq}$ algorithm multiplies the two #VE$_{\neq}$ factors that represent the CSP instance and sums out B from the product. The #VE$_{\neq}$ algorithm returns the following answer:

A	C	Partition(s)	#
S_1	S_1	$\{\{A,C\}\}$	7
S_1	S_1	$\{\{A\},\{C\}\}$	6
S_1	S_3	$\{\{A\}\}\ \{\{C\}\}$	7
S_3	S_1	$\{\{A\}\}\ \{\{C\}\}$	7
S_3	S_3	$\{\{A,C\}\}$	8
S_3	S_3	$\{\{A\},\{B\}\}$	8

We also know that the s-constant S_1 represents the set of elements $\{x \in \mathsf{D}(A) | A \neq x_2 \wedge A \neq x_3 \wedge A \neq x_7\}$ and the s-constant S_3 represents the set $\{x_3, x_7\}$ (see Appendix C).

The answer from the #VE$_{\neq}$ algorithm needs to be translated to sets of substitutions and constraints accompanying each #VE$_{\neq}$ tuple and its value. We start with computing the generic result of summation, parfactor [00], by summing out $f(A,B,C)$ from the parfactor [0] without compensating for disappearance of B:

$$\langle \{A \neq x_2, C \neq x_2\}, \{g(A,C), g(x_2,x_9)\}, \mathcal{F}_f \rangle, \qquad [00]$$

where $\mathcal{F}_f = \sum_{f(\ldots)} \mathcal{F}$

Next, for each #VE$_{\neq}$ tuple we generate one or more parfactors by modifying the parfactor [00]:

- $\langle S_1, S_1, [\{\{A,C\}\}] \rangle$ – A and C are equal, their domain is $\{x \in \mathsf{D}(A) | A \neq x_2 \wedge A \neq x_3 \wedge A \neq x_7\}$. We add the description of the domain to the parfactor [00], apply substitution $\{A/C\}$ to the parfactor [00], and bring the factor component to the power 7:

$$\langle \{C \neq x_2, C \neq x_3, C \neq x_7\}, \{g(C,C), g(x_2,x_9)\}, \mathcal{F}_f{}^7 \rangle; \qquad [01]$$

- $\langle S_1, S_1, [\{\{A\},\{C\}\}]\rangle$ – A and C are different, their domains are respectively $\{x \in \mathsf{D}(A) | A \neq x_2 \wedge A \neq x_3 \wedge A \neq x_7\}$ and $\{x \in \mathsf{D}(C) | C \neq x_2 \wedge C \neq x_3 \wedge C \neq x_7\}$. We add the descriptions of the domains to the parfactor [00], add constraint $A \neq C$, and bring the factor component to the power 6:

$$\langle \{A \neq x_2, A \neq x_3, A \neq x_7, C \neq x_2, C \neq x_3, C \neq x_7, A \neq C\}, \{g(A,C), g(x_2,x_9)\}, {\mathcal{F}_f}^6 \rangle; \quad [02]$$

- $\langle S_1, S_3, [\{\{A\}\}\ \{\{C\}\}]\rangle$ – A and C have disjoint domains, their domains are respectively $\{x \in \mathsf{D}(A) | A \neq x_2 \wedge A \neq x_3 \wedge A \neq x_7\}$ and $\{x_3, x_7\}$. We add the description of the domain of A to the parfactor [00], apply substitution $\{C/x_3\}$, and bring the factor component to the power 7; we repeat the process for substitution $\{C/x_7\}$:

$$\langle \{A \neq x_2, A \neq x_3, A \neq x_7\}, \{g(A,x_3), g(x_2,x_9)\}, {\mathcal{F}_f}^7 \rangle, \quad [03]$$
$$\langle \{A \neq x_2, A \neq x_3, A \neq x_7\}, \{g(A,x_7), g(x_2,x_9)\}, {\mathcal{F}_f}^7 \rangle; \quad [04]$$

