THESE

Pour l'obtention du titre de
DOCTEUR EN INFORMATIQUE

spécialité : aide à la décision

Inférence probabiliste structurée dans les modèles graphiques probabilistes orientés-objet

Candidat : Lionel TORTI

JURY

Acknowledgments

Contents

Notations

P	a probability distribution
X, Y, Z	random variables, BN's nodes or classes attributes
x, y, z	instantiated random variables
$\mathbf{X}, \mathbf{Y}, \mathbf{Z}$	set of random variables
$\mathbf{x}, \mathbf{y}, \mathbf{z}$	set of instantiated random variables
$\text{Val}(X)$	the set of values of X, also called its domain
$\text{Val}(\mathbf{X})$	a set obtained by the Cartesian product $\text{Val}(\mathbf{X}) = \bigotimes_{X \in \mathbf{X}} \text{Val}(X)$
$(X \perp\!\!\!\perp Y)$	X is independent of Y
$(X \perp\!\!\!\perp Y \mid Z)$	X is conditionally independent of Y given Z
$P \models (X \perp\!\!\!\perp Y \mid Z)$	P models the conditional independence $(X \perp\!\!\!\perp Y \mid Z)$
$G = (V, E)$	an undirected graph over nodes V and edges E
$\vec{G} = (V, E)$	a directed graph over nodes V and arcs E
$\pi(X)$	the parents of X
$\text{Ch}(X)$	the children of X
Nghbrs_X	the neighbors of X
$X \leftrightarrow Y$	the arc $X \rightarrow Y$ or $X \leftarrow Y$

Introduction

Computer science is the study of algorithms and computational problems. Remarkably, problems that are easily solved by humans are not always as easily solved by computers. Modeling and solving problems is one of the most challenging aspect of computer science. Humans and computers skills are profoundly different: computers are extremely good at analyzing and manipulating large amounts of data. For computers, there is no difference between sorting a list of ten or thousand of integers. However, such scale changes is intractable for human minds. For humans, reasoning and infering about concepts is easy: differentiating chairs from tables is trivial for humans, but is more complex for computers. In fact, simulating human intelligence, called Artificial Intelligence (AI) is certainly the most difficult and challenging problem of computer science. Creating an AI is much more difficult than expected and nowadays, AI researchers focus on different subfields that are relevant to an aspect of human intelligence: knowledge representation, planning, machine learning and reasoning are the most popular ones. This thesis belongs to the reasoning branch of AI, but it also has connections with knowledge representation and machine learning.

Reasoning is about reproducing human decision processes: given some knowledge about the world, what decision is the best? Humans answer such problems everyday, but, surprisingly, simulating such task is incredibly difficult. The first difficulty is how to represent knowledge and logic was at first thought as the best formalism to represent knowledge. Unfortunately, knowledge bases using logic were confronted to paradoxes, such as the infamous "titi is a bird thus titi can fly" but what if titi is a penguin? Handling exceptions is not an easy task for logic and soon the focus was set on alternative theories. Among them, probability theory introduced a new aspect to reasoning: uncertainty. However, probability theory cannot be used as is, since representing knowledge using joint probability distribution consumes too much memory and too much time to exploit. In the eighties, a solution emerged exploiting conditional independencies to reduce the memory cost of representing joint probability distributions and offering tractable inference algorithms to process the knowledge base. That solution is still used today and offers the framework on which this thesis is based: Probabilistic Graphical Models (PGMs).

PGMs are a family of mathematical models combining probability theory and graph theory. They are used in many domains but are historically associated with decision theory and uncertain reasoning. They are, however, used in machine learning, most especially in data mining. For troubleshooting, risk management, reliability and most domains were expert knowledge is used, Bayesian Networks (Pearl, 1988) are a valued framework for reasoning under uncertainty. Bayesian Networks (BNs) use a Directed Acyclic Graph (DAG) to represent random variables and their conditional dependencies. In a BN a random variable conditional probability distribution is conditioned by the variable's parents in the graph. As a consequence, BNs offer an intuitive framework for representing causal dependencies (but all dependencies in a BN are not necessarily causal). BNs have received a remarkable amount of contributions by the AI community and have been used in many industrial applications. Their popularity stimulated the need for handling problems of ever increasing size. However BNs turn out to be inadequate for large scale real-world applications due to high design and maintenance costs (Mahoney and Laskey, 1996; Koller and Pfeffer, 1997). Indeed, defining a BN requires to specify explicitly probabilistic dependencies and conditional probabilities over the whole set of its random variables. This may lead to unrealistic modeling costs when dealing with complex systems. Furthermore, BNs designs are static: any change in the topology of their graphical structure induces significant update costs.

A proposed solution to enhance BNs expressive power was to extend BNs using the object-oriented paradigm (Mahoney and Laskey, 1996; Koller and Pfeffer, 1997). This first paradigm shift led to the emergence of different families of BNs extensions: object-oriented, entity-relationship and first-order paradigms. During the last decade, the PGMs community has worked actively on first-order extensions and the object-oriented paradigm has been somewhat neglected: since the introduction of Object-Oriented Bayesian Networks (Koller and Pfeffer, 1997; Bangsø and Wuillemin, 2000a,b), the amount of contributions on object-oriented PGMs has actually been relatively small (Pfeffer, 1999; Bangsø et al., 2003; Bangsø and Olesen, 2003; Bangsø, 2004; Bangsø et al., 2006). However, in many industrial applications, efficient frameworks for constructing large-scale complex systems are strongly needed. These systems usually boil down to experts modeling large-scale BNs by aggregating hierarchically small network fragments repeated many times. In addition, all the relations between these fragments are usually fully specified, thus resulting in modeling *closed worlds*. For these domains, object-oriented frameworks seem more suitable than first-order logic extensions. Object-Oriented frameworks assume that many parts of a large BN are similar and can thus be described as instances of a generic class defined only once. This scheme induces low construction costs. In addition, maintenance costs are kept as low as possible since a modification in a class definition updates many areas of the BN at once. Furthermore, repetitions of structures in the BN (multiple instances of the same class) can speed-up inference by performing computations within classes, caching them and using the cache for all their instances. This process allows algorithms like *Structured Variable Elimination* to outperform classical BN inference engines by orders of magnitude (Pfeffer, 1999).

To promote Object-Oriented Bayesian Networks, the SKOOB ANR project (`http://skoob.lip6.fr`) regroups several industrials, experts and research laboratories with the purpose of defining a framework in which modeling large-scale complex network is tractable. We have found that state-of-the-art object-oriented extensions of BNs have not fulfilled this goal, as the current theories clearly lack some useful object-oriented features. From a modeling perspective, existing object-oriented frameworks offer very few object-oriented features. Many features are either improperly defined or absent (inheritance, subtype polymorphisms, abstraction). However, modeling is only half of the problem: since Object-Oriented Bayesian Networks can be used to model large-scale systems, experts can model networks that are intractable using current state-of-the-art probabilistic inference. Consequently, besides defining a strong object-oriented framework, we must also define probabilistic inference algorithms that can be exploited on large-scale systems.

This thesis regroups the contribution of the LIP6 in the SKOOB ANR project, and its goal is to offer a solid framework for specifying large-scale complex systems and tractable probabilistic inference algorithms. All the contributions in this thesis have either been implemented in the aGrUM C++ framework (`http://agrum.lip6.fr`) or in the SKOOL language (see appendix A). The first chapter of this thesis presents PGMs, focusing on BNs and Markov Networks. These two models are at the core of many applications and have been used as a basis to many high-level PGMs. The second chapter focuses on evidence updating, an application of probabilistic inference that updates marginal probabilities when random variables are observed. In the third chapter we introduce several BN extensions and discuss the different representation paradigms used to encode knowledge in such frameworks. The fourth chapter presents a new framework: an object-oriented PGM built upon Probabilistic Relational Models. We redefine and extend several existing concepts, such as class inheritance, subtype polymorphisms, and present new object-oriented features such as multiple inheritance and attributes types inheritance. The fifth chapter focuses on structured probabilistic inference. We redefine SVE and point-out several of its flaws. We then propose a generalization of structured inference and expose how d-separation analysis can be used conjointly with structured inference to increase probabilistic inference performance. Finally, in the last chapter, we present a pattern discovery algorithm which mine repeated instances patterns in a system to offer better performance to structured inference.

Chapter 1

Probabilistic Graphical Models

For several decades, representing uncertainty has been a challenge for the AI community and probability theory was seen as a promising framework to answer this challenge. However, uncertain reasoning using probabilities was considered intractable until the first appearance of PGMs (Pearl, 1988). These graphs exploit independencies present in probability distributions to factorize them and offer a tractable representation for uncertain reasoning. In this thesis, we will focus on BNs, a directed PGM successful in modeling probabilistic expert systems (Cowell et al., 1999). We will also discuss Markov Networks (MNs), an undirected counterpart of BNs, which play an important role in probabilistic inference. By permitting to model uncertainty using probabilities, BNs quickly became an important tool in many computer science fields and they are definitively a milestone on the road leading to AI. Not only BNs were rapidly considered by academics as a sound framework for uncertain reasoning, they were also quickly used in many industrial applications: troubleshooting, risk management, reliability, automatic classification, data mining, multi agent systems, robotics… BNs industrial success can be attributed to three facts: (i) they are a simple data structure, inducing low implementation costs; (ii) they require only basic probability theory knowledge; (iii) they are easily understandable by non computer scientists.

We will introduce BNs through three chapters. The first chapter focuses on the mathematical justification of BNs. Chapter 2 introduces two important algorithms for probabilistic inference in BNs. Chapter 3 presents several BN extensions. We will neither cover structure and parameter learning, nor approximate inference. Still these topics are as important as exact inference and the reader should refer to Pearl (1988); Jordan (1999); Koller and Friedman (2009) for a complete presentation of these topics. Koller and Friedman (2009) was a great source for writing this chapter, most definitions are taken from there and the reader should refer to it for any additional material. Pearl (1988) is also an excellent starting point for anyone who wishes to grasp a better understanding of PGMs. In this chapter, we will first introduce the core notions of PGMs, then present MNs and BNs. We will then show the similarity between both models by explaining how BNs can be transformed into MNs. Finally, we will present several alternative representations of conditional probability distributions commonly used by experts when modeling BNs.

1.1 Independencies, graphs and I-maps

The PGMs main problem is to model state spaces: probability theory is well known to be under the curse of dimensionality. Modeling complex systems with a minimal amount of parameters is the main purpose of BNs emergence. To do so, BNs exploit conditional independencies to factorize joint probability distributions and offer a tractable framework for probabilistic inference, i.e., making possible to compute marginal probabilities (see chapter 2). To understand the link between graphical properties and conditional independence we will first explain the similarity between graphs and probability.

Example 1.1 *Consider an experiment in which we toss two fair dice D_1 and D_2. Let X be the random variable representing the result of the first toss using D_1. X's domain is the values Val$(X) = \{1, 2, 3, 4, 5, 6\}$. Let Y be the random variable representing the result of the second toss using D_2 and Val(Y) = Val(X). Obviously, the realization of X does not influence our knowledge about Y's possible outcomes. In such case, we say that X and Y are marginally independent, denoted $(X \perp\!\!\!\perp Y)$. Now, suppose we consider another experiment for which we have the three dice D_1, D_2 and D_3, such that D_1 and D_2 are fair but not D_3 (it always gives a six). We also change how the experiment realizes itself: we first toss D_1 and if the result is even we toss D_2, otherwise we toss D_3. We denote by random variable X the result of the first toss and by Y the result of the second. Now, we do no longer have marginal independence between X and Y since X's resolution has a direct influence of Y's outcomes. Y's result also influences our beliefs about X outcomes: if $Y \neq 6$ we are certain that X's value is not an odd number.*

Marginal independencies are intuitive notions. We are surrounded by events realizing themselves that are not interconnected. In the field of uncertain reasoning, there exists another kind of independence that proves to be much more useful to represent related events. It is called conditional independence.

Example 1.2 *Suppose we have two computers represented by random variables C_1 and C_2 (Val(C_1) = Val(C_2) = $\{OK, NOK\}$) connected through the same power plug $Power$ (Val$(Power)$ = $\{on, off\}$) and we observe that C_1 is broken. The cause is either a power surge or some random failure. A simple way to check if there has been a power surge is to look at C_2. If C_2 does not work then it is very likely that a power surge occurred: C_1 and C_2 are correlated. However, once we learned that a power surge occurred or not, C_2's state becomes irrelevant for troubleshooting C_1: if $Power$ is observed, C_1 and C_2 are independent. In such a case, we say that C_1 is conditionally independent of C_2 knowing $Power$, denoted $(C_1 \perp\!\!\!\perp C_2 | Power)$.*

Conditional independencies are very useful when modeling correlation among random variables. We will see that they are at the core of PGMs by being the link between the graphical part and the probabilistic part of these models. Independencies and conditional independencies in probability distributions follow a set of axioms called the graphoid axioms.

Definition 1.1 (Conditional Independence) *Let $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$ be three sets of random variables. We say that $\mathbf{X}$ is conditionally independent of $\mathbf{Y}$ given $\mathbf{Z}$ in a probabilistic distribution P if $P \models (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} | \mathbf{Z} = \mathbf{z})$ for all values of $\mathbf{z} \in$ Val$(\mathbf{Z})$. If the set $\mathbf{Z}$ is empty, then instead of writing $(\mathbf{X} \perp\!\!\!\perp \mathbf{Y} | \emptyset)$, we write $(\mathbf{X} \perp\!\!\!\perp \mathbf{Y})$ and say that $\mathbf{X}$ and $\mathbf{Y}$ are (marginally) independent. We denote by $\mathcal{I}(P)$ the set of all independence assertions present in a probability distribution P.*

Note that conditional independence entails that $P(X, Y|Z) = P(X|Z) \times P(Y|Z)$. Conditional independence in a probability distribution satisfies the semi-graphoid axioms (Pearl and Paz, 1987).

Definition 1.2 (Graphoid axioms) *The following four axioms are called the graphoid axioms:*

a) Symmetry:
$(\mathbf{X} \perp\!\!\!\perp \mathbf{Y} | \mathbf{Z}) \Rightarrow (\mathbf{Y} \perp\!\!\!\perp \mathbf{X} | \mathbf{Z})$.

b) Decomposition:
$(\mathbf{X} \perp\!\!\!\perp \mathbf{Y}, \mathbf{W} | \mathbf{Z}) \Rightarrow (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} | \mathbf{Z})$.

c) Weak union:
$(\mathbf{X} \perp\!\!\!\perp \mathbf{Y}, \mathbf{W} | \mathbf{Z}) \Rightarrow (\mathbf{X} \perp\!\!\!\perp \mathbf{Y} | \mathbf{Z}, \mathbf{W})$.

d) Intersection (only if P is positive[1]):
$(\mathbf{X} \perp\!\!\!\perp \mathbf{Y} | \mathbf{Z}, \mathbf{W}) \wedge (\mathbf{X} \perp\!\!\!\perp \mathbf{W} | \mathbf{Z}, \mathbf{Y}) \Rightarrow (\mathbf{X} \perp\!\!\!\perp \mathbf{Y}, \mathbf{W} | \mathbf{Z})$.

For a ternary relation such as $(\bullet \perp\!\!\!\perp \bullet | \bullet)$, properties a,b and c characterize a structure called a semi-graphoid and properties a, b, c and d characterize a structure called a graphoid.

Graphoid and semi-graphoid structures occur frequently in many domains (Studeny, 1990). For instance, separation in directed and undirected graphs verify the same structure. PGMs exploit graphoids by using graphs to encode the independencies present in a probability distribution. Such representation offers several advantages: it factorizes the probability distribution, it helps expert analysis of models and it helps design algorithms (either for probabilistic inference or for data mining).

Definition 1.3 (Active path) *Let $G = (V, E)$ be an undirected graph, let $X_1 - \cdots - X_k$ be a path in G, and let $\mathbf{Z} \subset V$. The path $X_1 - \cdots - X_k$ is active given $\mathbf{Z}$ if and only if $\forall X \in \{X_1, \cdots, X_n\}$, $X \notin \mathbf{Z}$.*

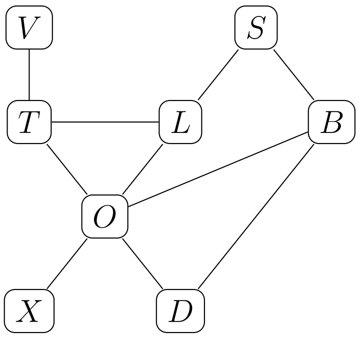
Example 1.3 *In figure 1.1a we can see that path $V - T - L - S$ is active given $\{B, O\}$. However, path $V - T - O - D$ is not active given $\{B, O\}$.*

[1]A probability distribution P is positive if and only if $P(\mathbf{X} = \mathbf{x}) > 0$, $\forall \mathbf{x} \in$ Val$(\mathbf{X})$.

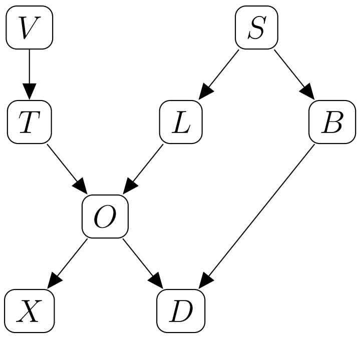
Definition 1.4 (Separation) *Let* $\mathbf{X}$, $\mathbf{Y}$ *and* $\mathbf{Z}$ *three sets of nodes in* G. *We say that* $\mathbf{Z}$ *separates* $\mathbf{X}$ *and* $\mathbf{Y}$ *in* G, *denoted* $sep_G(\mathbf{X};\mathbf{Y}|\mathbf{Z})$, *if and only if there is no active path between any node* $X \in \mathbf{X}$ *and* $Y \in \mathbf{Y}$ *given* $\mathbf{Z}$. *We define the separations associated with G to be:*

$$\mathcal{I}(G) = \{sep_G(\mathbf{X};\mathbf{Y}|\mathbf{Z})\}.$$

Example 1.4 *Looking at figure 1.1a we can find several separations:* $sep(\{X\};\{D\}|\{O\})$ *or* $sep(\{V,S\};\{X,D\}|\{B,O\})$.



(a) An undirected graph.



(b) A directed graph.

Figure 1.1: Separation in undirected and directed graphs follows the graphoid axioms.

Let $\vec{G}$ be a directed graph and X_i and X_j two nodes in $\vec{G}$. We denote by $X_i \leftrightarrow X_j$ the existence of a directed link between X_i and X_j in $\vec{G}$, i.e., $X_i \leftarrow X_j$ or $X_i \rightarrow X_j$ exists in $\vec{G}$.

Definition 1.5 (Active trail) *Let* $\vec{G} = (V,E)$ *a directed graph, let* $X_1 \leftrightarrow \cdots \leftrightarrow X_k$ *be a trail in* $\vec{G}$ *and let* $\mathbf{Z} \subset V$. *The trail* $X_1 \leftrightarrow \cdots \leftrightarrow X_k$ *is active given* $\mathbf{Z}$ *if and only if:*

- *whenever we have a v-structure* $X_{i-1} \rightarrow X_i \leftarrow X_{i+1}$ $(1 < i < k)$, *then* X_i *or one of its descendants is in* $\mathbf{Z}$*;*
- *no other node along the trail is in* $\mathbf{Z}$.

Example 1.5 *In figure 1.1b the trail* $V \rightarrow T \rightarrow O \leftarrow L \leftarrow S$ *is active given* $\{X,D\}$ *and trail* $L \leftarrow S \rightarrow B$ *is not active given* $\{S\}$.

Definition 1.6 (D-separation) *Let* $\mathbf{X}$, $\mathbf{Y}$ *and* $\mathbf{Z}$ *three sets of nodes in* $\vec{G}$. *We say that* $\mathbf{Z}$ *d-separates* $\mathbf{X}$ *and* $\mathbf{Y}$ *in* $\vec{G}$, *denoted d-sep*$_{\vec{G}}(\mathbf{X};\mathbf{Y}|\mathbf{Z})$, *if and only if there is no active trail between any node* $X \in \mathbf{X}$ *and* $Y \in \mathbf{Y}$ *given* $\mathbf{Z}$. *We define the directed separations associated with* $\vec{G}$ *to be:*

$$\mathcal{I}(\vec{G}) = \{\text{d-sep}_{\vec{G}}(\mathbf{X};\mathbf{Y}|\mathbf{Z})\}.$$

Example 1.6 *Looking at figure 1.1b we can see that d-sep*$(\{D\};\{S\}|\{B,O\})$ *and that d-sep*$(\{V\};\{S\}|\{T,L\})$ *but that* O *does not d-separate* V *from* S.

In the following sections we will discuss how we can use directed and undirected graphs to represent the set of conditional independencies present in a probability distribution. The main notion is to map conditional independencies to directed or undirected separations in graphs.

Definition 1.7 (I-map, minimal I-map and P-map) *Let G be a graph (directed or not) and P be a probability distribution. We say that G is an Independence map (I-map) of P if and only if $\mathcal{I}(G) \subseteq \mathcal{I}(P)$. We say that G is a minimal I-map of P if and only if removing any edge of G makes it no longer an I-map of P. Finally, if $\mathcal{I}(G) = \mathcal{I}(P)$ we say that G is a perfect map (P-map) of P.*

We may, by abuse of notation, say that graph G is an I-map to graph G', in such case it simply entails that $\mathcal{I}(G) \subseteq \mathcal{I}(G')$.

1.2 Markov Networks

MNs uses factors to encode probability distributions. They play an important role as factors are closely related to the MN's graph and operations over them are central for probabilistic inference.

Definition 1.8 (Factor) *Let $\mathbf{X}$ be a set of random variables. We define a factor ϕ to be a function ϕ : Val$(\mathbf{X}) \mapsto \mathbb{R}$. A factor is nonnegative if all of its entries are different from 0. The set of random variables $\mathbf{X}$ is called the scope of the factor and denoted Scope(ϕ).*

When used in MNs, factors do not map random variables values to probability. In most cases, they assign some nonnegative value that represents the amount of belief that an outcome may realize itself. Regarding probabilistic inference, we require two operators over factors: multiplication and marginalization.

Definition 1.9 (Factor multiplication) *Let $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$ be three disjoint sets of random variables, and let $\phi_1(\mathbf{X}, \mathbf{Y})$ and $\phi_2(\mathbf{Y}, \mathbf{Z})$ be two factors. We define the factor product $\phi_1 \times \phi_2$ to be a factor ψ : Val$(\mathbf{X}, \mathbf{Y}, \mathbf{Z}) \mapsto \mathbb{R}$ as follows:*

$$\psi(\mathbf{X} = \mathbf{x}, \mathbf{Y} = \mathbf{y}, \mathbf{Z} = \mathbf{z}) = \phi_1(\mathbf{X} = \mathbf{x}, \mathbf{Y} = \mathbf{y}) \times \phi_2(\mathbf{Y} = \mathbf{y}, \mathbf{Z} = \mathbf{z})$$

Definition 1.10 (Factor Marginalization) *Let $\mathbf{X} = \{X_1, \cdots, X_n\}$ be a set of random variables, and let $\phi(\mathbf{X})$ be a factor. We define the factor marginalization $\sum_{X_i} \phi(\mathbf{X})$, to be a factor ψ : Val$(\mathbf{X} \backslash \{X_i\}) \mapsto \mathbb{R}$ as follows:*

$$\psi(X_1, \cdots, X_{i-1}, X_{i+1}, \cdots, X_n) = \sum_{x_i \in Val(X_i)} \phi(X_1, \cdots, X_i = x_i, \cdots, X_n). \tag{1.1}$$

Factor marginalization is also referred as "summing out variable X of ϕ". Both operations are critical for inference performance, however we do not yet concern ourselves with such issues. Factors do not necessarily assign probability values, however MNs are used to make probabilistic inference and to do so they are coupled with specific probability distributions called Gibbs distributions.

Definition 1.11 (Gibbs distribution) *Let $\mathbf{X} = \{X_1, \cdots, X_n\}$ be a set of random variables and P_Φ be a probability distribution over $\mathbf{X}$. P_Φ is a Gibbs distribution parameterized by a set of factors $\Phi = \{\phi_1(\mathbf{C}_1), \cdots, \phi_k(\mathbf{C}_k)\}$ such that $\bigcup_{i=1}^{k} \mathbf{C}_i = \mathbf{X}$ if and only if:*

$$P_\Phi(X_1, \cdots, X_n) = \frac{1}{Z} \tilde{P}_\Phi(X_1, \cdots, X_n),$$

where

$$\tilde{P}_\Phi(X_1, \cdots, X_n) = \prod_{i=1}^{k} \phi_i(\mathbf{C}_i)$$

is an unnormalized measure and

$$Z = \sum_{X_1, \cdots, X_n} \tilde{P}_\Phi(X_1, \cdots, X_n)$$

is a normalizing constant called the partition function.

Computing the partition function is one of the most challenging aspects of MNs inference algorithms. Since we focus on directed models in this thesis, we will not detail the different algorithms to compute the partition function. The reader should refer to Koller and Friedman (2009) for additional material.

Definition 1.12 (Gibbs Distribution factorization) *We say that a Gibbs distribution P_Φ with $\Phi = \{\phi_1(\mathbf{C}_1), \cdots, \phi_k(\mathbf{C}_k)\}$ factorizes over a Markov Network Graph $\mathcal{H}$ if each $\mathbf{C}_i$, $1 \leqslant i \leqslant k$ is a complete subgraph of $\mathcal{H}$.*

Each complete subgraph $\mathbf{C}_i$ is called a *clique* and the factors that parameterize a MN are also called clique potentials. We can now give the definition of a MN.

Definition 1.13 (Markov Network) *A Markov Network is a pair $\mathcal{H} = (P_\Phi, G)$ where P_Φ is a Gibbs distribution that factorizes over G and G is an undirected graph that is an I-map of P_Φ.*

In the precedent section, we have seen that separation defines a structure equivalent to independencies in a probability distribution. In fact, if a probability distribution factorizes over an undirected graph then the graph is an I-map of the probability distribution.

Theorem 1.1 (Soundness) *Let P be a distribution over $\mathbf{X}$, and $\mathcal{H}$ a Markov Network structure over $\mathbf{X}$. If P is a Gibbs distribution that factorizes over $\mathcal{H}$, then $\mathcal{H}$ is an I-map for P.*

Proof in Koller and Friedman (2009). Not only the fact that a probability distribution factorizes over an undirected graph gives us information about independencies existing in it, but it also gives us the guarantee that some dependencies represented by edges can exist.

Theorem 1.2 (Completness) *Let $\mathcal{H}$ be a Markov Network structure, X and Y two random variables in $\mathcal{H}$ and $\mathbf{Z}$ a set of random variables in $\mathcal{H}$. If X and Y are not separated given $\mathbf{Z}$ in $\mathcal{H}$, then X and Y are dependent given $\mathbf{Z}$ in some distribution P that factorizes over $\mathcal{H}$.*

Proof in Koller and Friedman (2009). The next theorem tells us that if an undirected graph is an I-map to a probability distribution, then there exists a factorization of that distribution over the graph.

Theorem 1.3 (Hammersley-Clifford) *Let P be a positive distribution*[2] *over $\mathcal{X}$, and $\mathcal{H}$ a Markov Network graph over $\mathcal{X}$. If $\mathcal{H}$ is an I-map for P, then P is a Gibbs distribution that factorizes over $\mathcal{H}$.*

Proof of theorem 1.3 is omitted and can be found in Koller and Friedman (2009). With theorems 1.1, 1.2 and 1.3 we have now enough theoretical justification to use MNs as a valid factorization of joint probability distributions. We will end this chapter with an important notion.