- $\langle S_3, S_1, [\{\{A\}\}\ \{\{C\}\}]\rangle$ – A and C have disjoint domains, their domains are respectively $\{x_3, x_7\}$ and $\{x \in \mathsf{D}(C) | C \neq x_2 \wedge C \neq x_3 \wedge C \neq x_7\}$. We add the description of the domain of C to the parfactor [00], apply substitution $\{A/x_3\}$, and bring the factor component to the power 7; we repeat the process for substitution $\{A/x_7\}$:

$$\langle \{C \neq x_2, C \neq x_3, C \neq x_7\}, \{g(x_3,C), g(x_2,x_9)\}, {\mathcal{F}_f}^7 \rangle, \quad [05]$$
$$\langle \{C \neq x_2, C \neq x_3, C \neq x_7\}, \{g(x_7,C), g(x_2,x_9)\}, {\mathcal{F}_f}^7 \rangle; \quad [06]$$

- $\langle S_3, S_3, [\{\{A,C\}\}]\rangle$ – A and C are equal, their domain is $\{x_3, x_7\}$. We apply substitutions $\{A/x_3\}$ and $\{C/x_3\}$ to the parfactor [00] and bring the factor component to the power 8; we repeat the process for substitutions $\{A/x_7\}$ and $\{C/x_7\}$:

$$\langle \emptyset, \{g(x_3,x_3), g(x_2,x_9)\}, {\mathcal{F}_f}^8 \rangle, \quad [07]$$
$$\langle \emptyset, \{g(x_7,x_7), g(x_2,x_9)\}, {\mathcal{F}_f}^8 \rangle; \quad [08]$$

- $\langle S_3, S_3, [\{\{A\}, \{B\}\}]\rangle - A$ and C are different, their domain is $\{x_3, x_7\}$. We apply substitutions $\{A/x_3\}$ and $\{C/x_7\}$ to the parfactor [00] and bring the factor component to the power 8; we repeat the process for substitutions $\{A/x_7\}$ and $\{C/x_3\}$:

$$\langle \emptyset, \{g(x_3, x_7), g(x_2, x_9)\}, {\mathcal{F}_f}^8\rangle, \quad [09]$$
$$\langle \emptyset, \{g(x_7, x_3), g(x_2, x_9)\}, {\mathcal{F}_f}^8\rangle; \quad [10]$$

The summation created many parfactors, but significantly fewer than would be created by a #CSP solver that does not produced lifted description as output.

The above example strongly suggests that an extension to the calculus of parfactors that would allow reasoning about sets of individuals belonging to some population is worth pursuing.

Appendix E

Splitting as Needed

It isn't necessary. It isn't a hot appendix. It's chronic.
— Maj. Margaret 'Hot Lips' O'Houlihan (M*A*S*H)

In this appendix we apply propositions introduced in Section 2.5.2 and present a simple lifted computation which illustrates the splitting as needed approach (Section 5.2.1). We use the same model and compute the same query as in Section 2.5.2.7, where we demonstrated how the C-FOVE algorithm performs inference with the use of the shattering operation (Section 5.2.1). .

Consider the parfactors from Example 2.7 (page 23), which represent the ICL theory from Example 2.6 (page 18) and Figure 2.5 (page 19). Assume $\mathcal{D}(Lot) = \{lot_1, lot_2, \ldots, lot_n\}$ and that it is observed that grass is wet on lot_1, which can be represented by the following parfactor:

$$\langle \emptyset, \{wet\_grass(lot_1)\}, \begin{array}{c|c} \mathcal{P}(wet\_grass(lot_1) = false) & \mathcal{P}(wet\_grass(lot_1) = true) \\ \hline 0.0 & 1.0 \end{array} \rangle.$$

Let Φ be a set of the three parfactors from Example 2.7 and the above parfactor (in what follows, we don't show details of factor components of parfactors):

$$\begin{aligned}\Phi = \{\, &\langle \emptyset, \{rain()\}, \mathcal{F}_1\rangle, && [01]\\ &\langle \emptyset, \{sprinkler(Lot)\}, \mathcal{F}_2\rangle, && [02]\\ &\langle \emptyset, \{rain(), sprinkler(Lot), wet\_grass(Lot)\}, \mathcal{F}_3\rangle, && [03]\\ &\langle \emptyset, \{wet\_grass(lot_1)\}, \mathcal{F}_4\rangle\}. && [04]\end{aligned}$$

Assume we want to compute $\mathcal{J}_{ground(wet\_grass(Lot)):\{Lot \neq lot_1\}}(\Phi)$. Note that this is the joint on $wet\_grass(Lot)$ for all lots except lot_1.

Let us first eliminate the parameterized random variable $sprinkler(Lot)$. We apply Proposition 2.3 and multiply the two parfactors that involve $sprinkler(Lot)$, namely [02] and [03]. Both parfactors represent the same number of factors, namely n, and no correction is necessary. We obtain the product

$$\langle \emptyset, \{rain(), sprinkler(Lot), wet\_grass(Lot)\}, \mathcal{F}_5\rangle, \qquad [05]$$

where $\mathcal{F}_5 = \mathcal{F}_2 \odot \mathcal{F}_3$. The new set of parfactors is as follows:

$$\begin{aligned}\Phi_1 = \{\, &\langle \emptyset, \{rain()\}, \mathcal{F}_1\rangle, && [01]\\ &\langle \emptyset, \{wet\_grass(lot_1)\}, \mathcal{F}_4\rangle, && [04]\\ &\langle \emptyset, \{rain(), sprinkler(Lot), wet\_grass(Lot)\}, \mathcal{F}_5\rangle\}. && [05]\end{aligned}$$

Next, we apply Proposition 2.1 and sum out $sprinkler(Lot)$ from parfactor [05]. No logical variable disappears from the parfactor and we do not need to compensate for it. We obtain the following parfactor:

$$\langle \emptyset, \{rain(), wet\_grass(Lot)\}, \mathcal{F}_6\rangle, \qquad [06]$$

where $\mathcal{F}_6 = \sum_{sprinkler(Lot)} \mathcal{F}_5$. The new set of parfactors is as follows:

$$\begin{aligned}\Phi_2 = \{ &\langle \emptyset, \{rain()\}, \mathcal{F}_1 \rangle, && [01]\\ &\langle \emptyset, \{wet\_grass(lot_1)\}, \mathcal{F}_4 \rangle && [04]\\ &\langle \emptyset, \{rain(), wet\_grass(Lot)\}, \mathcal{F}_6 \rangle \}. && [06]\end{aligned}$$

Next, we again apply Proposition 2.3 and multiply the parfactors involving parameterized random variable *rain*(), [01] and [06], and obtain the following parfactor:

$$\langle \emptyset, \{rain(), wet\_grass(Lot)\}, \mathcal{F}_7 \rangle, \qquad [07]$$

where $\mathcal{F}_7 = \mathcal{F}_1^{\frac{1}{n}} \odot \mathcal{F}_6$. Parfactor [01] represents one factor, while parfactor [06] represents n factors, and we needed to compensate for this when computing parfactor [07]. The new set of parfactors is as follows:

$$\begin{aligned}\Phi_3 = \{ &\langle \emptyset, \{wet\_grass(lot_1)\}, \mathcal{F}_4 \rangle && [04]\\ &\langle \emptyset, \{rain(), wet\_grass(Lot)\}, \mathcal{F}_7 \rangle \}. && [07]\end{aligned}$$