Definition 1.14 (Markov Blanket) *For a given graph $\mathcal{H}$ and a random variable X in $\mathcal{H}$, we define the Markov blanket of X in $\mathcal{H}$, denoted $MB_{\mathcal{H}}(X)$, to be the neighbors of X in $\mathcal{H}$.*

A Markov blanket contains all the variables that separate a node from the rest of the network. Thus, a Markov blanket is the only required knowledge to predict the node's possible outcomes (Pearl, 1988). Markov blankets are a very useful notion that can be found throughout most PGMs. For MNs, they are used for classification or for expert knowledge elicitation.

1.3 Bayesian Networks

Where MNs rely on undirected graphs, BNs are defined using directed graphs. It may look like a small difference at first, but MNs and BNs applications differ greatly. The most notable difference lies in the fact that BNs use conditional probability distributions whereas MNs combine factors with a partition function.

[2]A positive distribution is strictly positive: $P(\mathcal{X} = \mathbf{x}) > 0$ for all $\mathbf{x} \in \text{Val}(\mathcal{X})$.

Definition 1.15 (Factorization w.r.t. a directed graph) *Let $\vec{G}$ be a directed graph over variables $\mathbf{X} = X_1, \cdots, X_n$. We say that a probability distribution P over $\mathbf{X}$ factorizes according to $\vec{G}$ if P can be expressed as the product:*

$$P(X_1, \cdots, X_n) = \prod_{i=1}^{n} P(X_i | \pi(X_i)). \tag{1.2}$$

Definition 1.15 is often called *the chain rule for BN* and is similar to definition 1.12 and to the chain rule in probability theory. We can now provide the definition of a BN.

Definition 1.16 (Bayesian Network) *A Bayesian Network is a pair $\mathcal{B} = (P, \vec{G})$ where P is a probability distribution and $\vec{G}$ a Directed Acyclic Graph (DAG) such that P factorizes over $\vec{G}$ and $\vec{G}$ is an I-map of P.*

BNs differ from MNs in two ways: they rely on a DAG where MNs use an undirected graph; the probability distribution associated with a BN factorizes differently than the one associated with a MN. These subtle differences have a huge impact on the use of these frameworks. The most notable one being the ease for experts to model knowledge using BNs. Eliciting conditional probability distributions as Conditional Probability Tables (CPTs) is an easier task than eliciting factors, simply because CPTs are filled with probabilities which are mathematical objects commonly manipulated by experts. However, we will see that theoretical results for BNs are similar to those for MNs.

Active trails in BNs are more complex than their undirected counterpart. This is due to the special treatment of v-structures. To understand their role it is convenient to illustrate active trails as flows of information. Depending on the graph's topology, such flows can be blocked or activated given evidence. There are three structures that can be found in a BN:

- **chains** $X \rightarrow Z \rightarrow Y$ or $X \leftarrow Z \leftarrow Y$ then evidence on Z blocks the trail between X and Y;
- **common parents** $X \leftarrow Z \rightarrow Y$ then evidence on Z also blocks the trail between X and Y;
- **v-structures** $X \rightarrow Z \leftarrow Y$ then evidence on Z does not block the trail between X and Y.

In the first two cases, the middle node blocks the flow of information once it is observed. In such cases, we usually remove the out-going arcs of an observed node to illustrate the fact that the flow is blocked. We can also check by summing out $Z = z$, in the first case we obtain three factors $\{\phi_1(X), \phi_2(X), \phi_3(Y)\}$. In the second case we obtain $\{\phi_1(X), \phi_2(Y)\}$. The last case is different, when observed Z activates the flow of information between X and Y. Consequently, any information on X will update our beliefs about Y. If we look at the resulting factors obtained after summing out $Z = z$ we obtain $\{\phi_1(X), \phi_2(X, Y), \phi_3(Y)\}$. We clearly see that a factor connects X and Y here. The fact that d-separation correctly represents independence assumptions for probability distributions that factorize over a BN graph is less intuitive than separation is for MNs. The following two theorems provide mathematical justification for d-separation in BNs.

Theorem 1.4 (D-separation soundness) *Let $\vec{G}$ be a directed graph whose nodes are a set of random variables $\mathbf{X}$ and let P be a joint distribution over $\mathbf{X}$. If P factorizes according to $\vec{G}$, then $\vec{G}$ is an I-map for P.*

Proof for theorem 1.4 is omitted, a complete proof can be found in Koller and Friedman (2009). Theorem 1.4 is very useful from a modeling perspective. Indeed, to obtain a valid BN we can specify direct dependencies and fill the resultant CPTs to obtain a factorized probability distribution for which the BN graph is an I-map. Consequently, defining a BN is a simple and intuitive task.

Theorem 1.5 (D-separation completeness) *Let $\vec{G}$ be a directed graph whose nodes are a set of random variables $\mathbf{X}$. If X and Y are not d-separated given $\mathbf{Z}$ in $\vec{G}$, then X and Y are dependent given $\mathbf{Z}$ in some distribution P that factorizes over $\vec{G}$.*

Proofs for theorems 1.4 and 1.5 are omitted and can be found in Koller and Friedman (2009). They simply state that d-separation is a valid graphical representation of conditional independence. Finally, we will end this chapter over BNs with an important result.

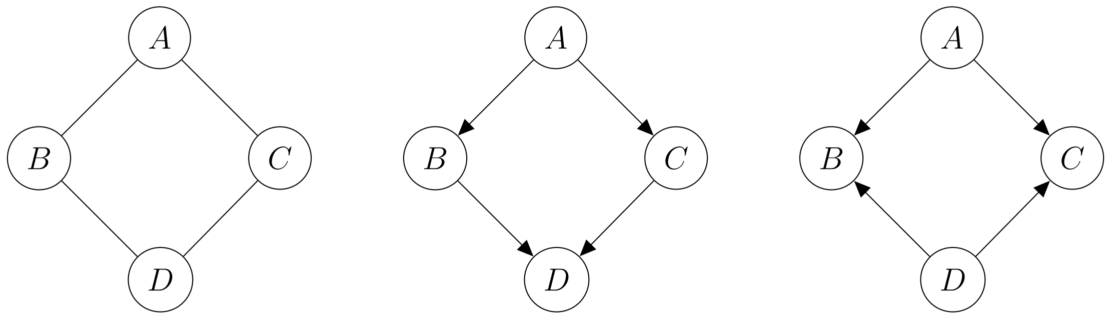
Theorem 1.6 *Let $\vec{G}$ be a directed graphs whose nodes are a set of random variables $\mathbf{X}$, and let P be a joint probability distribution over the same space. If $\vec{G}$ is an I-map for P, then P factorizes according to $\vec{G}$.*

Theorem 1.7 *For almost all distributions P that factorize over $\vec{G}$, that is, for all distributions except for a set of measure zero in the space of CPD parameterizations, we have that $\mathcal{I}(P) = \mathcal{I}(\vec{G})$.*

Proofs for both theorems can be found in Koller and Friedman (2009). Theorem 1.7 tells us that almost all probability distributions that factorize over a BN graph are perfectly represented in terms of independence assumptions. The set of probability distributions for which this statement is untrue includes distributions with extreme probability values (many zero probability values).

Definition 1.17 (Markov Blanket) *For a directed graph $\vec{G} = (V, E)$, we define the Markov blanket of $X \in V$, denoted $MB_{\vec{G}}(X)$, to be X's parents, children and children's parents.*

In opposition to Markov blankets in undirected graphs, a Markov blanket of some node X in a directed graph includes nodes that are not direct neighbors of X. This is because nodes with common children form v-structures, inducing dependencies among parents of the same nodes. Thus if a node X has a child Y that has a parent Z that is not connected to X, there is a v-structure $X \rightarrow Y \leftarrow Z$ that induces a dependency between X and Z if Y is observed. Consequently, Z must be in X's Markov blanket.



(a) A MN in which sep($A; D|\{B, C\}$) and sep($B; C|\{A, D\}$).

(b) A BN in which d-sep($A; D|\{B, C\}$) but where B and C are not d-separated by $\{A, D\}$.

(c) A BN in which d-sep($B; C|\{A, D\}$) but where A and D are not d-separated by $\{B, C\}$.

Figure 1.2: MNs can represent set of independencies that BNs cannot.

1.4 From Bayesian Networks to Markov Networks

MNs and BNs are closely related, but they both cannot necessarily represent independencies that the other cannot. First, let us illustrate by the following example a case where they do.

Example 1.7 *Let us first consider the following BN $\mathcal{B} : A \rightarrow B \rightarrow C$. It defines the following joint probability distribution: $P(A, B, C) = P(A) \times P(B|A) \times P(C|B)$. Now let us consider the MN over $\mathcal{M} : A - B - C$. The joint probability distribution of this MN would be $P(A, B, C) = \frac{1}{Z}\phi_1(A, B) \times \phi_2(B, C)$. In this case, possible values for the MN's factors could be $\phi_1(A, B) = P(A) \times P(B|A)$ and $\phi_2(B, C) = P(C|B)$. The partition function Z would then be equal to 1. We can also see that $\mathcal{B}$ and $\mathcal{M}$ both encode the same set of independencies: d-sep($A; C|B$) and sep($A; C|B$).*

The previous example illustrates a situation in which both a BN and a MN encode the same set of independencies. However, in most situations MNs and BNs with similar graphical structures encode different dependencies.

Example 1.8 *Let A, B, C and D be four random variables and let us suppose we want to model the following independencies $(A \perp\!\!\!\perp D|\{B, C\})$ and $(B \perp\!\!\!\perp C|\{A, D\})$. Figure 1.2a illustrates a MN for which both independencies exist and no other. Figure 1.2b is an attempt to model the same set of independencies using a BN. Unfortunately, whereas we have d-sep($A; D|\{B, C\}$), B and C are not d-separated by $\{A, C\}$. Figure 1.2c is another unsuccessful attempt to model figure 1.2a independencies. If we look at the joint probability distributions induced by figures 1.2b and 1.2c we have (respectively) $P(A, B, C, D) = P(A) \times P(B|A) \times P(C|A) \times P(D|B, C)$ and $P(A, B, C, D) = P(A) \times P(D) \times P(B|A, D) \times P(C|A, D)$. If we consider those conditional probability distributions as factors, they induce links that are non-existing in figure 1.2a: $B - C$ for figure 1.2b and $A - D$ for figure 1.2c. Conversely, note that the set of independencies represented by figure 1.2b cannot be represented precisely by any MN.*

In most cases, we can transform a model into the other by losing independence assumptions (this usually implies that we add links or arcs). As we will see, transforming a BN into a MN is simple: we add an edge for each pair of nodes such that there exists a CPT containing them both, the CPTs are then directly used as factors and the partition function equals 1. The process is important since several inference algorithms reason over MNs. However, transforming a MN into a BN is more difficult and will not be discussed in this thesis (Koller and Friedman, 2009). In the reminder of this chapter, we will set the focus on transforming BNs into MNs.

CPTs are normalized factors, thus they can be used to infer a MN from a BN. Let us consider the BN illustrated in figure 1.2b. It has the following CPTs: $\{P(A)$, $P(B|A)$, $P(C|A)$, $P(D|B, C)\}$. We know that factors in MN are also called clique potentials because all random variables in a same factor form a clique in the MN graph. This is not the case for figure 1.2b since CPT random variables B, C, and D do not form a clique in the BN graph. Given how BNs are defined, such missing only occurs among parents of a node and adding such edges are called moralization[3].

Definition 1.18 (Moral Graph) *The moral graph $\mathcal{M}(\vec{G})$ of a DAG $\vec{G}$ over $\mathbf{X}$ is the undirected graph over $\mathbf{X}$ that contains an undirected edge between X and Y if: (i) either $X \leftarrow Y$ or $X \rightarrow Y$ exists in $\vec{G}$; (ii) if there exist a node Z such that $X \rightarrow Z \leftarrow Y$.*

BN graphs can already be moral, as figure 1.2c for instance. Adding edges to moralize a BN graph obviously erase some independencies, however there exists several properties of moralized graphs.

Property 1.1 *Let $\vec{G}$ be a DAG. The moralized graph $\mathcal{M}(\vec{G})$ is a minimal I-map for $\vec{G}$.*

Proof in Koller and Friedman (2009). This is an unsurprising result since removing any edge that was not added during moralization breaks dependencies existing in the BN graph and removing moral edges would create false independencies.

Property 1.2 *If a DAG $\vec{G}$ is moral, then its moralized graph $\mathcal{M}(\vec{G})$ is a perfect map of $\vec{G}$.*

Proof in Koller and Friedman (2009). This is even a less surprising result since both graphs are identical if we drop directions. We will see in chapter 2 that moralization is the first step in two main inference algorithms: Variable Elimination and Shafer-Shenoy.

[3]It is called moralization since adding edges among unconnected parents *marry* them.

1.5 Modeling using Bayesian Networks

When using BNs to model specific domains, experts usually require specific functionalities. We will now present the most used ones. For BNs, local structures are the different data structures used to represent conditional probability distributions. It is important to differentiate the data structure used for the probabilistic computations from the ones used for modeling purpose. Different data structures induce different costs for constructing a BN. CPTs are a straightforward representation of a discrete conditional probability distribution and are considered as the classic data structure used to encode probabilities in BNs. However, the difficulty to fill CPTs grows exponentially with the number of random variables. Regarding ergonomics, there has been a considerable amount of research carried out by the different software companies selling BN oriented products. However, in our case we are more interested in alternative data structures that require fewer parameters, which help reducing the memory consumption of large CPTs and the number of computations. We will also take a glimpse to deterministic and probabilistic functions that are a must-have feature when dealing with expert knowledge. Such functions span from standard probability distributions to logical operators.

		Z	
X	Y	z	$\bar{z}$
x	y	0.9	0.1
x	$\bar{y}$	0	1
$\bar{x}$	y	0.99	0.01
$\bar{x}$	$\bar{y}$	0.99	0.01

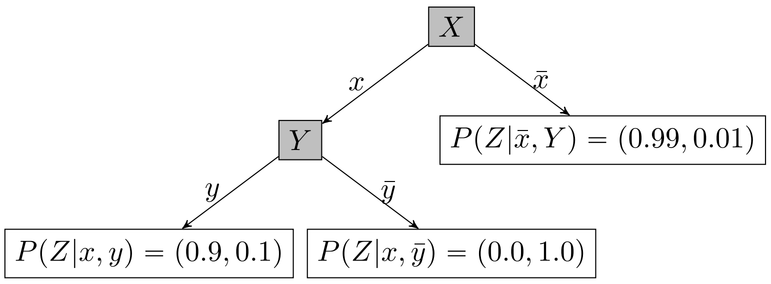

X
x
$\bar{x}$
Y
$P(Z|\bar{x}, Y) = (0.99, 0.01)$
y
$\bar{y}$
$P(Z|x, y) = (0.9, 0.1)$
$P(Z|x, \bar{y}) = (0.0, 1.0)$

Figure 1.3: CPTs do not exhibit local structure present in conditional probability distributions.

Context Specific Independence. Multidimensional tables prevent from exploiting any structure in the conditional probability distribution, i.e., exploiting the fact that conditional probability distributions are constant for different instantiations of random variables. Such independence is called context specific independence and can be represented using trees or rules (Boutilier et al., 1996; Koller and Friedman, 2009). In many cases, context specific independence can be exploited to factorize CPTs. In figure 1.3, we can see that the distributions $P(Z|X = \bar{x}, Y = y)$ and $P(Z|X = \bar{x}, Y = \bar{y})$ are equal, i.e., that $(Z \perp\!\!\!\perp Y|X = \bar{x})$. However, $P(Z|X = x, Y = y)$ and $P(Z|X = x, Y = \bar{y})$ are not equal, thus when $X = x$, Z is dependent of Y. These independencies can be exploited by a tree representation, as in figure 1.3.

Deterministic Functions. When confronted to real world applications, we frequently encounter deterministic relationships, i.e., cases where the state of a random variable is known with no uncertainty if its parents are known. Such variables are called deterministic variables. These deterministic variables are easily identifiable by the fact that probability values are either equal to one or zero. A deterministic function is a triple (X_f, f, ϕ_f) where X_f is a discrete random variable in a BN, f a function such that $f : \pi(X_f) \mapsto \text{Val}(X_f)$ and ϕ_f a CPT that maps $\pi(X_f)$ to zeros or ones. Deterministic functions can be considered observed if all their parents are observed. Classic deterministic functions are logical or, logical and, and logical xor. There is also many specific functions, such K-gates which are true only if at least K parents are true:

$$P(X|\pi(X)) = \begin{cases} 1 & \text{if } |\{Y = True, Y \in \pi(X)\}| \geqslant K, \\ 0 & \text{otherwise.} \end{cases}$$

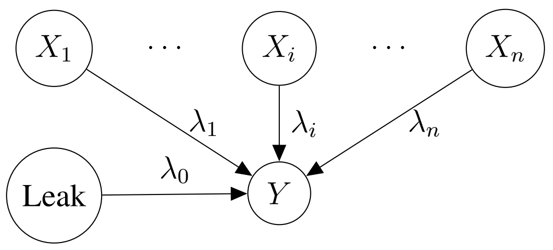


Figure 1.4: A *Noisy-or* gate can be used when causes are independent from each other.

Parametric Distributions Another classic conditional probability distribution variation is to use parametric distributions, i.e., conditional distributions dependent over a parameter usually constant or context dependent. Formally, such parameters are usually considered as observed random variables and thus they do not require a prior distribution, which can be problematic if the parameter is not discrete[4]. Usually, the parameter is directly part of the CPT specification:

X	
True	False
$1 - \lambda$	λ

***Noisy-or*, *Noisy-and* and generalized linear models.** In many situations, a consequence can have multiple independent causes. Such situations are represented using a *Noisy-or* function. Figure **??** illustrates a *Noisy-or* with $n + 1$ parameters where Y and the X_i are binary discrete random variables. Each λ_i is the probability $P(Y = y|X_i = x_i, X_j = \bar{x_j}), j = 1, \cdots, i-1, i+1, \cdots, n$, except for the leak probability, λ_0, representing an unknown cause. Since we suppose each cause to be independent from one

[4]Problematic in a sense that mixing discrete and continuous probability distributions is not a trivial matter (Koller and Friedman, 2009).

another, we can infer the probability distribution $P(Y|X_1, \cdots, X_n)$ using the following equation:

$$P(y|x_1, \cdots, x_n) = (1 - \lambda_0) \prod_{i=1}^{n} (1 - \lambda_i)^{x_i},$$

were we suppose x_i to equal 0 or 1. There have been considerable work on *Noisy-or* and the class of similar functions, such as an axiomatization of the *Noisy-or* and *Noisy-and* (Cozman, 2004; Xiang, 2010). *Noisy-or* are part of a more general class of probabilistic functions called generalized linear models (Koller and Friedman, 2009).

Probabilistic Functions and Discretization. Opposing deterministic functions, probabilistic functions are functions which return probability values different from zeros and ones. They are usually classic discrete probability distributions: the Poisson distribution, the Bernoulli distribution, the binomial distribution, the geometric distribution, and the negative binomial distribution are the most used ones. We can also proceed with the discretization of continuous distributions, since exponential or normal distributions are often useful to model some continuous random variable behavior.

Chapter 2

Probabilistic Inference

Probabilistic inference is a family of algorithms used to compute probability values of the form $P(\mathbf{Q} = \mathbf{q}|\mathbf{E} = \mathbf{e})$, where $\mathbf{Q}$ is a set of random variables called the query and $\mathbf{e}$ is a set of observations of the realizations of some random variables called evidence. $P(\mathbf{Q} = \mathbf{q}|\mathbf{E} = \mathbf{e})$ is the mathematical version of the question "What is the probability of $\mathbf{q}$ knowing $\mathbf{e}$?". Such questions are called conditional probability queries or evidence updating and in most cases we will compute the probability distribution[1] $P(\mathbf{Q}|\mathbf{e})$. Since many inference algorithms designed for BNs transform them into MNs and given the fact that such transformation is trivial, we will usually concern ourselves with inference algorithms designed for MNs. For clarity, we will also qualify probabilistic inference as *plain* inference, as we will only consider probabilistic inference in this thesis. Inference is not only about evidence updating. For some applications, we may want to infer the most probable explanation (MPE), i.e., given some observed variables finding a maximum probability assignment to the rest of them. We could also want to compute the maximum a posteriori (MAP) of a query, i.e., given some evidence, finding an assignment to a subset of hypothesis variables that maximizes their probability. MPE is a special case of MAP. Algorithms for computing MPE are similar to evidence updating. However, those handling MAP are known to be much harder. Another application is, given a utility function, to find an assignment to a subset of decision variables that maximizes the expected utility of the problem (MEU). Yet another method, called sensitivity analysis, studies the probability variations when the prior distribution changes. These inference tasks are outside the scope of this thesis and the reader should refer to (Pearl, 1988; Jordan, 1999; Koller and Friedman, 2009; Darwiche, 2009) for additional materials.

Both exact and approximate inference are NP-Hard problems (Cooper, 1987; Dagum and Luby, 1993). We will not discuss approximate inference and the reader should refer to Koller and Friedman (2009) if needed. As it is often the case with NP-Hard problems, many practical inference applications are tractable. The key parameter in inference complexity is the MN's tree-width. Unfortunately, computing a graph's tree-

[1]We will often drop the notation $P(\mathbf{X} = \mathbf{x})$ in favor of $P(\mathbf{x})$ which is a commonly accepted abuse of notation.

width is also NP-Hard (Arnborg et al., 1987; Robertson and Seymour, 1986; Dechter, 1996). Consequently, it is difficult to estimate the complexity of inference without trying to infer over it. As a general rule, the size of the largest factor is a good tree-width estimator and we will show that it is correlated to the largest clique of the associated MN. Tree-width is not the only parameter that influences inference complexity. Indeed each inference algorithm relies on different NP-Hard problems. Among them, we can find minimal cutsets, optimal elimination orders or graph triangulation (Garey and Johnson, 1979a; Arnborg et al., 1987). Finally, operators over factors (factorization and marginalization) are critical for inference. There have been a large amount of research to implement cost-effective data structures and caching methods for manipulating them (Druzdzel and Suermondt, 1994; Huang and Darwiche, 1996; Shachter, 1998; Arias and Diez, 2007; Koller and Friedman, 2009; Grant, 2010).

The outline of this chapter is the following: we will first discuss and define evidence, as they are central for practical use of inference. Afterwards, we will present the Variable Elimination algorithm and follow up with the transformation of a MN into a junction tree. This allows us to introduce another inference algorithm called the Shafer-Shenoy algorithm. We will conclude by giving references on other inference approaches that are less relevant with this thesis. Finally, Koller and Friedman (2009) was an important source for most definitions and proofs present in this chapter.

2.1 Evidence representation

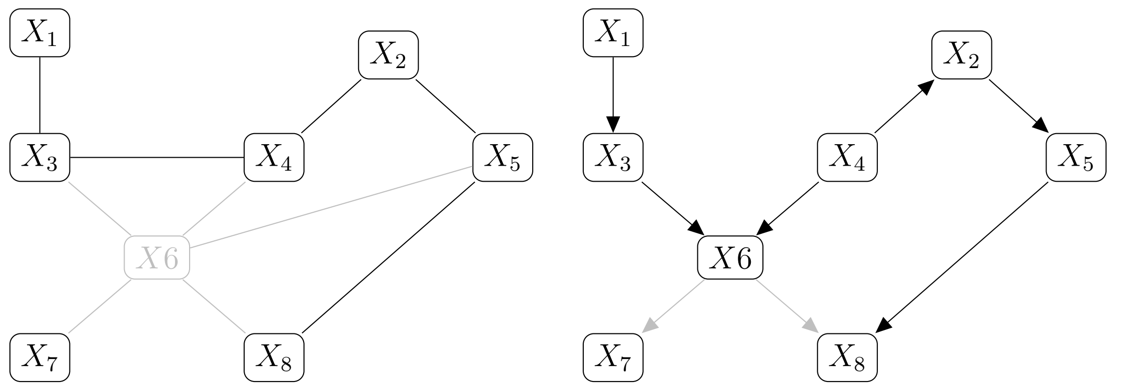
How evidence are represented and used is central for inference. Remarkably, there exist different evidence representations in BNs and MNs. But, before we explain evidence mechanics we will first define what evidence are.

Definition 2.1 (Evidence) *Let X be a random variable, an evidence e is the likelihood measure $e(X) = p(e|X)$.*

We will distinguish hard evidence from soft ones.

Definition 2.2 (Hard and soft evidence) *A hard evidence e over X maps one value of X to 1 and all other values to 0. In all other cases, the evidence is said to be a soft evidence.*

Hard evidence encodes a total knowledge over a random variable's possible outcomes, on the contrary soft evidence represent partial knowledge. Let us consider a hard evidence over a boolean variable X such that $P(X = true) = 0.4$. Stating that $e(X = true) = 1$ is equivalent to stating that $P(X = true|X = true) = 1$. If $P(X = true|X = true) = 1$ we have $P(X = false|X = true) = 0$. The interpretation of the evidence $e(X = true) = 1$ is pretty straightforward. Now let us consider soft evidence with a more complex example. Suppose that H represents some patient's health and that we proceed with a test to diagnose some disease. The test tells us that: (i) there is 60% chance for the test to be positive if the patient is infected; (ii)



(a) In a MN, when a node is observed with hard evidence it *removes* the node from the network.

(b) In a BN, when a node is observed with hard evidence, it *removes* the node's out-going arcs from the network.

Figure 2.1: Evidence does not impact MNs and BNs identically: in gray the nodes, edges and arcs removed from their graph.

there is 0.01% chance for the test to be negative if the patient is healthy. So we have $P(T = positive|H = ill) = 0.6$ and $P(T = negative|H = healthy) = 0.01$. We represent such observation using soft evidence and by the likelihood $e(H = ill) = 0.6)$ and $e(H = healty) = 0.01$. Note that we did not use the conditional probability distribution $P(T|H)$ but only some of its values. We could also have added a node in the network that encodes the conditional probability $P(T|H)$.

Now, let us consider how we can include evidence into MNs andBNs. We have seen that evidence are likelihoods: they define the probability $P(e|X)$ for some random variable X in the network. Adding such evidence in MNs simply adds the factor $e(X)$ in the pool of factors of the MN. For BNs we can add a new node e as a child of the observed node X, and assign the conditional probability distribution $P(e|X)$. When random variables are observed with hard evidence another approach exists: if we are certain about the variable's outcome we can update the probability distribution accordingly. To do so we update each factor containing the observed variable by summing the variable out after taking into account evidence. For MNs this results in projecting each factor ϕ containing the observed variable over its sub-factor $\phi' = \sum_X \phi \times e$, where X is the observed variable and e the factor encoding evidence over X. Figure 2.1a illustrates the consequence of removing a variable with hard evidence: all edges among the node and its neighbors are removed. The same operation can be done in BNs, but its graphical interpretation is different. Let X be the observed node, and $\text{Ch}(X)$ its children. Then all arcs $X \rightarrow Y$, $Y \in \text{Ch}(X)$ are removed from the BN's DAG. However, arcs $Z \rightarrow X$, $Z \in \pi(X)$ are left and so is node X. Indeed, to prevent from mixing directed and undirected arcs, node X is not removed from the BN's DAG and its conditional probability distribution is replaced by $P'(X|\pi(X)) = P(X|\pi(X)) \times e(X)$.

In most applications, we will use hard evidence. Thus we must choose between the two possible representations. Removing observed variables reduces the network's complexity, but each time a node is observed we must cache all concerned factors before applying any update. If the number of observed variables is high, caching the tables can consume large amounts of memory. On the other hand, the first solution offers a more general framework for handling evidence and is also less invasive, i.e., there is no need to alter the network which can be cumbersome in some implementations.

2.2 Variable Elimination

The inference algorithms we will present in this chapter are both applied on MNs. Using MN inference algorithm for inference in BNs is unproblematic since BNs can be transformed trivially into MNs. In this section, we will present Variable Elimination (VE), a probabilistic inference algorithm that can be used for evidence updating in BNs and MNs (Zhang and Poole, 1994, 1996; Dechter, 1999). VE is remarkable both for its simplicity and its efficiency. To understand its principle, we must first recall how random variables can be eliminated from joint probability distributions:

$$P(\mathbf{Q}|\mathbf{E}=\mathbf{e}) = \frac{\sum_{\mathbf{X}} P(\mathbf{Q},\mathbf{X},\mathbf{E}=\mathbf{e})}{P(\mathbf{E}=\mathbf{e})}.$$

The issue with this straightforward approach is that it consumes too much memory and take too much computational time. VE exploits the factorization from the joint probability distribution and the fact that eliminating a random variable in such factorization often requires to consider only a subset of factors. this entails a substantial speed gain and preserves the low memory consumption resulting of the joint probability distribution factorization.

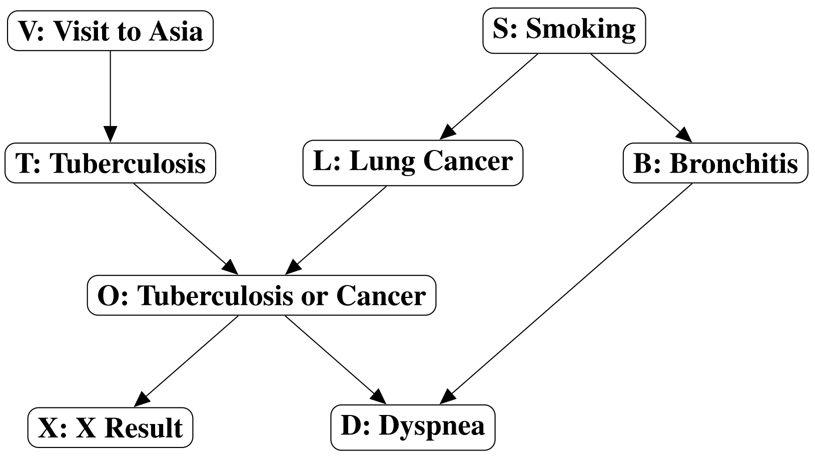


Figure 2.2: The Asia BN.

Example 2.1 *Figure 2.2 illustrates the Asia BN (Lauritzen and Spiegelhalter, 1988). The joint probability distribution over the random variables of this example can be* *factorized as follow:*

$$P(V,T,L,B,O,X,D,S) = P(V) \times P(S) \times P(T|V) \times P(L|S) \times P(B|S) \\ \times P(O|T,L) \times P(X|O) \times P(D|O,B).$$

If we want to eliminate S, we only need to consider a small subset of factors: those that have S in their scope. Thus, we need to factorize the following factors:

$$\phi(S,L,B) = P(S) \times P(L|S) \times P(B|S).$$

We can then sum out S from the resulting factor and pick another candidate for elimination:

$$P(V,T,L,B,O,X,D) = P(V) \times P(T|V) \times P(O|T,L) \times P(X|O) \\ \times P(D|O,B) \times \sum_{S} \phi(S,L,B).$$

The order in which variables are eliminated is critical for VE performances. Finding a good elimination order is a difficult task and is closely related to triangulation (see chapter 2.3). Finding an optimal elimination order is a NP-Hard problem (Kjærulff, 1990; Bodlaender, 1993).

Example 2.2 *Let us consider two possible elimination order: $\mu = (O, T, B, L, S, D, V)$ and $\nu = (V, S, T, L, B, O, D)$. Both elimination orders μ and ν eliminate all variables except X. Thus, they illustrate two elimination orders that compute the same query, here $P(X)$. Table 2.1 illustrates the factors created by both elimination orders. Each created factor is the result of factorizing all factors containing the eliminated variable. We can see that elimination order μ creates factors far bigger than elimination order ν. Such sizes impact memory consumption and the time required by computations.*

μ	Created factors	Size
O	$\phi_1(T,L,B,O,X,D)$	36
T	$\phi_2(V,T,L,B,X,D)$	36
B	$\phi_3(V,S,L,B,X,D)$	36
L	$\phi_4(V,S,L,X,D)$	32
S	$\phi_5(V,S,X,D)$	16
D	$\phi_6(V,X,D)$	8
V	$\phi_7(V,X)$	4

ν	Created factors	Size
V	$\phi_1(V,T)$	4
S	$\phi_2(S,L,B)$	8
T	$\phi_3(T,L,O)$	8
L	$\phi_4(L,B,O)$	8
B	$\phi_5(B,O,D)$	8
O	$\phi_6(O,X,D)$	8
D	$\phi_7(X,D)$	4

Table 2.1: Factors created using VE and different elimination orders.

The algorithm

Algorithm 1 inputs are a set of factors Φ, a set of queries $\mathbf{Q}$, a set of evidence $\mathbf{e}$, and an elimination order t over variables in Φ. We denote by Scope(ϕ) the scope of factor ϕ and by Scope(Φ) the union $\bigcup_{\phi \in \Phi}$ Scope(ϕ). In this version of VE we chose to represent evidence as factors over the observed random variables. Algorithm 1 first step is to eliminate each random variable X, following the order defined by t (lines 1-12). Eliminating a variable is done by multiplying together each factor ϕ_X such that $X \in Scope(\phi_X)$ (lines 3-7), by multiplying evidence factors over X if necessary (lines 8-9) and by summing out X of the resulting factor (line 10). Φ is updated by replacing all factors in Φ_X by ϕ_X (lines 11-12). Then, VE factorizes all remaining factors into ϕ_Q (note that after eliminating all random variables $X \notin \mathbf{Q}$ we have Scope$(\Phi) = \mathbf{Q}$). The factor is then returned after normalization (lines 13-18).

Algorithm 1: Variable Elimination

```
   Input: $\Phi$: a set of factors, $\mathbf{Q}$: a set of queries, $\mathbf{e}$: a set of evidence, $t$: an
          elimination order
   Output: $\phi_{\mathbf{Q}}$: a factor over $\mathbf{Q}$ encoding $P(\mathbf{Q}|\mathbf{e})$
 1 foreach $X \in t$ do
 2     if $X \notin \mathbf{Q}$ then
 3         let $\Phi_X$ be a subset of $\Phi$ such that $\phi_X \in \Phi_X$ iff $X \in$ Scope$(\phi_X)$;
 4         Scope$(\phi)$ = Scope$(\Phi_X)$;
 5         $\phi = \mathbb{1}$;
 6         foreach $\phi_X \in \Phi_X$ do
 7             $\phi = \phi \times \phi_X$;
 8         if $\exists e \in \mathbf{e}$ such that $X \in Scope(e)$ then
 9             $\phi = \phi \times e$;
10         $\phi = \sum_X \phi$;
11         $\Phi = \Phi \backslash \Phi_X$;
12         $\Phi = \Phi \cup \{\phi\}$;
13 Scope$(\phi_{\mathbf{Q}}) = \mathbf{Q}$;
14 $\phi_{\mathbf{Q}} = \mathbb{1}$;
15 foreach $\phi \in \Phi$ do
16     $\phi_{\mathbf{Q}} = \phi_{\mathbf{Q}} \times \phi$;
17 $normalize(\phi_{\mathbf{Q}})$;
18 return $\phi_{\mathbf{Q}}$;
```

Now, let us analyze Algorithm 1 complexity. VE proceeds with the elimination of variables by factorization and marginalization. Initially, we have n factors and during the elimination process m factors will be created. At step i, all factors containing variable X_i are factorized in one factor ψ_i. Then, X_i is summed out of ψ_i resulting in factor ϕ_i. Let C_i be the number of entries of ψ_i and $C_{max} = max_i C_i$. Then the cost of creating all ψ_i is at most to $\sum_{i=1}^{n+m} C_i \leqslant (n+m) \times max_{1 \leqslant i \leqslant n}(C_i) = O(nC_{max})$. Creating factor ϕ_i requires to process all entries of factor ψ_i, thus it costs exactly C_i. Consequently, the complexity of VE is in $O(nC_{max})$. Since, a factor's size is exponential in the number of variables it contains, if ψ_i contains k_i variables of at most size v, then $C_i \leqslant v^{k_i}$. If $k_{max} = max_i|Scope(\psi_i)|$, then VE complexity is $O(n^{k_{max}})$. In other words, the complexity of probabilistic inference using VE is proportional to the largest factor's size induced by the elimination order.

Variable Elimination optimizations

Example 2.3 *Consider that the Asia BN's variables are not observed and suppose we eliminate variable D first. The only factor over D is $P(D|O,B)$ and suppose that its values are:*

O	B	$D = present$	$D = absent$
true	*present*	*0.90*	*0.10*
true	*absent*	*0.70*	*0.30*
false	*present*	*0.80*	*0.20*
false	*absent*	*0.10*	*0.90*

What values will take factor $\phi(O,B) = \sum_D P(D|O,B)$? If D is not observed, it will produce a factor filled with ones:

O	B	$\phi(O,B)$
true	*present*	$0.90 + 0.10 = 1.0$
true	*absent*	$0.70 + 0.30 = 1.0$
false	*present*	$0.80 + 0.20 = 1.0$
false	*absent*	$0.10 + 0.90 = 1.0$

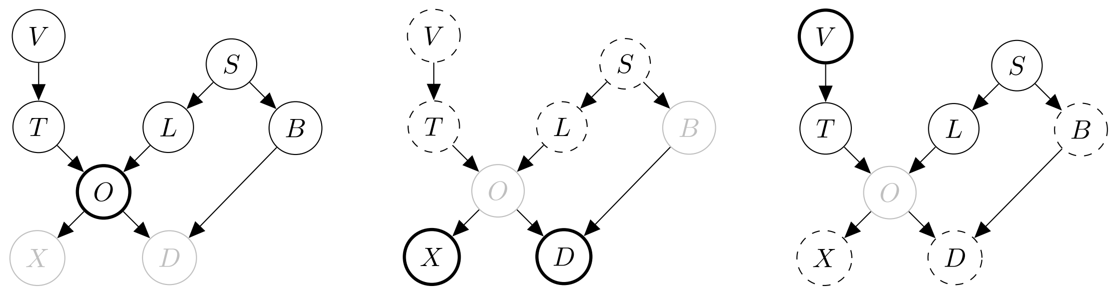
Now if have the evidence that $D = present$, we must add a factor over D such that $e(D = present) = 1.0$ and $e(D = absent) = 0.0$. Consequently, $\phi(O,B) = \sum_D P(D|O,B) \times e(D)$ is no longer filled with ones:

O	B	$\phi(O,B)$
true	*present*	$1.0 \times 0.90 + 0.0 \times 0.10 = 0.90$
true	*absent*	$1.0 \times 0.70 + 0.0 \times 0.30 = 0.70$
false	*present*	$1.0 \times 0.80 + 0.0 \times 0.20 = 0.80$
false	*absent*	$1.0 \times 0.10 + 0.0 \times 0.90 = 0.10$

Variables such as D are called barren nodes and can be discarded when they are not observed.

Definition 2.3 (Barren random variables) *Let $\mathcal{B}$ a BN, $\mathbf{X}$ the set of random variables of $\mathcal{B}$, $\mathbf{Q} \subseteq \mathbf{X}$ a set of queried random variables and $\mathbf{e}$ a set of evidence over variables in $\mathbf{X} \backslash \mathbf{Q}$. A random variable $X \in \mathbf{X}$ is said to be a barren random variable if X is a leaf, $X \notin \mathbf{Q}$ and $X \notin \mathbf{e}$.*

Barren random variables are one among many properties that can be exploited to speed-up inference algorithms. Here, discarding barren variables consists in ignoring variables that do not provide any information. However, barren nodes are not the only nodes that can be discarded, sometimes only a small part of a BN is required to answer certain queries. This is due to a simple fact: if a node X has only children that are barren nodes, then it is also a barren node, i.e., its elimination will not affect our beliefs about other nodes in the network. The BayesBall (BB) algorithm is a polynomial algorithm that finds the set of required random variables to answer a given query with respect to a set of queried variables and evidence (Shachter, 1998). BB is a simple and effective algorithm, easily adaptable to VE.



(a) Node O is queried, X and D are observed. Here all nodes are required.

(b) X and D are queried. O is observed. Here only X, D, B and O are required.

(c) V is queried and O is observed. Here V, T, S, L and O are required.

Figure 2.3: With respect to different queries (solid nodes) and observations (dotted nodes), the set of required nodes (black nodes) may vary drastically.

BB principle is simple: starting from the queried nodes, the algorithm parses the graph looking for required nodes, i.e., it discards nodes that are not relevant with respect to the query. Two parameters are considered when the algorithm iterates over a node: if the node was reached either by a parent or child and if it is observed or not. When a node is reached by one of its children, if it is not observed we mark the node *on the top and on the bottom* and its parents and children will be visited. If the node is reached from one of its parents, if it is observed we mark it *on the top* and only its parents will be visited. If the node is not observed, we mark it *on the bottom* and only its children will be visited. The different marks prevent from reconsidering a node twice (*bottom* marked) and to distinguish required nodes (*top* marked) from unwanted ones (*top* unmarked). From there it is simple to adapt VE to exploit BB: (i) reduce the CPT of any node with an observed parent accordingly; (ii) add each required node's CPT to the set of initial factors; and (iii) evidence factors for any observed node that is also required. Figure 2.3 illustrates three different configurations of queries and evidence for the Asia BN.

2.3 From Variable Elimination to junction trees inference algorithms

Remarkably, VE (Zhang and Poole, 1994, 1996; Dechter, 1999) is *younger* than junction tree based algorithms (Lauritzen and Spiegelhalter, 1988; Shenoy and Shafer, 1990; Jensen et al., 1990). We chose to present VE first, because it offers a more intuitive grasp of probabilistic inference. Moreover, its simplicity makes it a good introduction to inference in PGMs. In fact, VE was introduced as a simplification of junction tree based algorithms. However, it's more appropriate to present junction tree based algorithms as a sophistication of VE. To understand the link between VE and junction tree based algorithms, we must first discuss triangulation and its importance for both inference approaches.

Variable Elimination and graph triangulation

We said that the elimination orders required by VE are closely related to graph triangulation. We will explain how these two problems are connected. First, let use define chordal graphs.

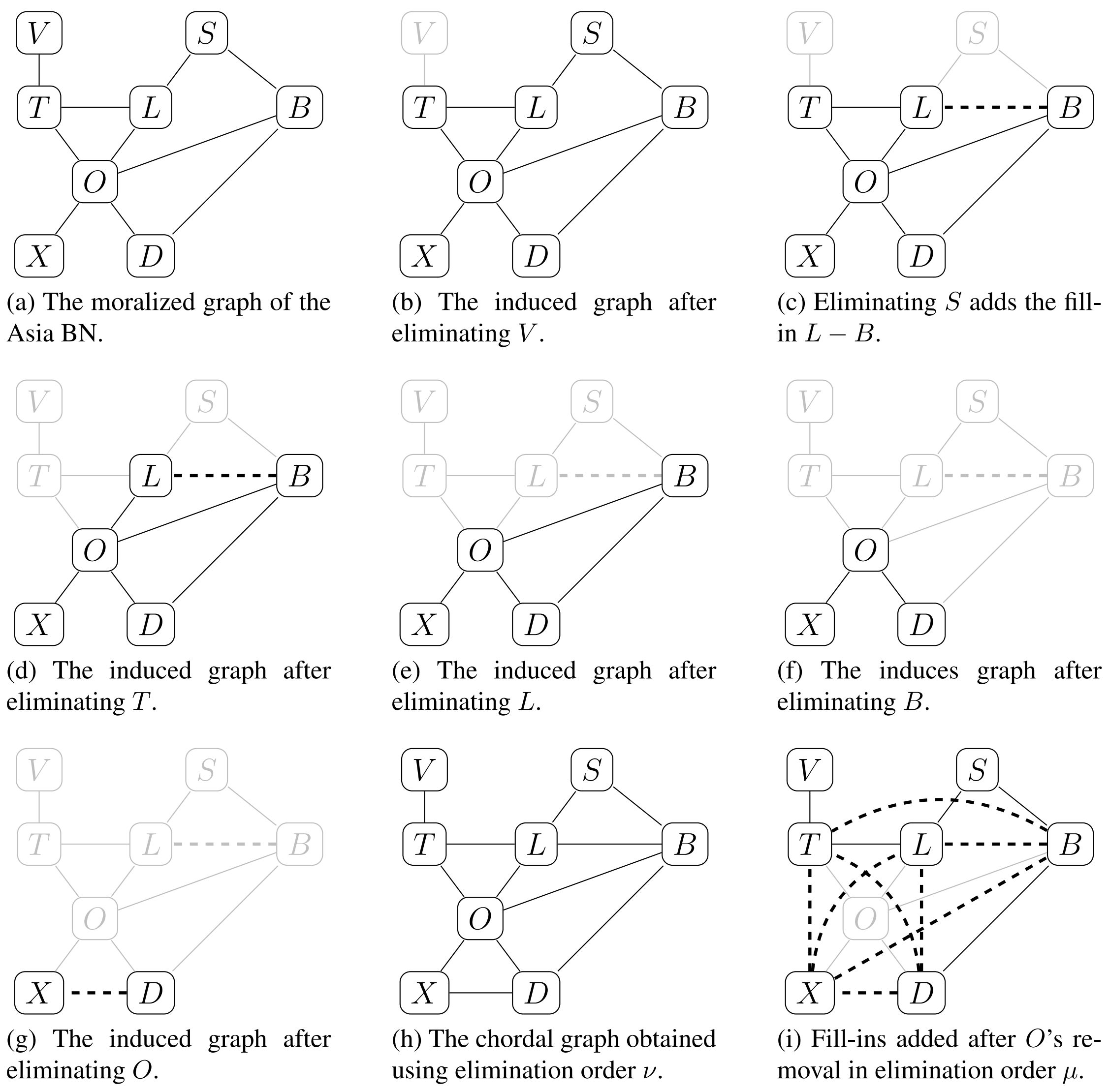
Definition 2.4 (Chordal graph) *Let $G = (V, E)$ be a undirected graph. G is chordal if all cycles of length $n > 3$ have a chord, i.e., for all cycles $l = X_1 - \cdots - X_n$ ($n > 3$) two nodes X_i and X_j exist in l such that the edge $X_i - X_j$ exists in E but not in l. Transforming a graph G into a chordal graph is called triangulating G.*

A naive graph triangulation algorithm is to randomly choose a node, remove it and connect all its neighbors. Edges added to connect neighbors after a node's elimination are called *fill-ins*. Eliminating a node and connecting its neighbors is the graphical representation of eliminating a random variable using VE. To understand why, let us define the induced graph of a set of factors.

Definition 2.5 (Induced Graph) *Let Φ be a set of factors. The induced graph of Φ is the undirected graph $G = \{V, E\}$, with $V = Scope(\Phi)$ and $(X, Y) \in E$ if and only if $\exists\ \phi \in \Phi$ such that $\{X, Y\} \subseteq Scope(\phi)$.*

Example 2.4 *Let us consider the Asia BN and elimination order ν from table 2.1. Figure 2.4b to figure 2.4g illustrate the induced graphs obtained after each step of VE using elimination order ν. We can remark that eliminating S and O respectively add fill-ins $L - B$ and $X - D$ (figure 2.4c and 2.4g). By adding these two fill-ins we obtain the chordal graph illustrated in figure 2.4h. Finally, figure 2.4i illustrates the induced graph after eliminating O using elimination order μ. We can clearly see that eliminating O first adds unnecessary fill-ins.*

The number of fill-ins added by an elimination is a good estimator of the eliminations quality. Ideally, we want to minimize that number, but finding minimal triangulations is NP-Hard. Consequently, we rely on approximate algorithms based on different



(a) The moralized graph of the Asia BN.

(b) The induced graph after eliminating V.

(c) Eliminating S adds the fill-in $L - B$.

(d) The induced graph after eliminating T.

(e) The induced graph after eliminating L.

(f) The induces graph after eliminating B.

(g) The induced graph after eliminating O.

(h) The chordal graph obtained using elimination order ν.

(i) Fill-ins added after O's removal in elimination order μ.

Figure 2.4: Eliminating variables one by one is equivalent to triangulating the BN's moralized graph.

scoring heuristics (Kjærulff, 1990; Bodlaender, 1993). Note that the NP-hard aspect of finding minimal triangulation is independent of the NP-hardness of inference. Thus, even if we have the optimal triangulation, we do not have the guarantee that inference will be tractable. Regarding triangulation, there has been a considerable number of contributions (Rose, 1970; Robertson and Seymour, 1986; Kjærulff, 1990; Bodlaender, 1993; Koller and Friedman, 2009). We will use a standard algorithm that finds good results while being fast, i.e., its execution time is negligible compared to inference time (for challenging networks). This algorithm iteratively eliminates nodes using a greedy search based on a score function. At each step the algorithm randomly chooses a node among all nodes with identical scores and eliminates it (Fishelson and Geiger, 2004;

Koller and Friedman, 2009). Classic score functions are:

- Min-neighbors: the score is equal to the number of neighbors;
- Min-weight: the score is the product of each neighbors domain size;
- Min-fill: the score equals the number of fill-ins added by eliminating that node.

We can also aggregate these functions in different manners. Depending on the instance difficulty, we may want to run several instances of the greedy search algorithm using different score functions to obtain the best elimination order.

Algorithm 2: Greedy Search for a Variable Elimination Ordering

```
  Input: $\mathcal{H} = (V, E)$: an undirected graph, $s$: a score function
  Output: $t$: a list of nodes in $V$
1 $t$: an empty list of nodes;
2 while $V$ is not empty do
3     Let $\mathbf{X}$ be the set of nodes that minimize $s$;
4     Randomly choose a node $X \in \mathbf{X}$;
5     foreach $(Y, Z) \in Nghbrs_X \times Nghbrs_X$ do
6         if $Y \neq Z$ and $(Y, Z) \notin E$ then
7             add edge $Y - Z$ to $E$;
8     Remove $X$ from $V$ and all edges $(X, Y) \in E$ with $Y \in V$;
9     Insert $X$ at the end of $t$;
```

Let us analyze the complexity of algorithm 2. Consider step i of the construction of the elimination ordering:i variables have been eliminated and there are $n - i$ remaining variables ($|\mathcal{X}| = n$). We will analyze algorithm 2 complexity in the number of calls to the score function s. At each step we must recompute scores for each node and since one of them is eliminated at each step, there is $\sum_{i=1}^{n}(n-i) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = O(n^2)$ calls to the score function.

Caching Variable Elimination computations

One of VE's particularity is its query oriented aspect. Indeed, a given elimination order will always compute the same query. If we consider elimination order ν from table 2.1, we can see that permuting variables X and D does not change the seven first steps of elimination order ν. The fact that most computations used to compute $P(X|\mathrm{e})$ are also used to compute $P(D|\mathrm{e})$ can be exploited to speed-up the computation of $P(D|\mathrm{e})$ if $P(X|\mathrm{e})$ has already been answered. This is the main purpose of junction tree based algorithms.

To understand how computations can be reused we will consider the elimination order ν of table 2.1: $\{V, S, T, L, B, O, D\}$ that computes $P(X|\mathrm{e})$. Figure 2.5 illustrates how we can represent each created factor by a specific node and connect them such that

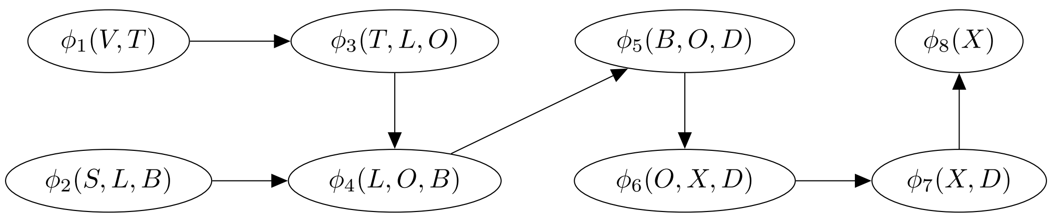


Figure 2.5: Each node represents a factor created using VE over elimination order ν. If an edge $\phi_i \rightarrow \phi_j$ exists then ϕ_i was used to compute ϕ_j.

edge $\phi_i \rightarrow \phi_j$ indicated that ϕ_i was used to compute ϕ_j. Now let us consider query $P(O|\mathbf{e})$ and an elimination order, call it ν', almost identical to ν: $\{V, S, T, L, B, D, X\}$. A subset of the factors created by elimination order ν' are identical to the ones created by ν, as illustrated by figure 2.6.

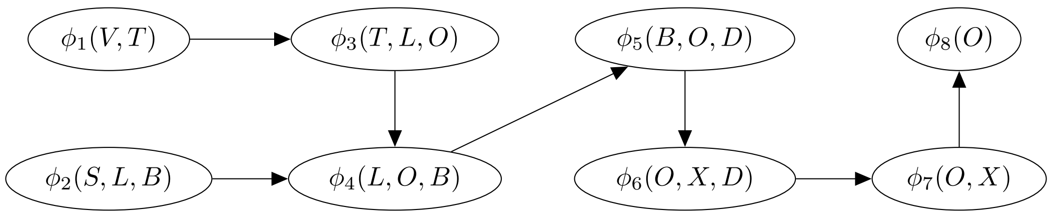


Figure 2.6: Computing $P(O|\mathbf{e})$ can be done using almost all factors created while computing $P(X|\mathbf{e})$.

Furthermore, we can note that each created factor is defined over a clique in the chordal graph. Cliques are used in many graph-based problems and have been studied from every angle (Luce and Perry, 1949; Karp, 1972).

Definition 2.6 (Clique) *Given a graph G, a clique* **C** *of G is any subgraph of G that is complete, i.e., all nodes in the clique are pairwise connected.*

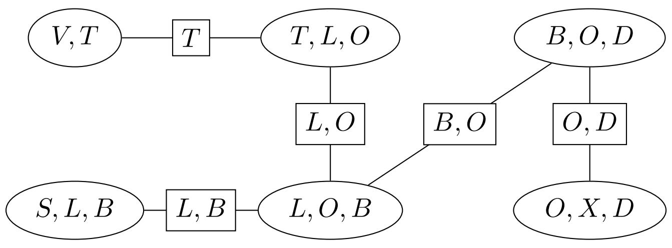


Figure 2.7: A junction tree built from the chordal graph of figure 2.4h.

Using definition 2.6 we can describe figures 2.5 and 2.6 using a junction tree. Figure 2.7 illustrates the junction tree in which nodes represent cliques in the chordal graph of figure 2.4h and two cliques are connected with respect to the factors dependencies. Junction trees are a very specific data structure. We will describe how junction trees are constructed and the Shafer-Shenoy algorithm using junction trees for probabilistic inference in the next section.

2.4 The Shafer-Shenoy inference algorithm

The Shafer-Shenoy (SS) inference algorithm transforms MNs into junction trees and computes marginal probabilities using a message passing scheme (Shenoy and Shafer, 1990). In section 2.3 we have detailed triangulation and gave a hint on how junction trees cache computations. We will now present formally junction trees construction steps, present the SS inference algorithm and we will explain the importance of the running intersection property.