We cannot eliminate *rain*() from parfactor [07], because $param(rain()) \not\supseteq param(wet\_grass(Lot))$. Instead, we apply Proposition 2.6 to parfactor [07] and perform counting on the logical variable *Lot*. We obtain

$$\langle \emptyset, \{rain(), \#_{Lot}[wet\_grass(Lot)]\}, \mathcal{F}_8 \rangle, \qquad [08]$$

where $\mathcal{F}_8$ is defined as in Equation 2.10. The new set of parfactors is as follows:

$$\begin{aligned}\Phi_4 = \{ &\langle \emptyset, \{wet\_grass(lot_1)\}, \mathcal{F}_4 \rangle && [04]\\ &\langle \emptyset, \{rain(), \#_{Lot}[wet\_grass(Lot)]\}, \mathcal{F}_8 \rangle \}. && [08]\end{aligned}$$

Now, we can sum out *rain*() from parfactor [08] (Proposition 2.1). We obtain

$$\langle \emptyset, \{\#_{Lot}[wet\_grass(Lot)]\}, \mathcal{F}_9 \rangle, \qquad [09]$$

where $\mathcal{F}_9 = \sum_{rain()} \mathcal{F}_8$. The new set of parfactors is as follows:

$$\Phi_5 = \{ \langle \emptyset, \{wet\_grass(lot_1)\}, \mathcal{F}_4 \rangle \qquad [04]$$
$$\langle \emptyset, \{\#_{Lot}[wet\_grass(Lot)]\}, \mathcal{F}_9 \rangle \}. \qquad [09]$$

Before we can multiply the two remaining parfactors, we need to apply Proposition 2.5 to parfactor [09] and expand the counting formula $\#_{Lot}[wet\_grass(Lot)]$. We obtain the following parfactor:

$$\langle \emptyset, \{wet\_grass(lot_1), \#_{Lot:\{Lot \neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_{10} \rangle, \qquad [10]$$

where $\mathcal{F}_{10}$ is defined as in Equation 2.8. The new set of parfactors is as follows:

$$\Phi_6 = \{ \langle \emptyset, \{wet\_grass(lot_1)\}, \mathcal{F}_4 \rangle \qquad [04]$$
$$\langle \emptyset, \{wet\_grass(lot_1), \#_{Lot:\{Lot \neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_{10} \rangle \}. \qquad [10]$$

Next, we multiply parfactor [04] by parfactor [10] (Proposition 2.3) and obtain:

$$\langle \emptyset, \{wet\_grass(lot_1), \#_{Lot:\{Lot \neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_{11} \rangle, \qquad [11]$$

where $\mathcal{F}_{11} = \mathcal{F}_{10} \odot \mathcal{F}_4$. The new set of parfactors is as follows:

$$\Phi_7 = \{ \langle \emptyset, \{wet\_grass(lot_1), \#_{Lot:\{Lot \neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_{11} \rangle \}. \qquad [11]$$

We sum out $wet\_grass(lot_1)$ from parfactor [11] (Proposition 2.1) and obtain

$$\langle \emptyset, \{\#_{Lot:\{Lot \neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_{12} \rangle, \qquad [12]$$

where $\mathcal{F}_{12} = \sum_{wet\_grasslot_1} \mathcal{F}_{11}$. The new set of parfactors is as follows:

$$\Phi_8 = \{ \langle \emptyset, \{\#_{Lot:\{Lot \neq lot_1\}}[wet\_grass(Lot)]\}, \mathcal{F}_{12} \rangle \}. \qquad [12]$$

We have

$$\mathcal{J}_{ground(wet\_grass(Lot)):\{Lot \neq lot_1\}}(\Phi) = \mathcal{J}(\Phi_8).$$

During computation, constants $lot_2, lot_3, \ldots, lot_n$ were not explicitly enumerated, we only needed to know that $\mathcal{D}(Lot) = n$. The biggest factor created during inference had size 8 (factor $\mathcal{F}_3$) for $n \leq 4$ and $2(n+1)$ (factor $\mathcal{F}_8$) for $n > 4$. We performed 1 expansion, 3 multiplications, 1 counting operation, and 3 summations. During inference presented in Section 2.5.2.7, the C-FOVE algorithm performed 2 splits, 5 multiplications, 1 counting operation, and 4 summations.