Building the junction tree

The first step in building a junction tree is to triangulate the MN's graph. We have seen in section 2.3 that eliminating random variables one-by-one adds fill-ins, i.e., eliminating a random variable will produce a new factor that links random variables that were not originally connected. When we triangulate a graph, we add fill-ins and by doing so we *fix* the elimination order: fill-ins are the graphical representation of intermediate factors created by successively eliminating random variables. Then cliques in the triangulated graph represent any intermediate factor created by VE and can be represented using a clique graph.

Definition 2.7 (Clique Graph) *Let G be an undirected graph. A clique graph $\mathcal{U}$ of G is an undirected graph with a node for each clique $\mathbf{C}_i$ in G if and only if there is no other clique $\mathbf{C}_j$ in G such that $\mathbf{C}_i \subseteq \mathbf{C}_j$. An edge $\mathbf{C}_i - \mathbf{C}_j$ exists in $\mathcal{U}$ only if $\mathbf{C}_i \cap \mathbf{C}_j \neq \emptyset$. Conversely, if $\mathbf{C}_i \cap \mathbf{C}_j \neq \emptyset$, then there exists a path linking $\mathbf{C}_i$ and $\mathbf{C}_j$ in $\mathcal{U}$.*

Junction tree based algorithms goal is to compute the probability $P(C_i)$ for each clique of a junction tree. Consequently, there is no need to represent a clique $\mathbf{C}_i$ if there exists a clique $\mathbf{C}_j$ such that $\mathbf{C}_j \subseteq \mathbf{C}_i$. For example, figure 2.7 is the clique graph of figure 2.6: cliques $\{O\}$ and $\{O, X\}$ have been removed since both of them are included in clique $\{O, X, D\}$. Note that clique graphs are hypergraphs, thus cliques are set of nodes and edges connecting two cliques $\mathbf{C}_i$ and $\mathbf{C}_j$ show that there exist edges between any pair of nodes in $\mathbf{C}_i \cap \mathbf{C}_j$. We can incorporate the clique graph construction in algorithm 2: each time a node X is eliminated, the clique $\mathbf{C}_X = \{X\} \cup \text{Nghbrs}_X$ is added to the clique graph if there is no clique $\mathbf{C}$ such that $\mathbf{C}_X \subseteq \mathbf{C}$. If $\mathbf{C}_X$ is added to the clique graph, edges are added between $\mathbf{C}_X$ and other cliques with respect to definition 2.7. Not all clique graphs are adapted to inference (explications will be given at the end of this section) and to obtain a suitable data structure for inference, we must extract a specific clique tree called a junction tree. Not all clique trees are junction trees, since junction trees must satisfy an important property called the running intersection property (note that clique graphs with cycles can satisfy the running intersection property).

Definition 2.8 (Running Intersection Property) *Let $\mathcal{T} = (V, E)$ be a clique tree over a set of cliques V and a set of edges E called separators E. We say that $\mathcal{T}$ has the running intersection property if, whenever there is a variable X such that $X \in \mathbf{C}_i$ and*

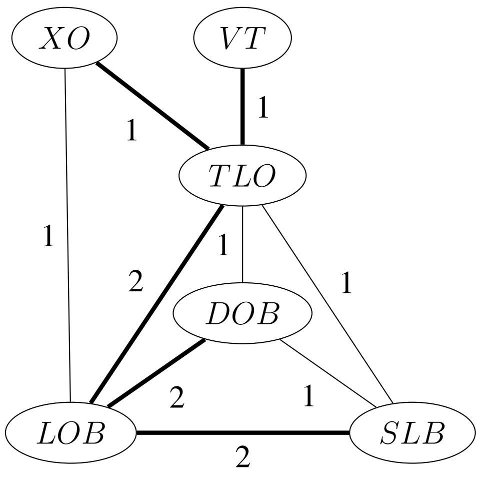
$X \in \mathbf{C}_j$, then X is also in every clique in the unique path in $\mathcal{T}$ between $\mathbf{C}_i$ and $\mathbf{C}_j$. A clique tree with the running intersection property is called a junction tree.

The running intersection property ensures the correctness of junction tree based inference algorithms. To understand why, we must first detail the SS algorithm. Given definition 2.8, the definition of a junction tree is straightforward.

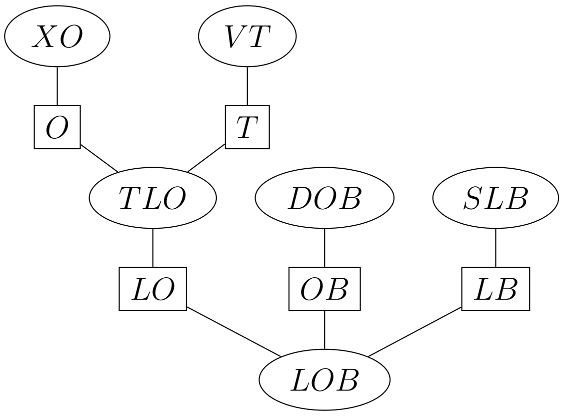
Definition 2.9 *A junction tree is a clique tree verifying the running intersection property.*

To extract a junction tree a possible solution is to use a maximum spanning tree algorithm over a weighted clique graph. A weighted clique graph is a clique graph for which each edge $\mathbf{C}_i - \mathbf{C}_j$ has the weight $|\mathbf{C}_i \cap \mathbf{C}_j|$ (see figure 2.8a).

Theorem 2.1 *A clique tree obtained by applying a maximum spanning tree algorithm on a clique graph will have the running intersection property.*



(a) A clique graph derived from the chordal graph of figure 2.4h. Each clique is a clique in figure 2.4h.



(b) The junction tree derived from figure 2.8a. Circle nodes are cliques and square nodes are separators $S_{ij} = C_i \cap C_j$.

Figure 2.8: A junction tree is obtained from the clique graph of a chordal graph.

Proof in Shibata (1988); Jensen (1988). Figure 2.8a is the clique graph built using the chordal graph from figure 2.4h. Thick edges represent those picked by a maximum spanning tree algorithm and figure 2.8b illustrates the junction tree built using them.

The algorithm

The SS algorithm can be decomposed in three steps: (i) cliques initialization, (ii) collect phase and (iii) diffuse phase. We denote by ϕ_i the factor associated with clique $\mathbf{C}_i$ such that Scope$(\phi_i) = \mathbf{C}_i$ and by ϕ_{ij} the factor associated with the separator $\mathbf{S}_{ij} = \mathbf{C}_i \cap \mathbf{C}_j$ such that Scope$(\phi_{ij}) = \mathbf{S}_{ij}$. We differentiate $\mathbf{S}_{ij}$ (resp. ϕ_{ij}) from $\mathbf{S}_{ji}$ (resp. from ϕ_{ji}).

Definition 2.10 (Valid clique assignment) *Let $\mathcal{M}$ be a MN and Φ be the factors associated with $\mathcal{M}$. Given a junction tree $\mathcal{T}$ constructed from $\mathcal{M}$, a clique $\mathbf{C} \in \mathcal{T}$ is a valid assignment for a factor $\phi \in \Phi$ if and only if Scope$(\phi) \subseteq \mathbf{C}$.*

The first step assigns a factor to a unique clique in the junction tree. Each factor ϕ_i is initialized as follow: $\phi_i = \mathbb{1} \times \phi_{X_1} \times \cdots \times \phi_{X_k}$, where $\{\phi_{X_1}, \cdots, \phi_{X_k}\}$ are the factors assigned to $\mathbf{C}_i$. Evidence are also assigned using definition 2.10.

Example 2.5 *In figure 2.8b, cliques are initialized as follows:*

- $XO \leftarrow P(X|O)$;
- $VT \leftarrow P(V) \times P(T|V)$;
- $TLO \leftarrow P(O|T, L)$;
- $DOB \leftarrow P(D|O, B)$;
- $SLB \leftarrow P(S) \times P(L|S) \times P(B|S)$;
- $LOB \leftarrow \mathbb{1}$.

The following rule explains under what conditions a clique $\mathbf{C}_i$ can send a message (either during the collect or diffuse phase) to a clique $\mathbf{C}_j$.

Rule 2.1 *Let $\mathbf{C}_i$ and $\mathbf{C}_j$ two cliques in a junction tree $\mathcal{T}$ such that $\mathbf{C}_i$ and $\mathbf{C}_j$ are neighbors. $\mathbf{C}_i$ can send a message to $\mathbf{C}_j$ if and only if it has received messages from all its other neighbors Nghbrs$_{\mathbf{C}_i} \backslash \{\mathbf{C}_j\}$.*

SS message passing scheme starts from an arbitrary clique, called the root clique. The first messages to be sent are called collect messages because they are sent from leaf cliques to the root clique. At the end of the collect step, the root clique received messages from all its neighbors. It can then send a message to each of its neighbors during the second step. These messages going from the root clique to the leaves are called diffuse messages. From a probabilistic perspective, at the end of the collect step we can update the root clique's factor so that it encodes the probability $P(\mathbf{C}_{root}, \mathbf{e})$, where $\mathbf{e}$ is a set of evidence. At the end of the diffuse step, each clique's factor can be updated to encode the probability $P(\mathbf{C}_i, \mathbf{e})$.

Algorithm 3 takes as inputs a set of factors, including evidence. Algorithm 3 uses several tables to store factors. Table $factor$ is used to store the initial cliques factors and factors associated with separators. Algorithm 3 results are stored in table $factor$, each factor encoding the probability $P(\mathbf{C}, \mathbf{e})$, where $\mathbf{C}$ is a clique in $\mathcal{T}$. Algorithm 3 starts with initializing each clique's factor (lines 1-5). Then, a clique is chosen as the root clique (any clique can do) and the collect phase is started (lines 6-10). When the collect phase has ended, the diffuse step is started (lines 11-16).

Procedure Collect applies the collect phase of algorithm 3. Procedure Collect purpose is to send collect messages from leaves to the root clique. Indeed, given rule 2.1, a

Algorithm 3: The Shafer-Shenoy algorithm.

```
Input: $\mathcal{T}$: a junction tree, $\Phi$: a set of factors, $\mathbf{e}$: a set of evidence
1  foreach Clique $\mathbf{C} \in \mathcal{T}$ do
2      $factor[\mathbf{C}] = \mathbb{1}$;
3  foreach Separator $\mathbf{S}_{ij} \in \mathcal{T}$ do
4      $factor[\mathbf{S}_{ij}] = \mathbb{1}$;
5      $factor[\mathbf{S}_{ji}] = \mathbb{1}$;
6  foreach Factor $\phi \in \Phi \cup \mathbf{e}$ do
7      Choose a single clique $\mathbf{C}$ such that Scope$(\phi) \subseteq \mathbf{C}$;
8      $factor[\mathbf{C}] = factor[\mathbf{C}] \times \phi$;
9  Choose a root clique $\mathbf{C}_r$ in $\mathcal{T}$;
10 foreach Neighbor $\mathbf{C}_i$ of $\mathbf{C}_r$ do
11     $Collect(\mathbf{C}_i, \mathbf{C}_r)$;
12 foreach Neighbor $\mathbf{C}_i$ of $\mathbf{C}_r$ do
13     $\phi = factor[\mathbf{C}_r]$;
14     foreach Neighbor $\mathbf{C}_j \neq \mathbf{C}_i$ of $\mathbf{C}_r$ do
15         $\phi = \phi \times factor[\mathbf{S}_{jr}]$;
16     $\mathbf{X}$ = Scope$(\phi) \cap \mathbf{S}_{ri}$;
17     $factor[\mathbf{S}_{ri}] = \sum_{X \in \mathbf{X}} \phi$;
18     $Diffuse(\mathbf{C}_i, \mathbf{C}_r)$;
```

Procedure Collect

```
Input: $\mathbf{C}_r$: a clique, $\mathbf{C}_s$: a clique
1  $\phi = factor[\mathbf{C}_r]$;
2  if $C_r$ is not a leaf then
3      foreach Neighbor $\mathbf{C}_i \neq \mathbf{C}_s$ of $\mathbf{C}_r$ do
4          $Collect(\mathbf{C}_i, \mathbf{C}_r)$;
5          $\phi = \phi \times factor[\mathbf{S}_{ir}]$;
6  $\mathbf{X}$ = Scope$(\phi) \backslash \mathbf{S}_{rs}$;
7  $factor[\mathbf{S}_{rs}] = \sum_{X \in \mathbf{X}} \phi$;
```

clique $\mathbf{C}_i$ can send a message to its neighbor $\mathbf{C}_j$ only if it has received messages from all of its other neighbors. Thus, when no message have been sent, only cliques with a single neighbors can send a message. Procedure Collect first parameter is the clique $\mathbf{C}_r$ that was asked to send a collect message to $\mathbf{C}_s$. The message is a factor over the separator between two cliques. The factor is obtained by factorizing the clique's initial factors with all received messages and then summing out variables that are not in the separator.

Procedure Diffuse

```
    Input: $\mathbf{C}_r$: a clique, $\mathbf{C}_s$: a clique
1   foreach Neighbor $\mathbf{C}_i \neq \mathbf{C}_s$ of $\mathbf{C}_r$ do
2       $\phi = factor[\mathbf{C}_r]$;
3       foreach Neighbor $\mathbf{C}_j \neq \mathbf{C}_i$ of $\mathbf{C}_r$ do
4           $\phi = \phi \times factor[\mathbf{S}_{jr}]$;
5       $\mathbf{X} = \text{Scope}(\phi) \backslash \mathbf{S}_{ri}$;
6       $factor[\mathbf{S}_{ri}] = \sum_{X \in \mathbf{X}} \phi$;
7       $Diffuse(\mathbf{C}_i, \mathbf{C}_r)$;
```

Procedure Diffuse applies the diffuse phase of algorithm 3. Procedure Diffuse purpose is to send messages from the root clique to leaf cliques. After the collect phase, the root clique has received messages from all of its neighbors. Thus it can send messages to each of its neighbors. Procedure Diffuse first parameter is the clique $\mathbf{C}_r$ that received a diffuse message from clique $\mathbf{C}_s$ and must send one to all its other neighbors.

Example 2.6 *In example 2.5 we initialized the cliques of junction tree from figure 2.8b. We will now proceed with the collect phase of algorithm 3 with clique LOB as the root clique.*

1. *$XO \rightarrow TLO$: $\phi_1(O) = \sum_X P(X|O)$;*
2. *$VT \rightarrow TLO$: $\phi_2(T) = \sum_V P(V) \times P(T|V)$;*
3. *$TLO \rightarrow LOB$: $\phi_3(L,O) = \sum_T P(O|T,L) \times \phi_1(O) \times \phi_2(T)$;*
4. *$DOB \rightarrow LOB$: $\phi_4(O,B) = \sum_D P(D|O,B)$;*
5. *$SLB \rightarrow LOB$: $\phi_5(L,B) = \sum_S P(S) \times P(L|S) \times P(B|S)$.*

At the end of the collect phase, clique LOB updates its factor by taking into account all received messages: $\phi_{LOB}(L,O,B) = \phi_3(L,O) \times \phi_4(O,B) \times \phi_5(L,B)$. The resulting factor is a joint probability distribution over random variables L, O and B. The diffuse step proceeds similarly: messages are sent from the root clique LOB to leaf cliques.

6. *$LOB \rightarrow TLO$: $\phi_6(L,O) = \sum_B \mathbb{1} \times \phi_4(O,B) \times \phi_5(L,B)$;*
7. *$TLO \rightarrow XO$: $\phi_7(O) = \sum_{T,L} P(O|T,L) \times \phi_2(T) \times \phi_6(L,O)$;*
8. *$TLO \rightarrow VT$: $\phi_8(T) = \sum_{L,O} P(O|T,L) \times \phi_1(O) \times \phi_6(L,O)$;*
9. *$LOB \rightarrow DOB$: $\phi_9(O,B) = \sum_L \mathbb{1} \times \phi_3(L,O) \times \phi_5(L,B)$;*
10. *$LOB \rightarrow SLB$: $\phi_{10}(L,B) = \sum_O \mathbb{1} \times \phi_3(L,O) \times \phi_4(O,B)$.*

At the end of the diffuse phase, each clique can update its factor to obtain the unnormalized joint probability distribution over the variables in the clique:

- $\phi_{XO}(X,O) = P(X|O) \times \phi_7(O)$;
- $\phi_{VT}(V,T) = P(V) \times P(T|V) \times \phi_8(T)$;
- $\phi_{TLO}(T,L,O) = P(O|T,L) \times \phi_1(O) \times \phi_2(T) \times \phi_6(T,L,O)$;
- $\phi_{DOB}(D,O,B) = P(D|O,B) \times \phi_9(O,B)$;
- $\phi_{SLB}(S,L,B) = P(S) \times P(L|S) \times P(B|S) \times \phi_{10}(L,B)$.

Shafer-Shenoy correctness and complexity

We will discuss here why SS collect and diffuse phases update each clique with the correct joint probability distribution. To do so, we will use the fact that VE and SS are closely related. Indeed a junction tree is a data structure to store intermediate computations performed during VE. In other words, VE can match the computations performed during a collect phase. Each time we project a clique's factor on a separator, at least one random variable is eliminated, thus when we are in a collect phase we continuously repeat a factorization operation (when we factorize a clique's factor with its incoming messages) and a marginalization operation (when we project a clique's factor over one of its separator).

Theorem 2.2 *Let $\mathcal{T}$ be a junction tree constructed from a MN $\mathcal{M} = (P, G)$. After the collect and diffuse phase of algorithm 3, each factor ϕ_i of a clique $\mathbf{C}_i$ obtained by the product of $\mathbf{C}_i$'s initial value and of all of its received messages is an unnormalized measure of the joint probability distribution $P(\mathbf{C}_i, \mathbf{e})$.*

Proof in Shenoy and Shafer (1990). If we consider the number of messages sent during the collect and diffuse steps, we can easily see that there are $2(n-1)$ messages sent: two for each edge in $\mathcal{T}$. Consequently, algorithm 3 complexity is impacted by the size of the largest clique in the junction tree. Thus, algorithm 3 complexity is of the same order of magnitude than algorithm 1 complexity. However, in practice algorithm 3 will be less performing than algorithm 1 for a single query. Indeed, where algorithm 1 can discard factors during the elimination process, algorithm 3 keeps track of all of them. Furthermore, the diffuse phase of algorithm 3 adds unnecessary computations if we are only concerned by a single random variable. But, if we do not change evidence then algorithm 3 is more performing for multiple queries. Ideally, we can imagine an amelioration of algorithm 3 where messages are sent only to update cliques containing the queried random variables, i.e., we only proceed with the collect phase of algorithm 3. Doing so results in applying algorithm 1 and caching intermediate factors.

Understanding the importance of the running intersection property

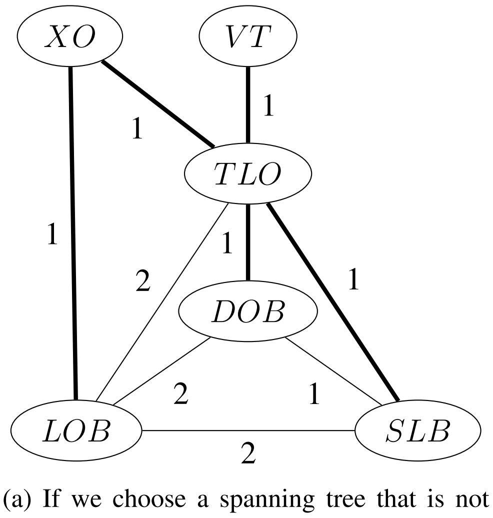
We will explain now why the running intersection property is required for inference using junction trees. Figure 2.9 shows a different spanning tree (figure 2.9a) and its derived clique tree (figure 2.9b) than the ones in figure 2.8. Since the cliques are identical to the junction tree in figure 2.8b we can use the same clique initialization than example 2.5. Let us do a collect phase using clique TLO as root, the messages sent from leaf nodes are:

1. $LBO \rightarrow XO$: $\phi_1(O) = \sum_{LB} \mathbb{1}$;
2. $XO \rightarrow TLO$: $\phi_2(O) = \sum_X P(X|O) \times \phi_1(O)$;
3. $SLB \rightarrow TLO$: $\phi_3(L) = \sum_{SB} P(S) \times P(L|S) \times P(B|S)$;
4. $DOB \rightarrow TLO$: $\phi_4(O) = \sum_{D,B} P(D|O,B)$;
5. $VT \rightarrow TLO$: $\phi_5(T) = \sum_V P(V) \times P(V|T)$.

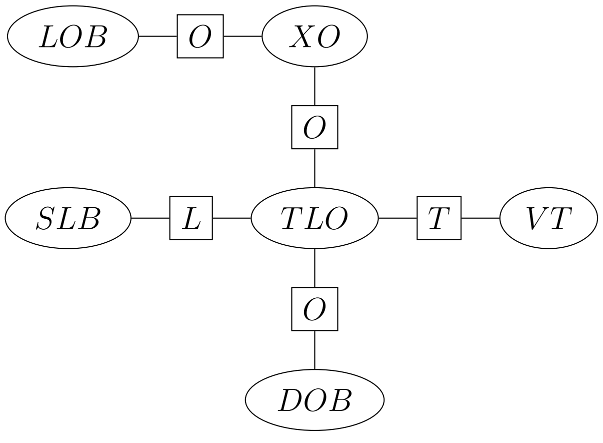
If we look at the messages received by TLO at the end of the collect phase, we obtain the following factorization of the joint probability distribution over T, L and O:

$$\begin{aligned}\phi_{TLO}(T,L,O) &= P(O|T,L) \times \phi_2(O) \times \phi_3(L) \times \phi_4(O) \times \phi_5(T) \\ &= P(O|T,L) \times \big(\sum_X P(X|O)\big) \times \big(\sum_{SB} P(S) \times P(L|S) \times P(B|S)\big) \\ &\quad \times \big(\sum_{D,B} P(D|O,B)\big) \times \big(\sum_V P(V) \times P(V|T)\big). \end{aligned} \tag{2.1}$$

This computation is incorrect, since we summed out twice variable B. Furthermore, evidence updating is not possible using the clique tree of figure 2.9: suppose we add evidence over L in clique LOB, then equation 2.1 will not take it into account. Indeed the evidence on L will be removed when the message from LOB to XO is sent. Let us consider the diffuse phase and focus on clique LOB.



(a) If we choose a spanning tree that is not maximal we will obtain a clique tree that will not have the running intersection property.



(b) A clique tree derived from figure 2.9a. Clearly, the running intersection property does not hold in this graph.

Figure 2.9: A clique tree that is not a junction tree: the running intersection property does not hold.

6. $TLO \to XO$: $\phi_6(O) = \sum_{TL} P(O|T,L) \times \phi_3(L) \times \phi_4(O) \times \phi_5(T)$;
7. $\{XO\} \to \{LOB\}$: $\phi_7(O) = \sum_X P(X|O) \times \phi_6(O)$.

Lets consider the factorization used to compute $\phi_{LOB}(T, L, O)$:

$$\begin{aligned}\phi_{LOB}(L,O,B) &= \mathbb{1} \times \phi_7(O)\\ &= \mathbb{1} \times \Big(\sum_{TL} P(O|T,L) \times \phi_3(L) \times \phi_4(O) \times \phi_5(T)\Big)\\ &= \mathbb{1} \times \Big(\sum_{TL} P(O|T,L) \times \big(\sum_{SB} P(S) \times P(L|S) \times P(B|S)\big)\\ &\quad \times \big(\sum_{D,B} P(D|O,B)\big) \times \big(\sum_{V} P(V) \times P(V|T)\big) \qquad (2.2)\end{aligned}$$

Clearly, the factorization computed in equation 2.2 is erroneous. Indeed, L and B are eliminated whereas they should not have been. If we look at the random variables distributions we see that the running intersection property is not satisfied, for example variable B is not present in cliques XO and TLO, yet it is in cliques LOB and DOB. The absence of B in the path between LOB and DOB explains its early elimination in equation 2.2 and the running intersection property purpose is to prevent such errors.

Shafer-Shenoy in clique graphs

To conclude our presentation of the SS algorithm, we will explain why this inference algorithm must be applied to junction trees only. Indeed, the reason why we cannot use a general clique graph as in figure 2.9a is that SS uses local computations. The concept of local computations first appeared in the initial work of Pearl on BNs and is at the core of the Belief Propagation algorithm (Pearl, 1982). The idea is to only rely on node's neighbors to update the belief of the nodes and using a message passing scheme to send beliefs updates throughout the network.

This message scheme does not work with graphs containing loops. If we look at figure 2.9a and consider messages received by clique SLB: DOB, TLO and LOB are all pairwise connected. Consequently, LOB required messages from XO, TLO and DOB before sending its message to SLB. Likewise, DOB required messages from LOB and TLO and TLO required messages from VT, XO, LOB and DOB. Thus, by rule 2.1, clique SLB, DOB, TLO and LOB cannot send any message.

2.5 Other approaches and conclusion

Probabilistic inference algorithms are not limited to the two approaches presented in this chapter. There is a considerable amount of different approaches, all exploiting distinct features of BNs or MNs. We will also discuss the reasons why we chose to focus on VE and SS.

Variable Elimination variants There exist several extensions and variations of VE, one of the most notable is the Bucket elimination framework that extends VE to other inference task (MPE and MAP) and to other algorithmic problems (constraint satisfaction among others) (Dechter, 1996, 1999; Darwiche, 2010). A generalization of VE to junction tree has been proposed in Cozman (2000a), where VE is changed to save intermediate computations resulting in a structure similar to junction trees. Finally, an optimization of VE has been proposed that focuses on eliminating values instead of variables, i.e., individuals factor's entries are eliminated at each iteration instead of an entire variable (Bacchus et al., 2003).

Other junction tree based algorithms Historically, junction tree algorithms where among the first inference schemes for BNs. Among them, we found different axiomatizations and message passing schemes often similar to SS (Lauritzen and Spiegelhalter, 1988; Shenoy and Shafer, 1990; Lepar and Shenoy, 1999). Currently, the most performing junction tree inference algorithm is Lazy Propagation (Madsen and Jensen, 1999). Lazy propagation exploits d-separation to prevent unnecessary computations. There has been an effort to derive junction tree algorithms without resort to graphical concepts (Draper, 1995; Darwiche, 1998), but these efforts have not produced a variable-elimination-like scheme for inference.

Belief propagation and cutset conditioning. Belief Propagation, also called the polytree algorithm, is a polynomial inference algorithm for polytree BNs (Pearl, 1982; Kim and Pearl, 1983; Pearl, 1988). It is a message passing algorithm which reasons on the BN's graph. This algorithm cannot be applied to non polytree BNs[2]. The reason is that cycles induce a repetition of messages sent by nodes in the same cycle. A solution, called Global Conditioning or loop-cutset conditioning, cuts the cycles by conditioning over the DAG's cutset (Pearl, 1986, 1988; Suermondt and Cooper, 1990). Finding the minimal cutset of a graph is NP-Hard (Garey and Johnson, 1979a; Hao and Orlin, 1994; Becker et al., 2000). However, it is possible to not globally condition the network over every cutset variables, but to limit their conditioning to smaller and sometimes distinct portions of the DAG. Such technique is called Local Conditioning and is by order of magnitude faster than Global Conditioning (Díez, 1996; Fay and Jaffray, 2000). Recursive Conditioning is another inference algorithm more inspired by the idea of conditioning than by the polytree algorithm message passing scheme. It relies on recursively splitting the graph to compute the value of a given query (Darwiche, 2001a; Allen and Darwiche, 2003; Grant, 2010). In the current state-of-the-art, global and local conditioning algorithms are not considered as *efficient* inference algorithms. However Recursive Conditioning offers interesting time-space trade-offs which can be crucial when using inference on limited hardware. This trade-off is emphasized by the fact that conditioning can be applied with a more popular inference algorithm: VE.

[2]At least for exact inference, if applied several times to non polytree BNs it results in an efficient approximate algorithm, called Generalized Belief Propagation (Yedidia et al., 2001).

Darwiche's approach to inference Junction trees algorithms and VE share the same complexity limitation due to MNs tree-width. One approach to overcome the limitation induced by high tree-width for exact inference is to exploit local structures. Local structures are alternate representations of factors, exploiting local symmetries in the probability distribution. However, where junction trees and VE algorithms can exploit specific operators for these local structures, they do not exploit them from a graphical point of view. This shortcoming has led to the development of a new family of inference algorithms which compile MNs into circuits by transforming them into a knowledge base in conjunctive normal form (CNF) (Darwiche, 2003; Chavira and Darwiche, 2005; Chavira et al., 2006; Chavira and Darwiche, 2007). Unfortunately, where this technique seems to have very promising results, it does no have a formal theoretical justification regarding complexity. Thus it is still uncertain if this approach offers better complexity results than the junction trees and VE algorithms (Darwiche, 2010).

Probabilistic inference as optimization A link between probabilistic inference and statistical physics have been recently discovered. This enables to approach probabilistic inference as an optimization problem (Koller et al., 2007; Koller and Friedman, 2009).

Discussion If we look at the recent contributions for probabilistic inference in BNs, VE and SS are old techniques. The current state-of-the-art is more focused on Darwiche's approach and approximate inference with the Generalized Belief Propagation algorithm (Darwiche, 2009). So why did we focused on these two approaches? The main reason comes from the fact that structured inference was developed to be used with VE. When we generalized SVE to obtain the SPI algorithm (see chapter 5.3), we discovered that structured inference was not limited to VE-like algorithms: it could easily be adapted to junction tree based algorithms. Consequently, VE and SS offered the best starting point to adapt structured inference to classic inference algorithms. Possibly, structured inference can be adapted to other inference approaches. However, structured inference was not well defined when this thesis started and we found necessary to first study, formalize and generalize the concepts proposed in Pfeffer (1999). Consequently, adapting structured inference to recent and refined inference algorithms was not thinkable. Hopefully, the work presented in this thesis will provide enough understanding of structured inference to adapt it to other inference schemes.

Conclusion With chapters 1 and 2, we have covered the main notions regarding BNs and probabilistic inference. We have presented MNs and BNs under the scope of conditional independencies and their link with graphoids. We have shown the similarities between MNs and BNs and presented several modeling specificities of BNs. We have then presented two inference algorithms that can be applied both on MNs and BNs: VE and SS. We then concluded by briefly presenting different inference algorithms and discussing the choice of VE and SS as standard inference algorithms. We will now present several BNs extensions that all share the common goal to extend BNs expressiveness to handle large scale complex systems.

Chapter 3

Bayesian Networks Extensions

Since their emergence, PGMs have been at the core of many contributions in the AI community. These contributions either focus on algorithmic aspects (inference, learning) or focus on extending existing models. Sometimes, both topics are intertwined as some extensions have dedicated algorithms. Computer sciences have always been a practical field, with many direct industrial applications. Yet, PGMs are remarkable as they joined many research fields in computer science. Probability elicitation, data mining, reliability, risk management, maintenance, simulation, classification and troubleshooting are the most popular use of PGMs. There are similarities between the recent evolutions of PGMs and programming languages. Indeed, the first programming languages were low level languages, unadapted for complex and big programs. Rapidly, new languages emerged offering modularity, high level syntax and programing paradigms. But in the end, these high level languages are compiled in assembly languages to be used by the computer's CPU. BN extensions emerged to offer better tools for modeling complex systems or specific domains, yet in the end probabilistic inference is almost always done on BNs. In a certain way, BNs (and MNs) are the assembly language of PGMs. Another similarity to programming languages is the recent explosion of new PGMs, all offering specific functionalities or representation paradigm. While we could hope for a consensus over these extensions, there is unfortunately few hopes to find one: as for programming languages the range of applications of PGMs is too big to hope for a universal framework. The purpose of this chapter is to present several BN extensions and focus on object-oriented ones. We cannot cover all existing extensions as there are too many of them and some of them require theoretical background that we will leave undiscussed. Here is a short review of the most remarkable ones: discrete probabilities do not allow to model many domains that require continuous random variables, thus continuous and hybrid (mixing continuous and discrete random variables) frameworks were proposed: Gaussian BNs, Gaussian MNs (Wermuth, 1980; Speed and Kiiveri, 1986; Shachter and Kenley, 1989; Lauritzen, 1996; Malioutov et al., 2006), exponential families (Csiszàr, 1975; Barndorff-Nielsen, 1978; Lauritzen, 1996; Geiger and Meek, 1998) and hybrid BNs (Lerner, 2002); different models have tried to mix directed and undirected graphs, such as factor graphs (Ksichischang et al., 2001; Frey, 2003) and conditional random fields (Lafferty et al., 2001; Sutton and McCallum, 2004, 2007); credal networks use belief functions instead of probabilities (Cozman, 2000b); mixing probabilities and utilities is the core of influence diagrams (Howard and Matheson, 1984; Shachter, 1986; Pearl, 2005); hidden Markov Models are especially known for their application in temporal pattern recognition, for example speech or handwriting (Rabiner and Juang, 1986; Rabiner, 1989); causal networks are another BN extension that set the focus on modeling and finding causal relations between variables (Pearl, 2009).

In this chapter, we will concern ourselves with specific extensions and focus on four distinct categories: early extensions (dynamic BNs (Thomas Dean, 1989), Multiply Sectioned BNs (Xiang et al., 1993b)), object-oriented frameworks (object-oriented BNs (Koller and Pfeffer, 1997; Bangsø and Wuillemin, 2000a)), relational frameworks (Probabilistic Relational Models (Friedman et al., 1999)) and first-order frameworks (Multi Entity BNs (Laskey, 2008), parfactors (Poole, 2003)). To prevent any ambiguity with the verb *model*, we will use the term *framework* to name PGMs, e.g., BNs and MNs are frameworks, the term *system* to designate a model, e.g., the Asia BN used in chapter 2 is a system, and we use the verb *model* to point out the process of creating a system using a framework. We chose these frameworks instead of others because they are the most relevant to the work presented in this thesis. Indeed, they are either strongly inspired by the object-oriented paradigm or are key frameworks in the current state-of-the-art. Still, many other extensions centered over different representation paradigms exist. Here is a short list: stochastic logic programs (Muggleton, 1996), Relational BNs (Jaeger, 1997), Bayesian Logic programs (Kersting and Raedt., 2001), Relational Markov Models (Anderson et al., 2002), Relational Markov Networks (Taskar et al., 2002), Entity-Relationship Probabilistic Models (Heckerman et al., 2007), Markov Logic Networks (Domingos and Richardson, 2007). The reader may also want to refer to other BNs extensions studies (Pfeffer, 1999; Bangsø, 2004; Getoor and Taskar, 2007; de Salvo Braz, 2007; Getoor and Taskar, 2007; Laskey, 2008; Koller and Friedman, 2009).

If we look at BN's history, we see that they appeared in the late 80's and have been continuously extended since. If we consider the integration of the object-oriented paradigm in PGMs, we can see that (Koller and Pfeffer, 1997) is a milestone in BNs history and we will denote by *early extensions* any BNs extension that appeared before (Koller and Pfeffer, 1997). To understand why, we must first explain what a representation paradigm is.

Definition 3.1 (Representation paradigm) *A representation paradigm is a set of desiderata used to structure knowledge bases. Paradigms differ in the concepts and abstractions used to represent the relations among atoms in the knowledge base (such as objects, functions, predicates, random variables, etc.).*

BNs are a propositional framework: as in propositional logic there is no first-order rule among random variables and atoms in the knowledge base must be defined and connected individually. We have mentioned Object-Oriented Bayesian Networks (OOBNs) which are an object-oriented framework, but extensions that are entity-relationship (Probabilistic Relational Models) and first-order (parfactors) frameworks also exist. Each of them will be detailed in this chapter. Before OOBNs, extensions were not explicitly defined by their representation paradigm, which is clearly not the case anymore (most new extensions are qualified as *relational*, *object-oriented* or *first-order* frameworks). However, we will see that in Dynamic Bayesian Networks (DBNs) and Multiply Sectioned Bayesian Networks (MSBNs) we can find buds that led to these new frameworks defined by and for representation paradigms. At first, the object-oriented paradigm was introduced as an engineering solution to model complex systems (Mahoney and Laskey, 1996), this preliminary work led to Multi-Entity Bayesian Networks creation, a framework presented in chapter 3.4. The object-oriented paradigm inspired several frameworks, the most notable being OOBNs (Koller and Pfeffer, 1997). Unfortunately, no existing BN extension using the object-oriented paradigm provides a formal definition of what is an object-oriented PGM. Since the object-oriented paradigm is mostly used as a programming language paradigm, we cannot use any existing definition and apply it to PGMs. The object-oriented paradigm is at the core of the work presented in this thesis, thus to understand the state-of-the-art and our choices, we must first properly define an object-oriented representation paradigm for PGMs. To do so, we propose five desiderata that we think are necessary in any complete object-oriented PGM.

Classes and instances. The first desideratum states that we will distinguish two kinds of objects: classes and instances. Classes are used to define a family of objects sharing common properties, e.g., a car. Instances are specific use of classes, e.g., John Doe's car. Classes definitions change from one framework to another, and in most cases classes and instances are undifferentiated and called *objects*. However, in all frameworks an object is a BN fragment, i.e., a BN with random variables referenced in CPTs that are not represented as nodes in the BN's DAG. We will denote by uppercase letters (C, D, $\dots$) classes and by lowercase letters instances (c, d, $\dots$). If c is an instance of C, we will say that c is an instantiation of C. Random variables associated to a class are called attributes and are denoted by capital letters (X, Y, $\dots$) and we will denote by $C.X$ the attribute X of class C and by $c.X$ the attribute X of instance c.

Class inheritance. The second desideratum is class inheritance. Class inheritance is a partial order among classes: given two classes C and D, we say that C is the super class of D if D is a specialization of C, i.e., if D exhibit all C's properties, specialize some of C's properties and/or have new ones. For example, we could specialize the class car into a specific brand for which we will have more precise failure probabilities. We denote by $C \rhd D$ if C is the direct super class of D, we say that D is a direct subclass of C. We say that C is a super class of D and D a subclass of C, denoted by $C \blacktriangleright D$, if there exists a list $\{C_1, \cdots, C_n\}$ such that $C_i \rhd C_{i+1}$ for $1 \leqslant i \leqslant n-1$ with $C_1 = C$ and $C_n = D$.

Abstraction. The third desideratum is abstraction. Abstraction is the possibility to use objects without having a total knowledge about them, e.g., to consider the failure state of a car it is not necessary to consider the network modeling the car's engine. When confronted to complex systems, abstraction is essential to reduce the complexity explosion due to increasingly large systems.

Polymorphism. The fourth desideratum is subtype polymorphism. Subtype polymorphism is one of the less intuitive feature we can have in an object-oriented PGM, yet we will see that it is a very powerful tool. In programming language theory, subtype polymorphism states that a given super class can be substituted by any of its subclass, i.e., functions that operate on instances of the superclass can also operate on instances of the subclass. In BNs the closest notion to a function is the conditional probability distribution of a node. Indeed, we can see a conditional probability distribution of some node X as a function $\phi(X, Y_1, \cdots ; Y_n)$, where $\{Y_1, \cdots, Y_n) = \pi(X)$. In an object-oriented framework, we can imagine that each node is associated with its class, thus we can rewrite ϕ as follows: $\phi(C.X, D_1.Y_1, \cdots, D_n.Y_n)$. Subtype polymorphism tells us that for any subclass D'_i of D_i we can substitute $D_i.Y_i$ by $D'_i.Y_i$ in ϕ such that $\phi(C.X, D_1.Y_1, \cdots, D_i.Y_i, \cdots, D_n.Y_n) = \phi(C.X, D_1.Y_1, \cdots, D'_i.Y_i, \cdots, D_n.Y_n)$.

Recursive data types. The fifth desideratum is recursive data types. Recursive data types simply state that any class can refer to itself, i.e., that we can define cyclic dependencies among attributes of the same class. This is a must have feature to enable any representation allowed by DBNs, however it becomes possible to create systems with cyclic dependencies. This is of course a major issue as BNs does not cope with cyclic dependencies, thus it is necessary to define rules that prevent from defining such cycles. We will see that each framework offers its solution, spanning from forbidding recursive definition to defining detection algorithms for on-the-fly warnings during modeling.

This chapter is organized as follows: we will first discuss early extensions including DBNs and MSBNs. We will then introduce OOBNs and differentiate OOBNs *à la Pfeffer* from OOBNs *à la Bangsø & Wuillemin*. After, we will present Probabilistic Relational Models and follow with two First-Order Probabilistic Models: Multi-Entity Bayesian Networks and parfactors. We will then conclude with a discussion on the reasons why we chose Probabilistic Relational Models as the base of our own object-oriented framework presented in chapter 4.

3.1 Early extensions

We will now present and discuss two early extensions: Dynamic Bayesian Networks and Multiply Sectioned Bayesian Networks. They are both remarkable because they offer many hidden object-oriented features and they are certainly involved in the first appearance of the object-oriented paradigm in PGMs.

Dynamic Bayesian Networks

In many applications we want to describe a system evolving over time. The most straightforward manner to do so is to describe the system at time t using its state at time $t-1$. From a modeling perspective, this enforces to define repeatedly BN fragments and connect them to span a BN over the desired number of time slices. DBNs purpose is to prevent the definition of every time slice by only defining the prior distribution (the first time slice) and the transitional distribution ($P(X_t | X_{t-1})$) (Thomas Dean, 1989; Smyth et al., 1997). DBNs are the first member of a family of BN extensions called *template based models*. These extensions use templates, i.e., a partial specification of a BN also called a BN fragment, to construct BNs. Templates are similar to classes in the object-oriented paradigm, which make DBNs the first PGMs to use object-oriented features.

Definition 3.2 (2-Time slice BN) *A 2-Time slice Bayesian Network is a Bayesian Network whose nodes are partitioned in two sets $\mathbf{X}_t$ and $\mathbf{X}_{t+1}$ called slices. A 2-TBN is a transitional model such that:*

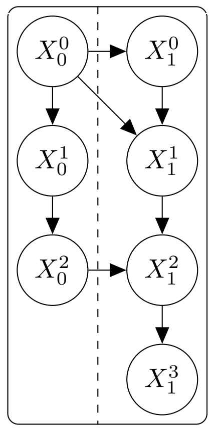
$$P(\mathbf{X}_{t+1}|\mathbf{X}_t) = \prod_{i=0}^{n-1} P(X^i_{t+1}|pa(X^i_{t+1})), \textit{ with } pa(X^i_{t+1}) \subset \mathbf{X}_t \cup \mathbf{X}_{t+1}$$

and the prior distribution

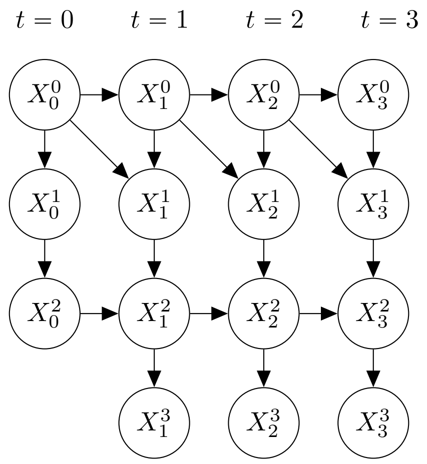
$$P(\mathbf{X}_0) = \prod_{i=0}^{n-1} P(X^i_0|\pi(X^i_0)) \textit{ where } \pi(X^i_0) \subset \mathbf{X}_0.$$

Given an *a priori* distribution $P(\mathbf{X}_0)$ and a transitional distribution $P(\mathbf{X}_t|\mathbf{X}_{t-1})$, it is easy to define a system evolving over time. This leads to the definition of DBNs.

Definition 3.3 (Dynamic Bayesian Networks) *A DBN of length T is a BN resulting from "unrolling" a 2-TBN in T time steps. Each state $\mathbf{X}_t$ of the DBN is called the slice t.*



(a) A 2-TBN represented by the first time slice (the prior distribution) and the second time slice (the transitional distribution) that defines $P(\mathbf{X}_t|\mathbf{X}_{t-1})$.



(b) A DBN can be obtained by unrolling a 2-TBN over the desired time length.

Figure 3.1: 2-TBNs are a simple and compact way to specify DBNs.

Figure 3.1a illustrates a 2-TBN that is unrolled over three time steps in figure 3.1b. Defining DBNs in such manner is space efficient: the modeling is reduced to its strict minimum making such models human readable and easily created by experts. There are however several shortcomings. For example, there is no possibility to modulate the probability distribution over time slices. Indeed, we could imagine that some component's probability failure varies drastically after some time period, i.e., we cannot model systems that do not strictly follow the first-order Markov assumption. Unrolling a 2-TBN over several time steps is nothing but a copy and paste of an I-map, thus it offers very few modeling options. Let us suppose we had an object-oriented framework, by using class inheritance we could easily construct DBNs with more subtle probability distributions. Nevertheless, these shortcomings do not prevent from using DBNs for many industrial problems, but most importantly, they do not prevent inference on large DBNs.

Inference in DBN can be a challenging problem. If we chose to span the network over hundreds or thousands of time steps, we can suppose that inference would be hard. However, defining DBNs using 2-TBNs provides structural information that can be exploited for inference by using constrained elimination orders. Simple constrained elimination orders include forward or backward elimination orders: variables are eliminated from slice 0 to T or from T to 0. Such elimination orders give a theoretical upper bound on the maximum clique size that is independent of the DBN's length. They also offer good experimental results and prevent from triangulating a DBN after its length changed (Kjærulff, 1994; Murphy, 2002; Darwiche, 2001b; Bilmes and Bartels, 2003). The key concept to exploit such elimination is the notion of interface.

Definition 3.4 (Interface) *Let $G_T = (\mathbf{X}, \mathbf{E})$ be the moralized graph of a DBN $\mathcal{B}_T = (P, \vec{G})$ of length T, i.e., there are T times slices $\{\mathbf{X}_0, \cdots, \mathbf{X}_{T-1}\}$. A subset $\mathbf{I} \subset \mathbf{X}$ is called an* interface *if for each path ρ between $X_0 \in \mathbf{X}_0$ and $X_T \in \mathbf{X}_T$ of G_T, $\mathbf{I}$ contain at least one node of ρ. A minimal interface $\mathbf{I}$ is such that for all $\mathbf{I}' \subset \mathbf{I}$, $\mathbf{I}'$ is not an interface.*

Interfaces cut a DBN in two, i.e., an interface is a subset that d-separate the past from the future. At first interfaces were limited to a single time-slice (Darwiche, 2001b).

Definition 3.5 (Forward and backward interfaces) *Let $\vec{G}_T = (\mathbf{X}, \mathbf{E})$ the DAG of a DBN of length T, the forward interface $\mathbf{I}_t^{\rightarrow}$ is the set of all the nodes in slice $t < T$ that have at least one child in slice $t + 1$. The backward interface $\mathbf{I}_t^{\leftarrow}$ is the set of all nodes X in slice $t > 0$ such that X, or one of its children, has a parent in slice $t - 1$.*

For example, in figure 3.1b the backward interface of slice 1 is the set $\mathbf{I}_1^{\leftarrow} = \{X_1^0, X_1^1, X_1^2\}$ and its forward interface is $\mathbf{I}_1^{\rightarrow} = \{X_1^0, X_1^2\}$. These specific interfaces are defined by nodes from the same time slice, yet we can define interfaces overlapping several time slices. For example, $\mathbf{I} = \{X_0^0, X_1^1, X_1^2\}$ is an interface for the DBN of figure 3.1b. Such interfaces can be exploited for inference by deducing a constrained elimination order from them. We can indeed eliminate nodes with respect to an interface instead of their time slice. Let us consider interface $\mathbf{I} = \{X_0^0, X_1^1, X_1^2\}$, that can be generalized to

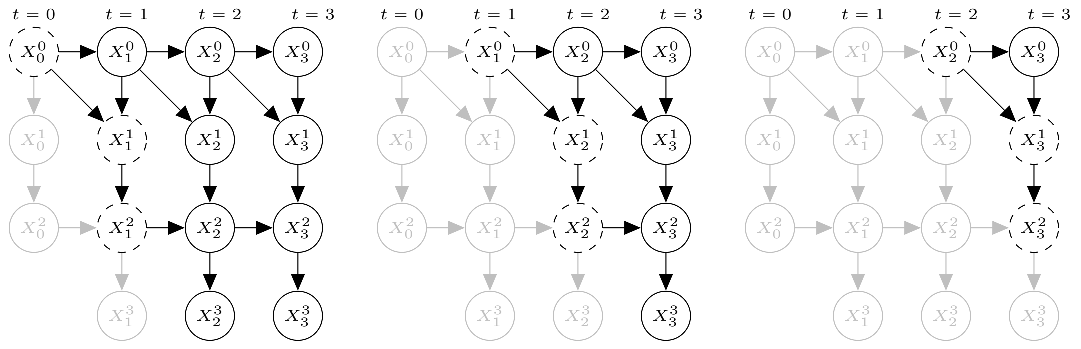


Figure 3.2: Illustrating the elimination ordering obtained by applying a forward elimination with interface $\mathbf{I}_t$ on the DBN of figure 3.1b.

$\mathbf{I}_t = \{X_{t-1}^0, X_t^1, X_t^2\}$ for $t > 0$. Figure 3.2 illustrates a forward elimination order obtained using $\mathbf{I}_t$. However, these interfaces are not necessarily optimal and a recent result proposes an algorithm for finding optimal interfaces in DBNs (Chopin and Wuillemin, 2010).

DBNs introduce several fundamental notions in PGMs. The first is the notion of BN fragments that are repeated as many times as required. Fragmenting a BN will be and is still one of the most used feature in BN extensions. Surprisingly, almost no extensions define fragments using object-oriented notions, i.e., as classes and instances. Another fundamental notion is used by DBNs dedicated inference algorithms: interfaces. Interfaces and their use in DBNs inference is the essence of structured inference, whose principle is to exploit structural repetition to speed-up inference. Furthermore, interfaces define a very simple relational scheme among BN fragments: time slices are connected by their interfaces. This implicitly defines two kinds of nodes: those that belong to the fragment's interface and those that are internal to the fragment. We will see that these notions are primordial for a strong object-oriented framework.

Multiply Sectioned Bayesian Networks

When we are confronted with complex domains, we naturally decompose a domain into subdomains. In computer programming, this concept is called modularity and softwares are decomposed into several parts called modules each with specific functionalities. MSBNs follow this thread of thought: instead of considering the domain with a single *flat* BN, MSBNs use a set of BN fragments to consider each subdomain individually. This offers several advantages, such as helping in modeling the network, i.e., the experts can focus on each specific subdomain and then consider how fragments interact. Splitting apart a BN also reduces inference complexity as each fragment can be processed individually. MSBNs have been first used for medical expert systems (Xiang et al., 1993b,a) and such domains are particularly well suited to a modular approach. Indeed, a disease can be diagnosed by tests that are either disease specific (a blood test) or shared among other diseases (fever). When experts consider a single disease they only need to consider relevant diagnoses, thus they only need to focus on a specific fragment. Furthermore all tests they may undergo while considering a specific disease translate into evidence in the associated fragment. Consequently, belief updating can be limited to the considered fragment, preventing any unnecessary computations. Both the usefulness of fragmenting knowledge in subdomains to model complex systems and the direct link among fragments and evidence updating justified the emergence of MSBNs. Before we start defining MSBNs, it is important to remark that they have considerably evolved since their first appearance in (Xiang et al., 1993b). Our point here is not to give an up-to-date presentation of MSBNs but rather the reasons why we think that they played a crucial role in object-oriented PGMs emergence.

Before we detail MSBNs, it is important to remark that their formalization does not reflect their modeling process. Indeed, we will present a MSBN as being the result of sectioning a BN into several fragments. Whereas, in practice we would model each fragment separately and then connect them, using random variables that would be shared among fragments. This difference in MSBN formalization and modeling process can be puzzling and has some undesired side effects that we will discuss shortly. We will first present the framework, starting by hypertrees definition.

Definition 3.6 (Hypertree, hypernode and hyperlink) *Let $\vec{G} = (V, E)$ be a connected graph sectioned into subgraphs $\{\vec{G}_i = (V_i, E_i)\}$. Let the $\vec{G}_i$s be organized as a connected tree $\mathbf{\Psi}$, where each node is labeled by a $\vec{G}_i$ and each link between $\vec{G}_k$ and $\vec{G}_m$ is labeled by the interface $V_k \cap V_m$ such that for each i and j, $V_i \cap V_j$ is contained in each subgraph on the path between $\vec{G}_i$ and $\vec{G}_j$ in $\mathbf{\Psi}$. Then $\mathbf{\Psi}$ is a hypertree over $\vec{G}$. Each $\vec{G}_i$ is a hypernode, and each interface is a hyperlink.*

Hypertrees generalize junction trees: its nodes are BN fragments and its edges are sets of shared nodes between two fragments and are not necessarily cliques. This structure is exploited to define a distributive inference algorithm in MSBNs (Yang, 2002). The nodes shared among the different sections of a MSBN play an important in the framework's formalization. Fragmenting a graph such that we can construct a hypertree is the inverse operation of creating each fragment and then connect them into a single graph. When modeling complex systems, such modular approach is necessary as considering the network in its entirety is not possible. However, it is necessary to define how fragments interact and to do so we must define the set of nodes that will be the interfaces among those fragments. Such nodes are called d-sepnodes in this framework.

Definition 3.7 (D-sepnodes and d-sepsets) *Let $\vec{G}$ be a directed graph such that a hypertree $\mathbf{\Psi}$ over $\vec{G}$ exists. A node X contained in more than one subgraph of $\mathbf{\Psi}$ is a d-sepnode if there exists one subgraph $\vec{G}_i$ in $\mathbf{\Psi}$ such that $\pi(X) \in \vec{G}_i$. A set $\mathbf{I}$ of nodes is called a d-sepset if every $X \in \mathbf{I}$ is a d-sepnode.*

D-sepnodes and d-sepsets clearly play the roles of junction trees separators. However, there is a condition on how the graph can be sectioned. Indeed, for each node present in several subgraphs, there must be at least one subgraph that contains all its parents. Fortunately, only a weak condition is required for that property to hold in a sectioned DAG.

Condition 3.1 *Let $\vec{G}$ be a directed graph such that a hypertree Ψ over $\vec{G}$ exists. A node X contained in more than one subgraph in $\vec{G}$ must be such that for every pair of subgraphs $\vec{G}_i$ and $\vec{G}_j$ that contain X with the parents $\pi_{ij}(X)$ in $\vec{G}_i \cup \vec{G}_j$, either $\pi_{ij}(X) \subseteq V_i$ or $\pi_{ij}(X) \subseteq V_j$.*

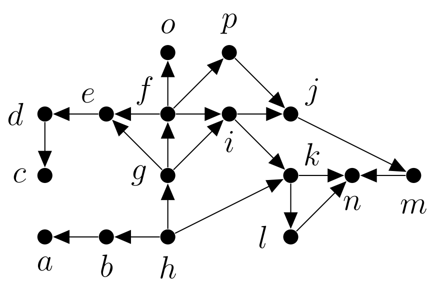
This condition is equivalent to the running intersection property. It ensures that moralization will be properly propagated in the hypertree and consequently, that inference using local message passing is possible in the hypertree structure defined by definition 3.6. When modeling a MSBN, d-sepnodes play an important role because they are shared among the different BN fragments, consequently they must be fully specified. This forces several constraints on the modeling process: before modeling each fragment we must fix each d-sepnode to prevent different names or domains, we must also ensure that for each d-sepnode there will be at least one fragment containing all the node's parents. Defining d-sepsets is nothing more than fixing the interfaces of each fragment, i.e., defining the set of dependencies a fragment relies on. The fragment containing all the parents of a d-sepnode is the resident fragment of that node, i.e., where the node is semantically defined. All these notions need to be clearly defined and properly used to offer an efficient framework for modeling large scale systems. Such notions are present in the MSBN framework but are not defined as modeling features but as topological constraints for the MSBNs inference scheme.

Theorem 3.1 *Let Ψ be a hypertree over a directed graph $\vec{G} = (V, E)$. For each hyperlink I that splits Ψ into two subtrees over $U \subset V$ and $W \subset V$, respectively $U \backslash I$ and $W \backslash I$ are d-separated by I if and only if each hyperlink in Ψ is a d-sepset.*

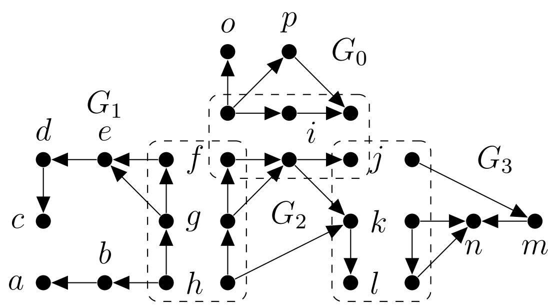
Proof omitted (see Yang (2002)). Theorem 3.1 shows the equivalence between d-sepset and separators: as a separator in a junction tree, a d-sepset d-separates one side of the hypertree from another. This leads us to the definition of a hypertree Multiply Sectioned DAG, the last notion required to define MSBNs.

Definition 3.8 (Hypertree Multiply Sectioned DAG (MSDAG)) *a MSDAG $\vec{G} = \bigcup_i \vec{G}_i$, where each $\vec{G}_i = (V_i, E_i)$ is a DAG, is a connected DAG such that there exists a hypertree Ψ over $\vec{G}$, and each hyperlink in Ψ is a d-sepset.*

Figure 3.3 illustrates the different graphical steps leading from a DAG to hypertree multiply sectioned DAGs. Figure 3.3a is a DAG that is decomposed into four different subgraphs in figure 3.3b: $G_0 = \{f, i, j, p, o\}$, $G_1 = \{a, b, c, d, e, f, g, h\}$, $G_2 = \{f, g, h, i, j, k, l\}$ and $G_3 = \{j, k, l, m, n\}$. Note that a MSDAG is not simply a fragmented DAG: d-sepnodes are repeated between adjacent subgraphs. For example, node f is in G_0, G_1 and G_2 and it is repeated in each subgraphs. The multiply sectioned DAG defines the hypertree of figure 3.4.



(a) A DAG with sixteen nodes.



(b) Decomposition of figure 3.3a in four subnetworks G_0, G_1, G_2 and G_3.

Figure 3.3: A DAG can be sectioned into subgraphs and then organized in a hypertree. By ensuring that the sectioning respects condition 3.1, we have the guarantee that each hyperlink is a d-sepset in the Hypertree Multiply Sectioned DAG (Yang, 2002).

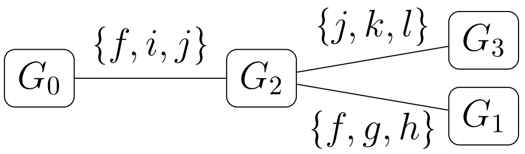


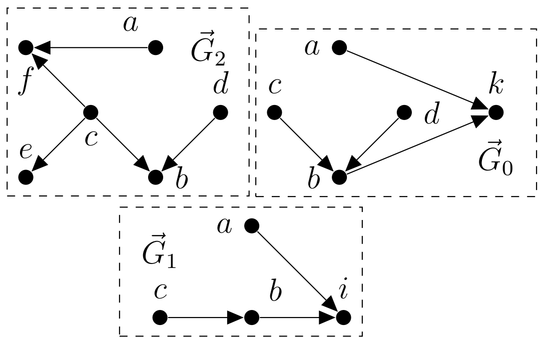
Figure 3.4: The subgraphs of figure 3.3b can be organized in a hypertree. Note that each hyperlink is a d-sepset in figure 3.3b (Yang, 2002).

Definition 3.9 (Multiply Sectioned Bayesian Network) *A MSBN M is a triplet $M = (V, \vec{G}, P)$: $V = \bigcup_{1 \leqslant i \leqslant n} V_i$ is the total universe where each V_i is a set of variables called a subdomain. $\vec{G} = \bigcup_i \vec{G}_i$ (a hypertree MSDAG) is the structure where nodes of each subgraph G_i are labeled by elements of V_i. Let X be a variable and $\pi(X)$ be all the parents of X in $\vec{G}$. For each X, exactly one of its occurrences (in a $\vec{G}_i$) containing $\{X\} \cup \pi(X)$) is assigned $P(X|\pi(X))$, and each occurrence in other subgraphs is assigned a uniform potential. $P = \prod_i P_i$ is the joint probability distribution, where each P_i is the product of the potentials associated with nodes in $\vec{G}_i$. Each triple $S_i = (V_i, \vec{G}_i, P_i)$ is called a subnet of M. Two subnets S_i and S_j are said to be adjacent if $\vec{G}_i$ and $\vec{G}_j$ are adjacent in the hypertree.*

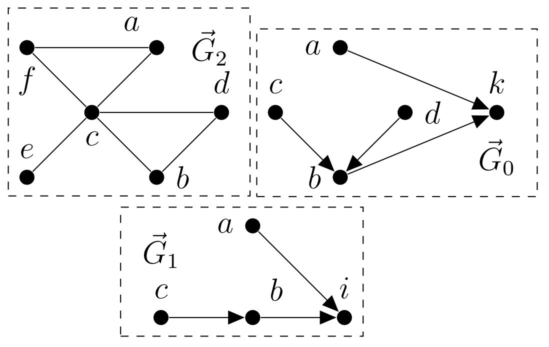
A MSBN is a structure composed of BN fragments (the subnets) connected by d-sepnodes. Each set of d-sepnodes shared between two fragments form a d-sepset that d-separates the hypertree in two. From an object-oriented perspective, each fragment can be seen as a class instantiation and the d-sepsets among each fragment can be seen as an interface[1] on which each class relies to define its probabilistic dependencies. This last point is surely the major contribution of MSBNs from a modeling perspective: d-

[1]Here, the term interface is used as in object-oriented programming languages or in UML. A formal definition of a PGM interface is given in chapter 4. Until then the reader should rely on his (object-oriented) intuition regarding the notion of interfaces in PGMs.

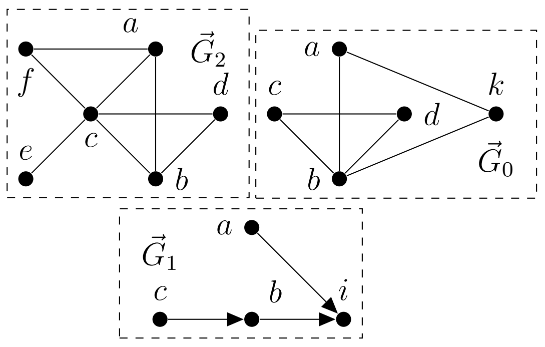
sepsets are the only relevant information that must be shared among subnets. Moreover, the conditional probability distribution associated to each d-sepnode is not relevant to ensure inference correctness and only a unique copy of the d-sepnode receives the *true* conditional probability distribution, where all others receive placebo conditional probability distributions. Such notions have also been developed in DBNs but MSBNs show us that the structural repetition characterizing DBNs and the fragmenting of a BN are not correlated: they can both be exploited separately.



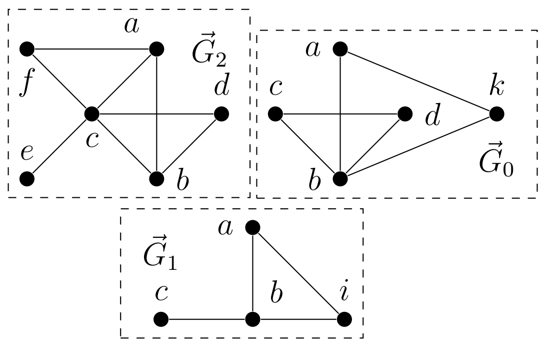
(a) A MSDAG composed of subgraphs $\vec{G}_0$, $\vec{G}_1$ and $\vec{G}_2$.



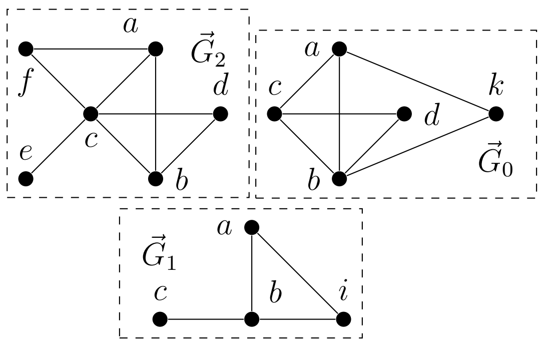
(b) $\vec{G}_2$ is chosen as the root node, it first moralizes itself and asks to its neighbors to send their fill-ins.



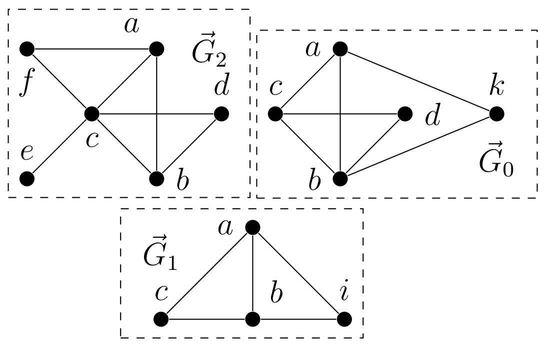
(c) $\vec{G}_0$ sends edges (a, b) and (c, d) to $\vec{G}_2$ and (a, b) is added to $\vec{G}_2$.



(d) $\vec{G}_1$ sends edge (a, b) to $\vec{G}_2$ and no edge is added to $\vec{G}_2$.



(e) $\vec{G}_0$ receives a diffuse message from $\vec{G}_2$ containing edges (a, b), (a, c) and (c, d).



(f) $\vec{G}_1$ receives a diffuse message from $\vec{G}_2$ and adds edge (a, c).

Figure 3.5: The collect phase of a distributed moralization in MSDAGs (Yang, 2002).

Inference in MSBNs requires dedicated algorithms for the moralization and triangulation of MSDAGs. Since d-sepnodes are present in different subgraphs, it is necessary to propagate fill-ins added during moralization and triangulation. To do so, a similar approach to inference in junction tree is proposed: a root node is chosen in the hypertree MSDAG, then a collect phase moralizes each subgraph from leaves to root. Then, a diffuse phase from the root to the leaves ensures that all subgraphs are correctly moralized. Figure 3.5 illustrates the collect and diffuse phase of the distributed moralization of MSDAGs. The first step illustrated by figure 3.5b consists in the root's subgraph moralization, here $\vec{G}_2$. Then $\vec{G}_2$ asks for collect messages from $\vec{G}_0$ and $\vec{G}_1$. In figure 3.5c, $\vec{G}_0$ is moralized and sends a collect message containing edges (a, b) and (c, d). Since $\vec{G}_2$ does not contain (a, b), it is added to $\vec{G}_2$. In figure 3.5d, $\vec{G}_1$ is moralized and the collect message with edge (a, b) is sent to $\vec{G}_2$. Since that edge is already in $\vec{G}_2$ no edge is added. Finally, $\vec{G}_2$ sends diffuse messages to $\vec{G}_0$ and $\vec{G}_1$ resulting in the addition of edge (a, c) to $\vec{G}_0$ and $\vec{G}_1$. Triangulating the moralized MSDAG is done similarly (Yang, 2002). The final step consists in transforming the moralized and chordal MSDAG into a linkage tree.

Definition 3.10 (Linkage tree) *Let $\vec{G}$ a subgraph in a hypertree MSDAG, $\mathbf{I}$ the d-sepset between $\vec{G}$ and an adjacent subgraph, and T a junction tree converted from $\vec{G}$. Repeat the following procedure in T until no node removal is possible:*

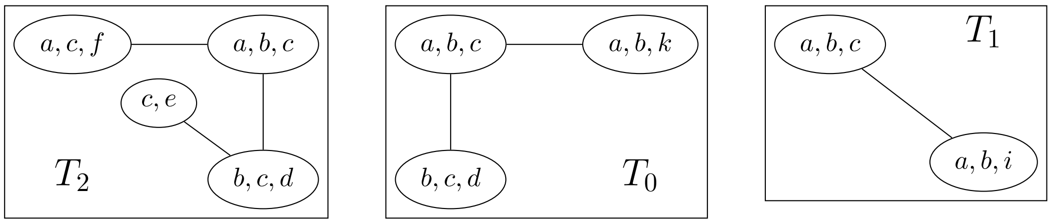
1. *Remove $X \notin \mathbf{I}$ if X is contained in a unique cluster $\mathbf{C}$.*
2. *After removal, if $\mathbf{C}$ becomes a subset of an adjacent cluster $\mathbf{D}$, merge $\mathbf{C}$ into $\mathbf{D}$.*

Let L be the resultant cluster graph. Then L is a linkage tree of T with respect to $\mathbf{I}$ if $\bigcup_{\mathbf{Q} \in L} Q = \mathbf{I}$, where each cluster $\mathbf{Q}$ in L is called a linkage. A cluster in T that contains $\mathbf{Q}$ (breaking ties arbitrarily) is called the linkage host of $\mathbf{Q}$.

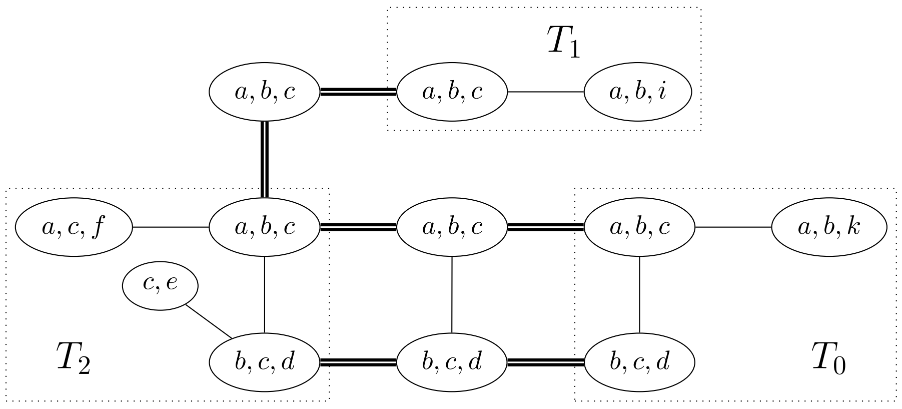
Figure 3.6 illustrates the construction of a linkage tree from a junction tree forest. The inference algorithm exploiting linkage trees is similar to classic junction trees inference algorithms. However, given the nature of linkage trees the algorithm is more complex. For instance, cliques initialization and message passing are more complex due to the repetition of linkage throughout the linkage tree. We will not detail the algorithm and the reader should refer to Yang (2002) for a complete presentation.

Early extensions shortcomings

The major flaw of these two extensions is the absence of any representation paradigm. In the case of DBNs, templates are only used to enable massive copy/pastes and where some inference algorithms exploit the structure, it is never used from a modeling perspective. However, it seems very intuitive how DBNs could benefit from object-oriented notions: class inheritance and abstraction can model systems in which the probability $P(X_t|X_{t-1})$ is not always identical. MSBNs at first lacked any representation paradigm, however this extension have seen an evolution in the early 2000's and have been adapted



(a) Local junction trees compiled from figure 3.5f.



(b) The linkage tree obtained from figure 3.6a.

Figure 3.6: A linkage tree obtained from a junction tree forest (Yang, 2002).

to the multi-agent representation paradigm. Indeed, sectioning BNs into fragments combined with the distributed inference algorithms defined for MSBNs result in a sound formalism to represent multi-agent systems. Yet, the absence of any object-oriented aspect is not justifiable, as we can easily imagine multi-agent systems in which we will want to model family of agents using class inheritance. Furthermore, the absence of abstraction does not enable to plug different fragments sharing common properties. However, if there is no clear representation paradigm in these models, we can still see the emergence of some object-oriented notions. Interfaces in DBNs are similar to the notion of d-sepset in MSBNs, and we will see that these two notions can be generalized to the notion of interfaces in the framework presented in chapter 4.1. Using templates to represent different time slices is also a basic use of classes, and so is the case of subgraphs in MSBNs. The fact that these notions emerge naturally in these early extensions is a good indicator of the need of a sound object-oriented framework in PGMs. In the next section we will discuss a first attempt to formalize PGMs using the object-oriented paradigm.

3.2 Object-Oriented Models

As said in the introduction of this chapter, object-oriented extensions of BNs were the first to offer a paradigm shift of BNs representation. As we will see, state-of-the-art object-oriented extensions are not satisfying: many of our object-oriented desiderata are either missing or poorly thought. This does not prevent these frameworks from being remarkable and the flaws we will point out are the sole consequence of these frameworks innovativeness. We will present two BN object-oriented extensions that, unfortunately, share a common name: Object-Oriented Bayesian Networks (OOBNs). To prevent any ambiguity we will differentiate them by their respective authors, thus calling them OOBNs *à la Pfeffer* for the framework from Koller and Pfeffer (1997); Pfeffer (1999) and OOBNs *à la Bangsø & Wuillemin* for the framework from Bangsø and Wuillemin (2000a,b).

Object-Oriented Bayesian Networks *à la Pfeffer*

The framework

OOBNs *à la Pfeffer* are a complex framework, thus we will first give an overview of them before we detail each of their features. OOBNs *à la Pfeffer* revolve around objects that are defined by a set of typed attributes. Types can either be simple or structured: simple ones refer to random variables; structured ones to collections of simple and structured types. Object's attributes are partitioned in three sets: inner, input and output attributes; and objects are either simple or complex. Simple objects contain only one output attribute and zero to n input attributes. They represent classic random variables in a BN. Complex objects are any other type of objects, i.e., a BN fragment without probabilistic semantics. Object's types are inferred by their attributes: for simple objects types are necessarily simple since simple objects contain only attributes; for complex objects, types are defined by a structured type obtained by the union of the object's attributes and encapsulated objects (recursive definitions are forbidden). Probabilities are introduced using *Object Oriented Network Fragments* (OONF). An OONF can either be simple or complex. A simple OONF associates a simple object with a stochastic function (Koller et al., 1997a), i.e., a more elaborate definition of a CPT. A complex OONF combines a DAG with a complex object, where nodes are the object's attributes and arcs connections among outputs attributes and inputs attributes. Finally, if required an Object-Oriented BN can be a class and reused each time it is needed.

OOBNs *à la Pfeffer* separate the knowledge base definition from its probabilistic semantics. The atomic unit in this framework are types that can either be basic or structured.

Definition 3.11 (Basic type) *A basic type is a set of values, which is either one of the predefined basic types (Booleans, Integers or Reals) or some user-defined enumerated sets (e.g., WEATHER* $= \{raining, cloudy, sunny\}$*). A basic variable is a variable that takes values in some basic type.*

In this framework, values refer to random variables domains values, e.g., each element of the set $\{raining, cloudy, sunny\}$ is a value and domains are called types. Typing is useful when we want to model several random variables with identical domains. Not only this avoids having to redefine each time the variable's domain but it also gives a simple mechanism to identify equivalent variables. Note that in this framework booleans, integers and reals are considered built-in types, i.e., that there is no need to define them in a system to use them. Where booleans and integers naturally define discrete random variables, real types do not. This is an issue since mixing continuous and discrete random variables is a non trivial matter and that there is no indication on how to proceed with both kinds of random variables.

Definition 3.12 (Structured type) *A structured type is a set of values defined by a type $\langle A_1 : t_1, \cdots, A_n : t_n \rangle$, where $A_1, \cdots, A_n$ are attribute labels and $t_1, \cdots, t_n$ are their corresponding (basic or structured) types. The set of values of this type are all those of the form $\langle A_1 : v_1, \cdots, V_n : v_n \rangle$ where each v_i is of type t_i. A structured variable is a variable which takes values in some structured type.*

A structured type enumerates the set of random variables and encapsulated structured types it contains. The fact that no probabilistic dependency is associated with a structured type is both interesting and puzzling. It is interesting because structured types gather almost all the information required to define probabilistic dependencies: variables are named and their domain is fixed. However, why not call structured types classes or class types? We will see that structured types purpose are to guarantee the coherence of probabilistic dependencies, however objects are defined by their types but they also define a type. We will see in chapter 4 that structured types are in fact interfaces.

Definition 3.13 (Simple object) *A simple object X is composed of a set of labeled input attributes $I_1, \cdots, I_k$ and a single output attribute labeled Output. All of X's attributes are basic variables.*

Simple objects define the probabilistic dependencies of attributes: input attributes are the output attribute's parents. However, there is still no conditional probability distribution associated with the output attribute. That the output attribute is necessarily labeled *Output* can be confusing and seems superfluous.

Definition 3.14 (Complex Object) *A complex object X is composed of a set of labeled attributes. The attributes are partitioned into three sets: the input attributes $\mathcal{I}(X)$, the output attributes $\mathcal{O}(X)$, and the encapsulated attributes $\mathcal{E}(X)$. The output attributes and encapsulated attributes are called value attributes, and denoted $\mathcal{A}(X)$. The input attributes are (basic or structured) variables. The value attributes are themselves objects.*

Both simple and complex objects definitions are confusing. The difference between structured types and objects is unclear. What we can understand is that complex objects are defined by a set of attributes separated in three categories: input, encapsulated and output attributes. Output attributes must be attributes allowed to be parents of attributes in other objects and input attributes must serve the purpose of parents in the definition of the object's attributes. But there is no place left for defining dependencies among the object's attributes. Another confusion comes from the fact that objects attributes are *labeled*, thus associated with a type in some structured or basic type definition. Why they are not directly typed or why objects are not simply associated with a type is puzzling. Furthermore, input attributes are variables, does that mean that input and output attributes are not typed? Why are they not variables?

Definition 3.15 (Simple object's full variable) *For a simple object X, its full variable $\mathcal{V}^+(X)$ and its output variable $\mathcal{V}(X)$ are defined to be X's output variable X.Output.*

Definition 3.16 (Complex object's full variable) *For a complex object X, its full variable $\mathcal{V}^+(X)$ is a structured variable whose value is $\langle A_1 : V_1, \cdots, A_n : V_n\rangle$, where $A_1, \cdots, A_n$ are X's value attributes, and V_i is $\mathcal{V}^+(X.A_i)$. Analogously, the output variable $\mathcal{V}(X)$ is a structured variable whose value is $\langle O_1 : U_1, \cdots, O_k : U_k\rangle$, where $O_1, \cdots, O_k$ are X's output attributes and U_i is $\mathcal{V}(X.O_i)$.*

Full variables are the underlying random variables of simple objects or the underlying set of random variables for complex objects. Both definitions clarify the previous ones: complex and simple objects are associated with types variables. These types are either basic or structured depending on the object's being simple or complex. The difference between objects and variables is that objects are organizational units, describing entities in the knowledge base and variables associate objects with actual random variables.

Definition 3.17 (Simple object-oriented network fragment) *A simple object-oriented network fragment F has a set of input attributes $\mathcal{I}(F)$ and a single value attribute Output, all of which are basic variables. The network consists of a conditional probability function defining a distribution over $Val($Ouput$)$ for each assignment of values in $Val(\mathcal{I})$.*

As for simple objects, simple object-oriented network fragments model a single random variable. The only difference with simple objects reside in the definition of conditional probability distributions: output attributes can be seen as nodes in a BN and the input attributes as the node's parents.

Definition 3.18 (Object-Oriented Network Fragment) *An object-oriented network fragment (OONF) F over the input attributes $\mathcal{I}(F)$ and the value attributes $\mathcal{A}(F)$ consists of a DAG whose nodes are the attributes of F, and, for each value attribute $A \in \mathcal{A}(F)$:*

- *For each input I, an annotation $B.\rho$, where B is a parent of A and ρ is an attribute of $\mathcal{V}(B)$. We require that the attributes $A.I$ and $\mathcal{V}(B).\rho$ have the same type. We also require that every parent of A be used to annotate at least one input of A.*

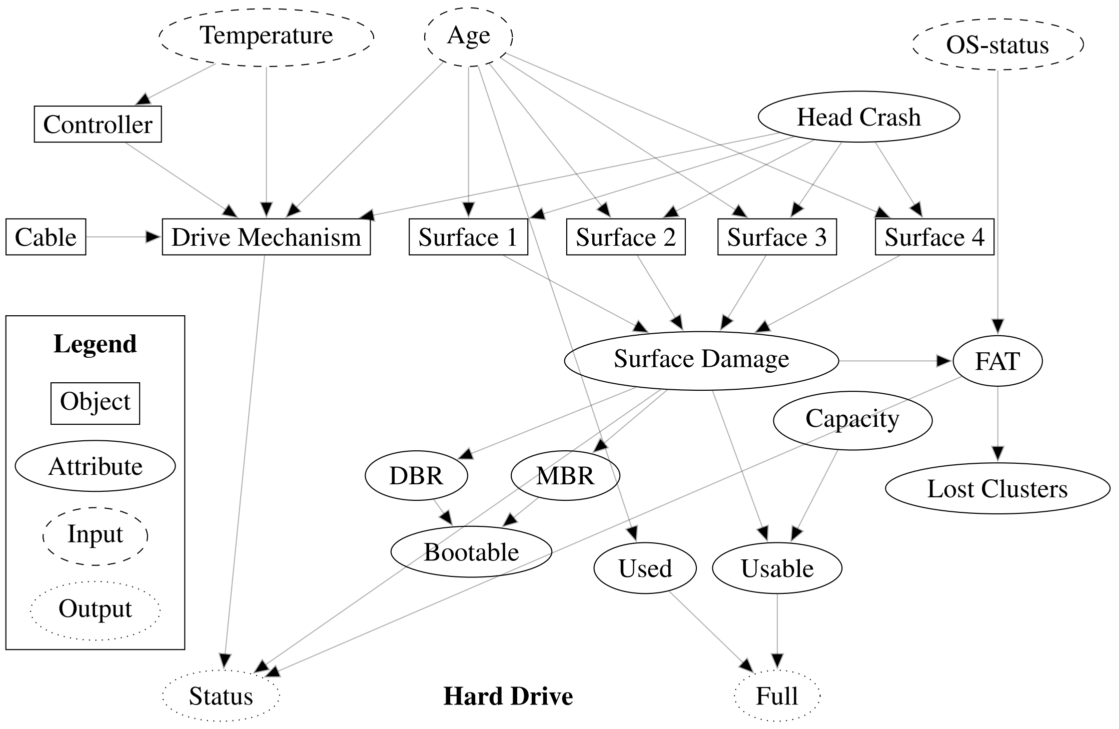


Figure 3.7: The hard drive object-oriented network fragment from the computer diagnosis example (Pfeffer, 1999).

- *An OONF F_A. If A is a simple attribute, then F_A must be a simple $OONF$.*

Object-Oriented network fragments associate complex objects with conditional probability distributions. Note that there is no real object-oriented aspect to an object-oriented network fragment and it should be called a plain network fragment. A remarkable feature of OOBNs *à la Pfeffer* is that object-oriented network fragments are similar to functions, i.e., they have parameters (the input attributes) and an output (the output attributes). If we look at Pfeffer and Koller preliminary work, we can see that they focused on stochastic functions and on a functional approach for BN specification (Koller et al., 1997a,b). Later, Pfeffer developed an implementation of OOBNs *à la Pfeffer* by extending the programming language Ocaml, a functional programming language (Pfeffer, 2001). Whereas OOBNs *à la Pfeffer* is often considered as the first and major object-oriented extension of BN, it is in fact largely influenced by the functional programming languages and is a strange mixture of the object-oriented paradigm and the functional paradigm.

Figure 3.7 is an object-oriented network fragment taken from the computer diagnosis example (Pfeffer, 1999). This object represents a computer's hard drive. We can see that it has the three input attributes *Temperature*, *Age* and *OS-status*. It also has the two output attributes *Status* and *Full*. We can distinguish encapsulated objects from encapsulated attributes by the nodes shapes: rectangle nodes are objects and ellipse nodes are attributes. Consequently, in-going and out-going arcs refer to input or output attributes of the encapsulated object when they pass by a rectangle node. For example, *Drive Mechanism*, which is an object, has three in-going arcs. Two from the input attributes *Temperature* and *Age*, this implies that they are both input attributes of *Drive Mechanism*. The third arc connects *Controller* to *Drive Mechanism*, i.e., some output attribute of *Controller* is an input attribute of *Drive Mechanism*.

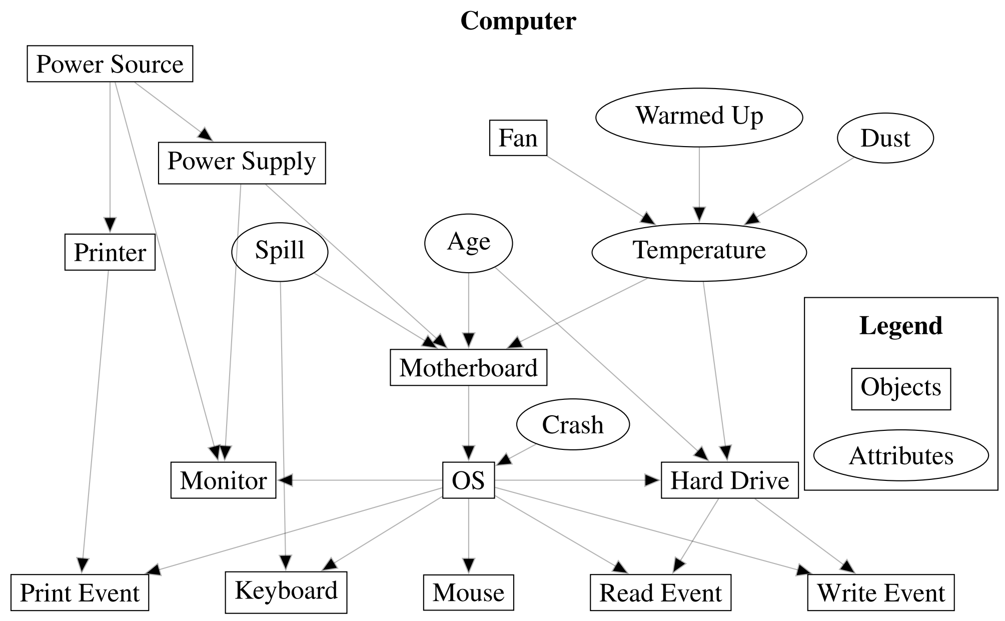


Figure 3.8: The computer diagnosis example (Pfeffer, 1999).

Definition 3.19 (Object-Oriented Bayesian Networks) *An Object Oriented Bayesian network consists of a set of class definitions $C_1, \cdots, C_m$, and a single* Situation *object (that has no input attributes) with an associated OONF. An object X in the model (including within a class definition) can only be associated with one of the defined classes $C_i (i = 1, \cdots, m)$, and C_i must be type-compatible with X.*

Here classes are introduced as *"generic object-oriented network fragments, which can be used multiple times in defining many similar objects"*. The fact that classes are introduced so late in the framework points out its awkwardness. Indeed, if classes are generic object-oriented network fragments, why do not directly call them classes? Supposing that classes are mostly instantiated once also shows the misunderstanding of the correct use of an object-oriented PGM. Mostly, OOBNs *à la Pfeffer* suffer from a poor choice of terms and misconception of what an object-oriented BN extension should be. Fortunately, Koller and Pfeffer were not unaware of these issues and proposed an extension to OOBNs *à la Pfeffer* called Probabilistic Relational Models (Friedman et al., 1999).

Figure 3.8 illustrates an Object-Oriented Bayesian Network *à la Pfeffer* for the computer diagnosis example (Pfeffer, 1999). Figure 3.7 details the *Hard Drive* object encapsulated in the *Computer* object, which is represented in figure 3.8. Note that *Computer* does not have any input or output attribute, this implies that *Computer* is a top-level object and models an entire system. Attributes of such high-level objects are often global variables and shared among many objects. For example, *Age* is both an input attribute of *Motherboard* and *Hard Drive*. If we look to figure 3.7, we can see that *Age* is an input attribute to all *Surface* objects, *Drive Mechanism* and is a parent of attribute *Used*.

In definition 3.19, the notion of type compatibility is mentioned. It guarantees the absence of recursive definitions: *"the value type of an attribute $X.A$ must be strictly simpler than the value type of X"*. In other words, if a class C' is encapsulated in an object-oriented network fragment C then C cannot be used (even indirectly) within C'. This restriction directly prevents the definition of recursive data types which is a necessary feature to model dynamic systems.

Class inheritance

OOBNs *à la Pfeffer* also introduce the notion of subtyping and inheritance, despite being rudimentary we will see that these notions will be used as a base to develop class inheritance. The first necessary notion of interface type.

Definition 3.20 (Interface type) *The interface type of a class C is a tuple $\langle I_1 : t_1, \cdots, I_k : t_{I_k} \rightarrow O_1 : t_{O_1}, \cdots, m : t_{O_m} \rangle$ where $\mathcal{I}(C) = \{I_1, \cdots, I_k\}$, $\mathcal{O}(C) = \{O_1, \cdots, O_m\}$ and t_A is the type of attribute A.*

Note that there is not much difference between interface types and structured types, but we will just ignore here the framework's incoherence. Interface types are what we could call the signatures of an object-oriented network fragments. In programming language a function is defined by its name, its inputs and its output[2]. Those three properties are the function's signature. Here an object-oriented network fragment is its set of input and output attributes, which matches the set of attributes required to connect the object with other ones in the system. Thus type interface is a form of abstraction as it hides encapsulated attributes. Typing classes is necessary to define a coherent class inheritance mechanism and the next step to define such an *is-a hierarchy* in OOBNs *à la Pfeffer* is the notion of interface subtyping.

Definition 3.21 (Interface subtype) *An interface type t' is a subtype of an interface type t if:*

- *If t has an output attribute named A, then t' must have a corresponding output attribute A. The output type of $t'.A$ must be a subtype of the output type of $t.A$.*
- *If t' has an input attribute A, then t must have a corresponding input attribute A. The type of $t.A$ must be a subtype of the type of $t'.A$.*

[2]A function outputs is not considered part of the functions signature in many programming languages, such as C or C++.

Interface subtyping tells us that any subtype is *family preserving*, i.e., that any existing attribute in a given type t will exist in any subtype of t, either with the same type or subtype. This introduces a very useful notion called overloading. In the object-oriented paradigm, overloading is a form of a specialization: it is when we replace an inherited property by a more specialized one. However, overloading guarantees type compatibility, i.e., the specialized property can be used wherever its ancestor was used. To allow such feature in OOBNs *à la Pfeffer* the constraint on input attributes is necessary: an input attribute of a subtype must be present in the super type. This goes clearly against the idea of specialization through subtyping. The reason why such constraint was added is to allow subtype polymorphism: if a subtype has more inputs that its super type, then when we use a subtype in place of a super type some dependencies will not be defined. We will see that OOBNs *à la Bangsø & Wuillemin* answer this issue by introducing default distributions to input attributes. Note that the inverse is not true: a subtype can have fewer input attributes than its super type. This is because in BN adding a child does not change a variables conditional probability distribution definition, thus when using such subtype in place of its super the underlying probability distribution is still coherent.

Definition 3.22 (Class inheritance) *C' is a subclass of C if $\mathcal{O}(C) \subseteq \mathcal{O}(C')$, and the projected interface type of C' onto $\mathcal{O}(C)$ is a subtype of the interface type of C. An is-a hierarchy over classes is a hierarchy $\sqsubset$ over the set of classes such that, if $C' \sqsubset C$, then C' is a subclass of C.*

This definition outlines the general idea of class inheritance in PGMs: it must preserve any declared random variable so when two instances c_1 and c_2 are connected we can replace either one, or both, by an instance of one of their respective subclasses. We will see in chapter 4 that the constraints present in OOBNs *à la Pfeffer* are only due to how the dependencies between classes are defined. This framework is remarkable in several ways. First of all, it is the first BN extension to use the object-oriented paradigm as a representation paradigm. The main purpose of adding object-oriented notions here is to fragment a BN into objects that are connected using strongly typed interface attributes. By setting the focus on how these fragments are connected, i.e., on the relations between network fragments, OOBNs *à la Pfeffer* initiated a large number of contributions that will focus on different sort of paradigm representations. Unfortunately, OOBNs *à la Pfeffer* suffer several shortcomings. The most important ones are the misuse of object-oriented terminology, the weak distinction between classes and instances, the impossibility to define recursive data types and, finally, the separation between the knowledge base (objects) and the probabilistic semantics (network fragments). The last point is paramount, as OOBNs *à la Pfeffer* suffers from many unnecessary definitions making this framework difficult to use. This issue is partially answered in IBAL, where OOBNs *à la Pfeffer* are represented using a simpler formalism (Pfeffer, 2001). Finally, it seems that OOBNs *à la Pfeffer* have been used for military applications. Unfortunately, there is only scarce information about such applications (Pfeffer, 1999; Pfeffer et al., 1999).

Inference in OOBNs *à la Pfeffer*

Several solutions exist to make inference in OOBNs *à la Pfeffer*, among them grounding the system into a *flat* BN is the most trivial solution. Since any OOBN *à la Pfeffer* can be transformed into a BN, we can use any existing inference algorithm for BN over OOBNs *à la Pfeffer* (Koller and Pfeffer, 1997; Pfeffer, 1999). Grounding is a common feature to any BN extension. For many small systems, this is the most convenient solution for inference: grounding a network is fast (polynomial in the size of the network (Pfeffer, 1999)) and there is a considerable amount of existing softwares for probabilistic inference, thus reducing considerably implementation costs. Unfortunately, ground inference is impractical when we deal with large and complex networks. Since the object-oriented paradigm is meant to help modeling such systems, ground inference often leads to large BN that most existing inference algorithms cannot handle. We will now discuss Pfeffer's solutions to this issue.

A first solution is to transform an OOBN *à la Pfeffer* into a MSBN (Koller and Pfeffer, 1997). However, inference over MSBN is not meant to deal with large networks with a strong structure, i.e., with many repeated subgraphs. Consequently, MSBN's inference algorithm does not offer any substantial speed up in inference time. However, merging MSBN and OOBNs *à la Pfeffer* to offer an object-oriented framework for multi-agent systems seems promising.

A second solution is to exploit the hierarchical nature of OOBNs *à la Pfeffer* to define a heuristic for finding good elimination orders (Pfeffer, 1999). Hierarchical inference use encapsulation among objects to infer sets of nodes that d-separate encapsulated objects from their containers. If we look at figure 3.8 before eliminating attributes of the *Computer* object, we would first eliminate each of its encapsulated objects. Suppose we first eliminate the *Hard Drive* object illustrated in figure 3.7, then we would eliminate each of its encapsulated objects and then eliminate its encapsulated attributes. This lead to the first version of Structural Variable Elimination (Pfeffer, 1999). Regrettably, the algorithm proposed in Pfeffer (1999) suffers from an overly complex formalization, making it obscure. Yet, experimental results seem to show good performances. However, there are few informations on the data used for experimentation, thus we cannot be too conclusive about the efficiency of this approach (Pfeffer, 1999).

The third and most promising solution is to exploit the strong structure present in OOBNs *à la Pfeffer*. If a class is reused several times in a system, then its DAG will be repeated as many times in the underlying ground BN. By exploiting an elimination order that removes encapsulated attributes before input and output attributes we can reuse the computation in each of the class' instances. Such elimination order is obtained by using the hierarchical nature of OOBNs *à la Pfeffer*. This approach completes the Structured Variable Elimination proposed in (Pfeffer, 1999). However, as for the second approach, experimental results using this technique are limited to Pfeffer (1999) and do not offer satisfying experimental data (Pfeffer, 1999).

Structured Variable Elimination has also been adapted to inference in Probabilistic Relational Models for open world systems. We will detail this specialization of SVE in chapter 3.3. Structural and hierarchical inferences are at the core of this thesis and it is undeniable that Pfeffer's algorithm is the first to propose such an approach. However, both the frameworks used and the experimental results are unsatisfactory. Indeed, OOBNs *à la Pfeffer* suffer from many flaws and SVE as it is formalized in (Pfeffer, 1999) suffers from them. As a consequence we have found very difficult to exhibit Pfeffer's work and directly extend it. Our version of structured inference is presented in chapter 5.

Object-Oriented Bayesian Networks *à la Bangsø & Wuillemin*

In Bangsø and Wuillemin (2000a,b) an extension of Koller's and Pfeffer's work is proposed. Both frameworks share equivalent expressive power, with the possibility to model dynamic OOBNs using Bangsø's framework. We will see that the main difference between both frameworks is the formalization: OOBNs *à la Bangsø & Wuillemin* is more centered on modeling and offers a simpler framework than OOBNs *à la Pfeffer*. A complete definition of OOBNs *à la Bangsø & Wuillemin* can be found in Bangsø's thesis (Bangsø, 2004). In Bangsø et al. (2003) a link between OOBNs *à la Bangsø & Wuillemin* and inference in MSBNs is provided. A medical decision support systems built using OOBNs *à la Bangsø & Wuillemin* is proposed in Bangsø and Olesen (2003) and in Bangsø et al. (2006) they are applied to virtual agents in role playing games.

The framework

Unlike OOBNs *à la Pfeffer*, OOBNs *à la Bangsø & Wuillemin* use classes instead of objects. As hinted previously, classes are BN fragments and serve as templates to repeat patterns several times in a system. In OOBNs *à la Bangsø & Wuillemin*, the knowledge base and the probabilistic semantics are not separated as in Pfeffer's framework. However, nodes are separated in three distinct subsets, following the same idea proposed in the previous section with subtle changes.

Definition 3.23 (Class) *A class* $\mathbf{C}$ *is a DAG over* $(I \cup P \cup O, E)$*, where* I *is the set of input nodes,* P *is the set of protected nodes,* O *is the set of output nodes, and* E *is the set of directed edges between nodes.*

- I*,* P*,* O *are pairwise disjoint.*
- *A node of* I *has no parents in* $\mathbf{C}$*, no children outside* $\mathbf{C}$ *and can have at most one referenced node outside* $\mathbf{C}$*.*
- *A node of* P *has neither children nor parents outside* $\mathbf{C}$*.*
- *A node of* O *has no parents outside* $\mathbf{C}$*.*

In OOBNs *à la Bangsø & Wuillemin*, nodes can represent different notions, most of them exist in OOBNs *à la Pfeffer*. They can represent encapsulated instances in a class, classic random variables and references. References is somewhat a generalization of OOBNs *à la Pfeffer* input nodes and give access to nodes defined in other classes.

Definition 3.24 (Node) *A node in an OOBN is either:*

- *an instance node: a node representing the instance of a class inside another class.*
- *a simple node: a node which is either a:*
  - *a reference node: a reference node can have only one referenced node and is a place-holder for it. An optional default prior distribution is used when no referenced node has been specified. A reference node cannot have parents, but it can have children. All input nodes are reference nodes. A protected node cannot be a reference node.*
  - *a regular node: the same as a node in BNs.*

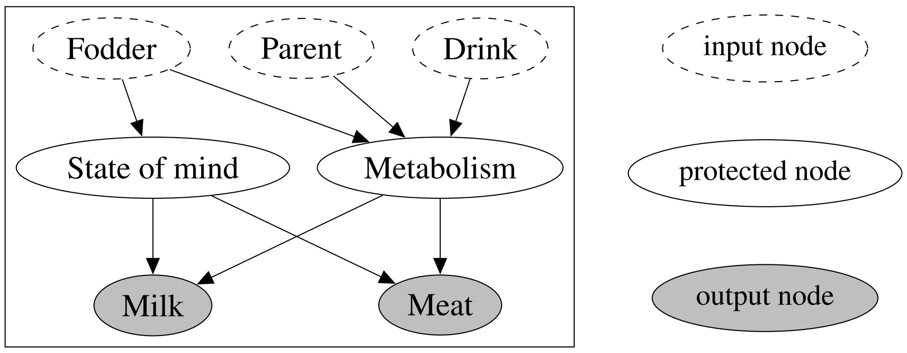


Figure 3.9: The different kinds of nodes in a class (Bangsø, 2004).

Figure 3.9 illustrates the precedent definitions. Classes are not different from object-oriented network fragments, yet OOBNs *à la Bangsø & Wuillemin* formalization helps modeling object-oriented domains as there is no ambiguity between classes and instances as in OOBNs *à la Pfeffer*. However, this framework takes a special care in defining the links between simple nodes and reference nodes, a topic left undefined in Pfeffer's framework. Formalizing the relationships between simple and reference nodes is done using different types of links.

Definition 3.25 (Link) *A link is either a:*

- *a directed link (simple node $\rightarrow$ regular node): links as in normal BNs.*
- *a reference link (simple node $\dashrightarrow$ reference node): $A \dashrightarrow B$ means "A is referenced by B". B is a reference node, A is the referenced node.*

Since reference nodes cannot have parents (in the sense of BNs), the type of link is given by the type of the node pointed to by the link. Figure 3.10 is a smaller version of the cow stock example (Bangsø and Wuillemin, 2000b; Bangsø, 2004) and illustrates the different types of links allowed in OOBNs *à la Bangsø & Wuillemin*. This example models a small stock of cows and estimates the expected income it will generate. Again,

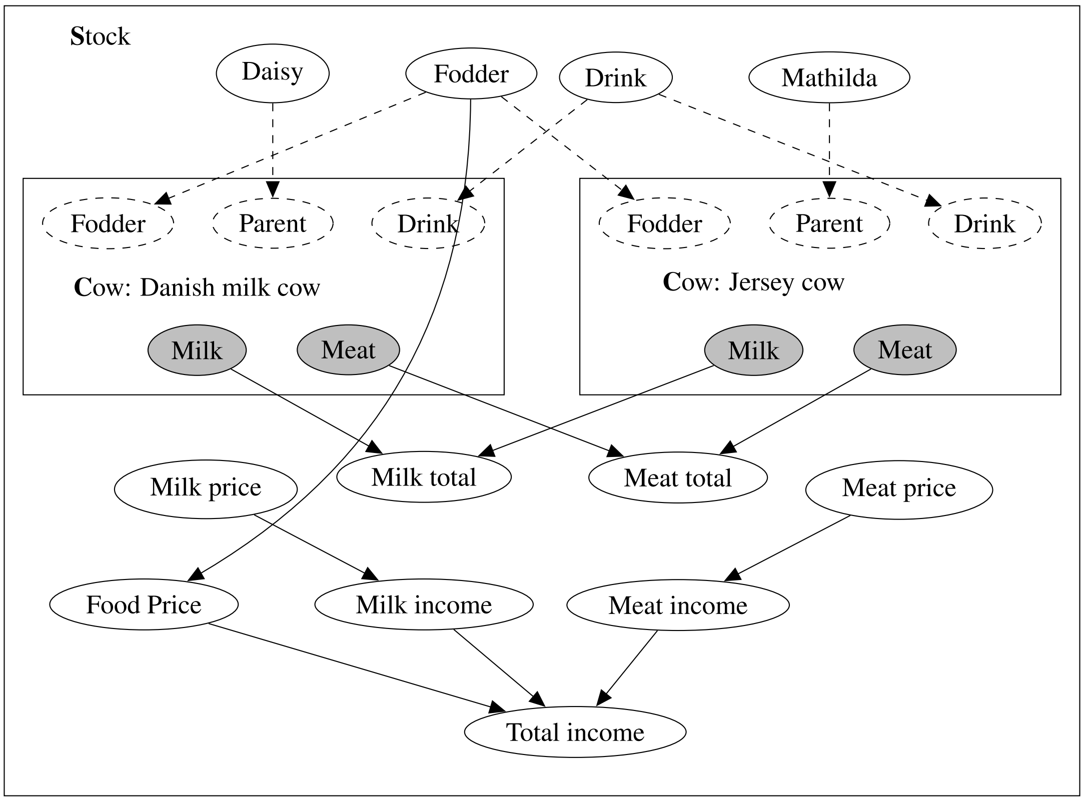


Figure 3.10: An instantiation of figure 3.9 in the cow stock example (Bangsø, 2004).

there are not many differences with OOBNs *à la Pfeffer*. The most important one being the existence of reference links and the constraints in the construction of the systems. Each set of nodes connected via reference links forms a tree. The constraints that prevent from creating inconsistent probability models, i.e., with cyclic dependencies, are defined using that reference tree. To prevent ill-conceived systems and to offer modeling guidelines the following rules restraining reference definitions are proposed in Bangsø (2004):

Rule 3.1 (Reference tree restriction) *The reference tree restriction entails that:*

1. *Input nodes cannot be used as referenced nodes for simple nodes specified in the same class but only for input nodes of (other) instances. Two input nodes in an instance cannot have the same referenced node.*

2. *Output nodes can be used as referenced nodes for output nodes in the encapsulating class and for input nodes in a different instantiation in the encapsulating class. Two output nodes of a class cannot have the same referenced node.*

3. *Protected nodes can be used as referenced nodes for input nodes of instances only. Protected nodes will always be referenced roots if they occur in a reference tree, as they can never be reference nodes.*

4. *A chain of reference links can go in both directions (further inside instances or further out) but once it begins going inside, it cannot go out again.*

The first rule entails that for two inputs nodes I_i and I_j of the same class, I_j cannot refer to I_i. This constraint prevents ill constructed classes, since there is not much sense in modeling a class with such feature. Furthermore, preventing two input nodes from referring the same node when the class is instantiated follow the same concept. If we allowed such constructions it could lead to unexpected loops since inputs nodes are supposed to be distinct nodes. If it is not the case, then there is no reasonable justification to have two input nodes instead of one. The second rule limits output nodes visibility. This prevents a class to gain access to an output node bypassing its encapsulating class. For example suppose we have three classes A, B and C such that B is encapsulated by A and has the output node X. Now, if we model C such that it requires an input attribute matching X, it cannot link its input node directly to B's output node. We must first add a reference output node to A that references X, name it X_A. Since A and C are at the same encapsulation level, we can connect C's input attributes with X_A. The third rule is derived from protected nodes definition, i.e., they are neither input nor output nodes. Since they cannot be output nodes, they can only be referenced by input nodes of encapsulated instances in their class. The final rule prevents ambiguous situations where two nodes at the same level will in fact be the same random variable in the associated probability distribution. In such situation, modeling undesired cyclic dependencies can be achieved without noticing it.

Dynamic OOBNs *à la Bangsø & Wuillemin*

OOBNs *à la Bangsø & Wuillemin* offer basic dynamic BNs modeling. To do so, temporal links are added to create links between input and protected nodes that are normally illegal. Such links are called temporal dependence links. These links represent dependencies between time slices and only exist between two adjacent time slices, thus they are discarded when checking for acyclicity.

Figure 3.11 is an example of a time slice class. The dashed links are temporal dependence links, they represent the probabilistic dependencies between time slices T_t and T_{t+1}. For each temporal dependence link $X \dashrightarrow Y$, where Y is an input node and X either an output or protected node, the node X in time slice T_t will be referenced by the input node Y of time slice T_{t+1}.

Class inheritance

We have seen that OOBNs *à la Pfeffer* force constraints on the input nodes of subclasses. OOBNs *à la Bangsø & Wuillemin* allow to define default probability distributions for input nodes. Doing so relax the OOBNs *à la Pfeffer* constraint over subclasses input nodes. Thus, they offer a generalization of OOBNs *à la Pfeffer* class inheritance mechanism.

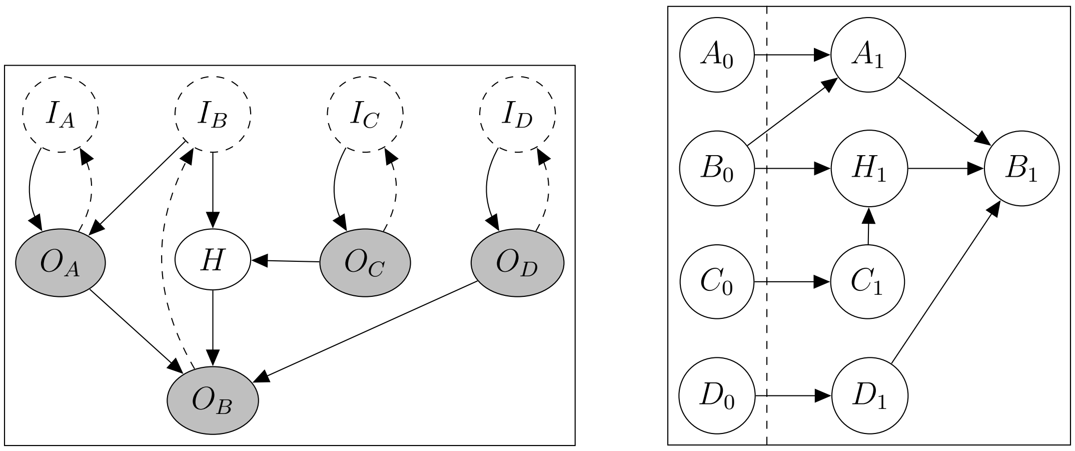


Figure 3.11: A time slice class with temporal dependence links (dashed arcs) and its associated 2-TBN.

Definition 3.26 (Class inheritance) *A class S over $(I_S \cup P_S \cup O_S, E_S)$ can be a subclass of a class C over $(I_C \cup P_C \cup O_C, E_C)$ (where C is a superclass of S), denoted $S \rhd C$ if $I_C \subseteq I_S$, $P_C \subseteq P_S$ and $O_C \subseteq O_S$.*

Definition 3.26 offers a more sound definition to class inheritance. Indeed, we want to use class inheritance to specialize classes by creating subclasses of them. The most straightforward manner to do that is to add nodes, either input, protected or output nodes, to the subclass.

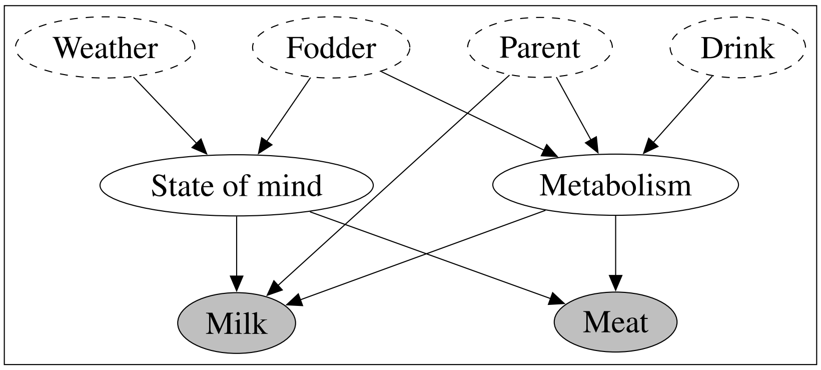


Figure 3.12: A subclass of the class of figure 3.9.

Figure 3.12 is a subclass of the *Cow* class of figure 3.9. As we can see, this subclass has more arcs that the class of figure 3.9. This implies that some CPTs have been specialized, i.e., they are a more precise probabilistic representation of the attributes of the specialized class of figure 3.12. Such feature is not present in OOBNs *à la Pfeffer*.

Definition 3.27 (Conditional probability table's scope) *The scope of the CPT associated with the node X_T, where C_T is the class where the node is originally defined for the first time, is a substructure of the class tree with C_T as the root. Each subclass of C_T is a member of the scope if and only if the CPT is not overwritten in that subclass or in one of its superclass.*

We will see that overwriting CPTs in subclasses is much more interesting than analyzing in which class it is unmodified. Yet such scope can be useful to prevent unnecessary memory usage. However, its use in inference is limited since we can hope reusing computations only if a class and its subclass share the same conditional probability distribution, which does seem compatible with class inheritance use.

Inference

Unfortunately, there is not much to say about inference in OOBNs *à la Bangsø & Wuillemin* from the perspective of structural inference. The main inference approaches consist in either grounding the system into a BN or transforming it into a MSBN (Bangsø, 2004).

OO Bayesian Networks flaws and shortcomings

OOBNs are the first BN extension based on a paradigm representation shift. Fragmenting BNs into instances of different classes helps with modeling complex systems and structural inference seems to considerably speed-up inference. We have seen that both OOBNs *à la Pfeffer* and OOBNs *à la Bangsø & Wuillemin* offer similar expressive power, while Pfeffer's framework focuses more on the formalization, Bangsø & Wuillemin priority is to offer a sound framework for modeling complex systems. Let us recall the five object-oriented desiderata introduced at the beginning of this chapter: (i) objects are defined by classes and used as instances, (ii) class inheritance, (iii) abstraction, (iv) subtype polymorphism, and (v) recursive data types. Let us consider how each of these desiderata is more or less well satisfied in either OOBNs *à la Pfeffer* or OOBNs *à la Bangsø & Wuillemin*.

For the first desideratum, we saw that OOBNs *à la Pfeffer* can use the notion of class but it is defined awkwardly and the difference among classes, objects and instances is very ambiguous. OOBNs *à la Bangsø & Wuillemin* offer a more sound formalization, yet there is some ambiguity about the presence of instances in classes. Indeed, in such situation the system is modeled as a unique class. Instances are only found in their encapsulating class. The fact that an instance can be connected to a class is an undesired feature, we would prefer specify the classes then instantiate them and connect the instances in a specific entity that is not a class. The second desideratum implementation is also unsatisfactory. For example, it is not possible to specialize input or output nodes, i.e., use a more specialized random variable type. This reveals another problem: random variable type specialization. Suppose we model some generic equipment with a failure node with the domain $\{\text{functional}, \text{dysfunctional}\}$. If we create a subclass to model a printer, we may want to specialize its failure states to {functional, no ink, paper jam}. Both the absence of ink and a paper jam are specific case of printers dysfunction, but such specialization is impossible with both frameworks. Abstraction in PGM consists in specifying a class with omitting probabilistic semantics. OOBNs *à la Pfeffer* complex objects are close to this notion: it provides the names of attributes that are present in any object-oriented network fragment over that object. However, complex objects must specify all the attributes that will be present in the associated object-oriented network fragment. It is also unclear if complex objects are defined by a structured type or if the type is inferred from the object. Ideally, we want to only specify the attributes that will be used by other classes, i.e., only specify output nodes. In both OOBN versions it is possible to proceed with subtype polymorphism since input nodes do not take into account the class to which the node belongs. However, if we allow random variable type specialization this form of subtype polymorphism is inefficient. Finally, the fifth desideratum is present in OOBNs *à la Bangsø & Wuillemin* but it is somewhat complicated. Ideally we would want to handle dynamic systems without introducing specific links. Thus allowing recursive data types creates the need of an efficient loop detection algorithm. We will see that such issue is correctly dealt with in Probabilistic Relational Models.

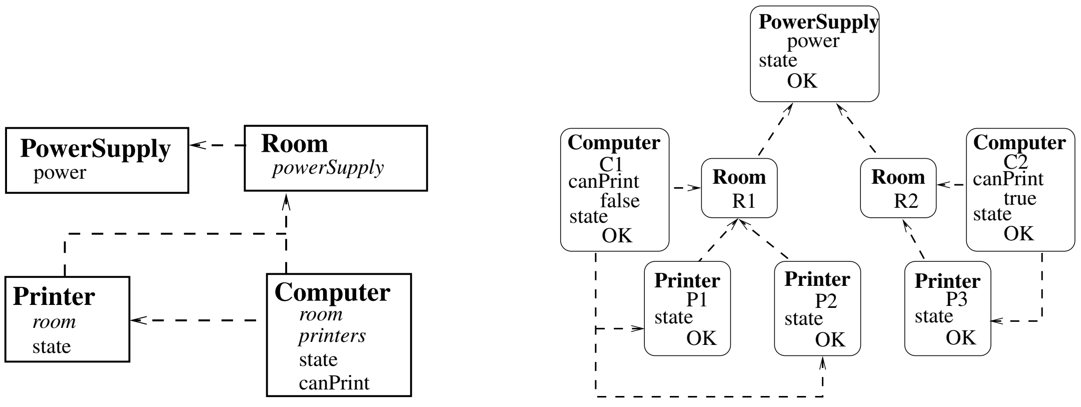
3.3 Probabilistic Relational Models

After introducing the object-oriented paradigm in BNs, the community focus have been set on how to define relationships among BN fragments. The entity-relationship paradigm was then used to extend the preliminary work on OOBNs *à la Pfeffer*. This new paradigm shift was initiated for machine learning purpose (Getoor and Taskar, 2007). Despite its intent to be used for machine learning, we will see that Probabilistic Relational Models (PRMs) fix many OOBNs *à la Pfeffer* flaws. PRMs are an extension of OOBNs *à la Pfeffer* (Friedman et al., 1999; Pfeffer, 1999). They extend Pfeffer's framework by introducing reference slots, a notion both inherited by frames, plate models and the entity-relationship paradigm (Wellman et al., 1992; Buntime, 1990; Koller and Pfeffer, 1998; Heckerman et al., 2007). An important feature of PRMs is structural uncertainty that we detail at chapter 3.3. As for OOBNs *à la Pfeffer*, PRMs exploit a form of structured inference.

Basic definitions

PRMs are inspired by relational languages, where the focus is set on classes of objects and by defining relations among these objects. There are many different graphs which represent PRMs, one among others are relational schema. Borrowed from the entity-relationship paradigm, they describe attributes of each class and define relationships among classes. As we will see, if a relation exists between two classes it does not necessary entails a probabilistic dependency. Figure 3.13a is a relational graph representing the power surge example. Note that a relational schema is only a pattern to an infinite number of possible worlds, for example figure 3.13b illustrates a possible world for the relational schema of figure 3.13a.

Example 3.1 (The power surge example) *The power surge example is a computer maintenance problem where a set of computers and printers, located in different rooms, are powered by one or more power supplies. Breakdowns are caused by power surges, also known as voltage spikes. They can be caused by lightning strikes, power outages, tripped circuit breakers, short circuits, power transitions in other large equipment on the same power line, malfunctions caused by the power company, etc. Depending on the voltage spike intensity and the overall quality and age, electronic devices, i.e., computer and printer, may breakdown after a power surge. Each equipment is described by its current state (*state*: $\{OK, NOK\}$). For computers, an additional attribute shows if we can print (*canPrint*: $\{can, cannot\}$). Finally, power supplies are described by their power state (*power*: $\{on, off, surge\}$), the surge state showing that at least one power surge occurred in the last 24 hours. Finally, a computer is usually plugged to one or more printers, it can print if at least one of its printers is functional.*



(a) A relational schema. (b) A possible instantiation of figure 3.13a.

Figure 3.13: In figure 3.13a a relational schema representing different objects and their relationships. In figure 3.13b, a possible instantiation of the schema represented in 3.13a. Rectangles are classes and rounded rectangles are objects. Dashed lines denote existing relations among classes (or objects). Classes are described using attributes and reference slots. For instances, they are assigned values.

PRMs are described using a set of classes, $\mathcal{X} = \{X_1, \cdots, X_n\}$. Each class X is associated with a set $\mathcal{A}(X)$ of attributes and a set $\mathcal{R}(X)$ of reference slots. An attribute A of class X is denoted $X.A$, and its space of values, i.e., the domain of the random variable it is representing, is denoted Val$(X.A)$. Remarkably, reference slots are not formally defined in any of PRMs presentations. This is surprising as it is their main feature. But, when we try to define them, we are confronted with several semantic issues. In Pfeffer (1999) a solution is to introduce them by defining a *relational knowledge base*. However, we find this approach to be too complicated: it introduces too many intermediate definitions that do not prove to be of any practical use. In chapter 4 we will propose our definition of reference slots and until then we will use the notions introduced in Getoor et al. (2007). We use $X.\rho$ to denote the reference slot $\rho \in \mathcal{R}(X)$. Reference slots are typed, i.e., for each $\rho \in \mathcal{R}(X)$ its domain type $domain(\rho)$ equals X and its range type $range(\rho)$ equals Y, where Y is a class in $\mathcal{X}$. The inverse reference ρ^{-1} of $\rho \in \mathcal{R}(X)$ is the reference slot such that: $domain(\rho^{-1}) = range(\rho)$ and $range(\rho^{-1}) = domain(\rho)$ and is defined in $\mathcal{R}(Y), Y \in \mathcal{X}$.

A reference defines a 1-to-n relation (one computer connected to many printers), in some cases, we are guaranteed that the relation is a 1-to-1 (one computer in one room). Since 1-to-n relations require specific treatment regarding probabilistic semantics, we will always differentiate single reference slots (1-to-1) from multiple ones (1-to-n). Note that the inverse of single reference slot is not necessarily single (for example the relation between *Computer* and *Room* is single, however its inverse is multiple). A slot chain $\mathbf{K}$ is a sequence $\rho_1, \cdots, \rho_k$ of reference slots such that for all i, $Range[\rho_i] = Dom[\rho_{i+1}]$. The inverse slot chain of $\mathbf{K}$ is the slot chain $\mathbf{K}^{-1}$ such that $\mathbf{K}^{-1} = \rho_k^{-1}, \cdots, \rho_1^{-1}$. We will say that a slot chain is single if all its reference slots are single, otherwise it is multiple. For a modeling perspective, slot chains are user defined while inverse slot chains are automatically inferred and used for inference purpose (see chapter 5.2).

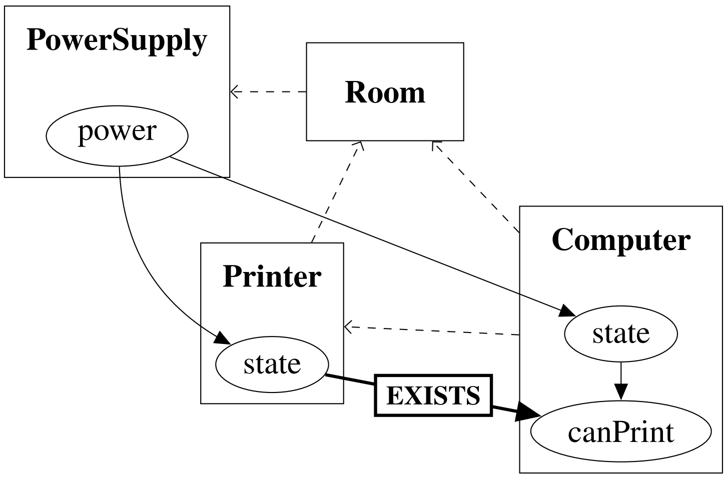


Figure 3.14: Attributes parents are either attributes of the same class or attributes obtained through a slot chain. If the slot chain is multiple then it is necessary to aggregate the parents information obtained using aggregators.

Figure 3.14 illustrate all these notions. Attributes are represented as nodes and probabilistic dependencies with arcs. We can see that classes *Computer* and *Printer* access *PowerSupply.power* using their reference slots over *Room*: *Printer.room.powerSupply.power* is the parent of *Printer.state*; *Computer.room.powerSupply.power* and *Computer.state* are parents of *Computer.canPrint*. Note that the arc between *Computer.canPrint* and *Printer.state* shows that it is a multiple relation. For probabilistic dependencies defined using multiple slot chains, it is necessary to aggregate the information received. Here we merge the data from all printers connected to a computer such that we obtain the value *true* if at least one printer is working and *false* otherwise. Aggregators are an important feature of PRMs and we will detail them further shortly. We must remember that a relational schema such as figure 3.14 is a pattern to an infinite number of possible worlds (also called instantiation of the schema). Until now, we have been elusive on how the probability distribution represented by a PRM is defined, we will now detail this aspect.

Definition 3.28 (Instance of a relational schema) *An instance $\mathcal{I}$ of a relational schema specifies: for each class X, the set of objects of X in $\mathcal{I}$, denoted $\mathcal{I}(X)$; a value for each attribute $x.A$ (in the appropriate domain) for each object x; a value y for each reference slot $x.\rho$, which is an object in the appropriate range type, i.e., , $y \in \mathcal{I}(Range[\rho])$. Conversely, $y.\rho^{-1} = \{x|x.\rho = y\}$. We use $\mathcal{A}(x)$ as a shorthand for $\mathcal{A}(X)$ and $\mathcal{R}(x)$ as a shorthand for $\mathcal{R}(X)$, where x is of class X. For each object x in the instance and each of its attributes A, we use $I_{x.A}$ to denote the value of $x.A$ in I.*

The previous definition needs some clarification about the vocabulary used. Usually an instance is always the instantiation of something, either a class, a schema or an attribute. In the PRM terminology, this is not the case: an instance is only the use of a relational schema to model a specific situation (called a world), as in figure 3.13b (Getoor et al., 2007). An instance does not have any uncertainty since all its attributes are observed (which is the purpose of the $I_{x.A}$ notation which should have been denoted $x.A = a$ or $x.a$ to preserve BNs classic notations). An object denotes the use of a class in an instance. There is no term dedicated to denote attributes and reference slots of instances, e.g., we cannot say that $x.A$ is the instantiation of $X.A$ in x. In chapter 4.1, we will introduce a more standard object-oriented terminology and discuss the practical reasons of our choices. In the remainder of our presentation of PRMs we will denote by the term *object* class instances.

Definition 3.29 (Relational Skeleton) *A relational skeleton σ_r of a relational schema is a partial specification of an instance of the schema. It specifies the set of objects $\sigma_r(X_i)$ for each class and the relations that hold between the objects. However, it leaves the values of the attributes unspecified.*

A relational skeleton σ_r represents a world for which we have partial information: we know the number of objects and their relationships but attributes values are uncertain. A relational skeleton serves the same purpose as BNs for reasoning under uncertainty, i.e., it factorizes a joint probability distribution. Introducing uncertain knowledge using probabilities, also called the probabilistic semantics of a PRM, is achieved using two components: a qualitative dependency structure S and the parameters θ_S associated with it. Both of them are illustrated in figure 3.14: the dependency structure defines the probabilistic dependencies for each attribute, i.e., their set of parents, denoted $\pi(X.A)$. Parameters are defined using CPTs. Recall that at class level we are defining a pattern, thus $\pi(X.A)$ stands for the formal parents of $X.A$ and for each object x of X, the set $\pi(x.A)$ will often be specific for each one of them.

Example 3.2 *Let us consider figure 3.13b and suppose that all attributes are unobserved. Figure 3.14 informs us that $Computer.canPrint$'s parents are $Computer.state$ and $Computer.printers.state$. Now consider object $C1$ in figure 3.13b, $C1.canPrint$'s parents are $C1.state$, $P1.state$ and $P2.state$. Note that $C1.canPrint$'s parents are different from $C2.canPrint$'s parents: $C2.state$ and $P3.state$.*

Attributes parents are either in the same class or reached from another one using a slot chain. Where the first case is standard, the second needs further explanations. We denote by $x.\mathbf{K}$ the set of objects that are $\mathbf{K}$-relatives of x, i.e., the set of objects accessible through the slot chain $\mathbf{K}$. Thus we need to define the probabilistic dependencies of $x.A$ using the multiset $\{y.B : y \in x.\mathbf{K}\}$. If the slot chain is single, then CPTs can be used. If not, we need to aggregate the information contained in the multiset (recall the arc labeled *exists* between *Computer.canPrint* and *Printer.state* in figure 3.14). This is where the notion of aggregation from database theory is used to answer this issue. It consists in using probabilistic or deterministic functions to create on-the-fly CPTs case-by-case, i.e., given the number $|\{y.B : y \in x.\mathbf{K}\}|$. Such functions are already used by experts to model BNs: noisy-or, k-gates, count, mean, min, max, exists, for all, binomial distributions, Poisson distributions, etc.

Definition 3.30 (Aggregate) *An aggregate γ takes a multiset of values of some ground type, and returns a summary of it in the form of a discrete multi-valued attribute. The type of the aggregate can be the same as that of its arguments. However, we allow other types as well, e.g., , an aggregate that reports the size of the set (with an upper and lower bounds on the possible sizes). We allow $X.A$ to have as a parent $\gamma(X.K.B)$; the semantics is that for any $x \in X$, $x.A$ will depend on the value of $\gamma(x.K.B)$.*

A ground type denotes attributes with identical domains, i.e., two attributes with identical ground type are two distinct random variables with identical domains. Aggregates are used to reduce an unknown number of attributes to a single one, enabling definitions of CPTs at class level. With this, we can now give a formal definition of PRMs.

Definition 3.31 (Probabilistic Relational Model) *A PRM Π for a relational schema R is defined as follows. For each class $X \in \mathcal{X}$ and each attribute $A \in \mathcal{A}(X)$, we have:*

- *a set of parents $\pi(X.A) = \{U_1, \cdots, U_l\}$, where each U_i has the form $X.B$ or $\gamma(X.\mathbf{K}.B)$, where $\mathbf{K}$ is a slot chain and γ an aggregate;*
- *a legal conditional probability distribution (CPD), $P(X.A|\pi(X.A))$.*

Given a PRM Π and a relational skeleton σ_r, we can build the ground BN of $\{\Pi, \sigma_r\}$.

Definition 3.32 (Ground BN) *A PRM Π and a relational skeleton σ_r define the following ground BN: there is a node for every attribute of every object $x \in \sigma_r(X), x.A$. Each $x.A$ probabilistically depends on parents of the form $x.B$ or $x.\mathbf{K}.B$. If $\mathbf{K}$ is not single-valued, then the parent is the aggregate computed from the set of random variables $\{y|y \in x.\mathbf{K}\}, \gamma(x.\mathbf{K}.B)$. The CPD for $x.A$ is $P(X.A|\pi(X.A))$.*

We can then write an equation equivalent of 1.2 for PRMs, i.e., a factorized expression of the probability distribution represented by $\{\Pi, \sigma_r\}$. We express it as the product over each attribute of each object in σ_r of the attribute's CPT. CPTs are inferred from the probabilistic semantics of each class in Π. We denote by $\sigma_r(X)$ the set of objects of X in σ_r.

$$\begin{aligned} P(I|\sigma_r, S, \theta_S) &= \prod_{x \in \sigma_r} \prod_{A \in \mathcal{A}(x)} P(I_{x.A}|I_{\pi(x.A)}) \\ &= \prod_{X \in \mathcal{X}} \prod_{A \in \mathcal{A}(X)} \prod_{x \in \sigma_r(X)} P(I_{x.A}|I_{\pi(x.A)}) \end{aligned} \tag{3.1}$$

Detecting cycles in Probabilistic Relational Models

PRMs are directed PGMs and, as for BNs, cyclic dependencies are forbidden. The fact that a relational schema represents an infinite number of worlds can be problematic if, for some of them, cyclic dependencies can be found. It is possible to detect such invalid worlds during the modeling process for small systems, however it is nearly impossible for complex ones. Generating the ground BN for a given instance is one solution to check for cycles, but it is possible to guarantee the probabilistic coherence of a system using dependency graphs. Furthermore, these dependency graphs are useful as a modeling tool to exhibit probabilistic dependencies.

Definition 3.33 (Instance dependency graph) *The instance dependency graph G_{σ_r} for a PRM Π and a relational skeleton σ_r has a node for each attribute of each object $x \in \sigma_r(X)$ in each class $X \in \mathcal{X}$. Each $x.A$ has the following edges:*

1. *Type I edges: for each formal parent of $x.A$, $X.B$, we introduce an edge from $x.B$ to $x.A$.*
2. *Type II edges: for each formal parent $X.\mathbf{K}.B$, and for each $y \in x.\mathbf{K}$, we define an edge from $y.B$ to $x.A$.*

Theorem 3.2 *Let Π be a PRM whose dependency structure S is acyclic relative to a relational skeleton σ_r. Then Π and σ_r define a coherent probability distribution over instantiations I that extend σ_r via 3.1.*

Proof omitted (see chapter 5 in Pfeffer (1999)). This result is unsurprising since the instance dependency graph topology is closely related to the ground BN, the difference being in the absence of aggregate representation. However, we can also infer acyclic properties of a PRM from its classes only.

Definition 3.34 (Class dependency graph) *The class dependency graph G_Π for a PRM Π has a node for each descriptive attribute $X.A$, and the following edges:*

1. *Type I edges: for any attribute $X.A$ and any of its parents $X.B$, we introduce an edge from $X.B$ to $X.A$.*

2. *Type II edges: for any attribute $X.A$ and any of its parents $X.\mathbf{K}.B$ we introduce an edge from $Y.B$ to $X.A$, where $Y = Range[X.\mathbf{K}]$.*

Theorem 3.3 *If the class dependency graph G_Pi is acyclic for a PRM Π, then for any skeleton σ_r, the instance dependency graph is acyclic.*

Proof omitted (see chapter 5 in Pfeffer (1999)).

Corollary 3.1 *Let Π be a PRM whose class dependency structure S is acyclic. For any relational skeleton σ_r, Π, and σ_r define a coherent probability distribution over instantiations I that extend σ_r via 3.1.*

In some situation, cycles can appear in the class dependency graph that are not present in the instance dependency graph. This will often be the case for temporal or recursive models. To represent such models using PRMs, we must guaranty the absence of cycles.

Definition 3.35 (Acyclic guaranty) *If reference slots $\mathcal{R}_{ga} = \rho_1, \cdots, \rho_k$ are guaranteed acyclic, we guaranty the existence of a partial ordering $\prec_{ga}$ such that if y is a ρ-relative for some $\rho \in \mathcal{R}_{ga}$ of x, then $y \prec_{ga} x$. We say that a slot chain $\mathbf{K}$ is guaranteed acyclic if each of its component ρ's is guaranteed acyclic.*

A simple way to represent these reference slots and slot chains is to use edge coloring. The edge coloring results in a colored class dependency graph that describes the direct dependencies among attributes.

Definition 3.36 (Colored class dependency graph) *The colored class dependency graph G_Π for a PRM Π has the following edges:*

1. *Yellow edges: if $X.B$ is a parent of $X.A$, we have a yellow edge $X.B \rightarrow X.A$.*

2. *Green edges: if $\gamma(X.\mathbf{K}.B)$ is a parent of $X.A, Y = Range[X.\mathbf{K}]$, and $\mathbf{K}$ is guaranteed acyclic, we have a green edge $Y.B \rightarrow X.A$.*

3. *Red edges: if $\gamma(X.\mathbf{K}.B)$ is a parent of $X.A, Y = Range[X.\mathbf{K}]$, and $\mathbf{K}$ is not guaranteed acyclic, we have a red edge $Y.B \rightarrow X.A$.*

Theorem 3.4 *Given a PRM Π, if every cycle in the colored class dependency graph for Π contains at least one green edge and no red edges, then for any skeleton σ_r, the instance dependency graph is acyclic.*

Proof omitted (see chapter 5 in Pfeffer (1999)).

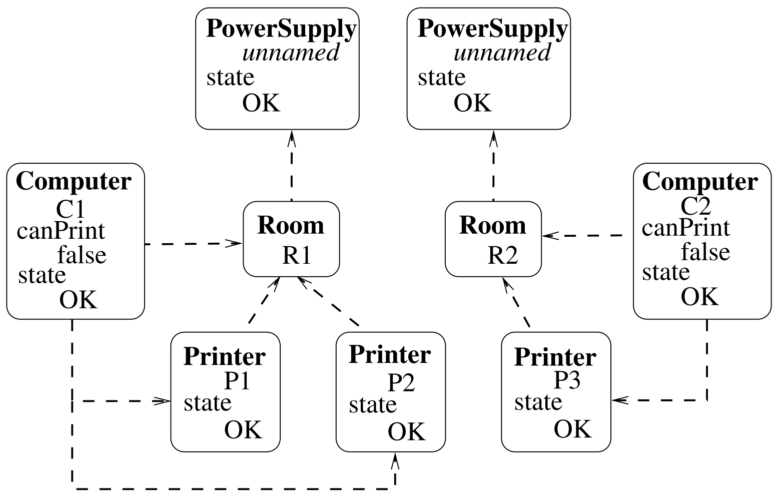


Figure 3.15: A relational skeleton with unnamed objects in which a tree-like world generation was used.

Structural uncertainty

BNs represent uncertainty: we describe a world with descriptive attributes and assign conditional probability distributions to them. BNs do not offer any mechanism to encode uncertainty about the existence of probabilistic relations: there is no uncertainty about the set of parents of a random variable. For some applications, we may want to represent such uncertainty over the structure of a BN. PRMs offer different solutions to represent structural uncertainty.

In PRMs we distinguish two sorts of objects: named and unnamed objects. Named objects are user-defined objects and represent the fact that we are certain about the existence of that object in our knowledge base. When all objects in a world are named, we are modeling a closed world. Sometimes, we may have insufficient named objects to derive a valid probability distribution. In such situation, we are modeling an open world: a world in which we must generate unnamed objects until we obtain a valid probability distribution. For example, in figure 3.14 we pretend that we could not name the object *power* of the *PowerSupply* class. In such case we must define rules to generate possible worlds. In Pfeffer (1999), a constraint is to force tree-like structures. In figure 3.14 we remove the named object *power*, this results in the open world illustrated in figure 3.15. However, generation rules influence greatly the resulting worlds. It is necessary to have a good understanding of the domain and good feedbacks from experts.

Open worlds and generating unnamed objects are a simplistic structural uncertainty feature. PRMs offer two notions to represent more complex structural uncertainty (Pfeffer, 1999): reference and existence uncertainty. Reference uncertainty represents uncertain knowledge about the existence of reference links among objects. A straightforward solution would be to assign a probability value to each possible link, however the number of such links grows exponentially with the number of objects. To reduce the number of required parameters, PRMs use a partition function and then consider the existence

	Operating System		
Room Partition	Apple	Linux	Windows
Secretary	0.5	0.0	0.34
Class Room	0.1	0.8	0.33
Prof. Office	0.4	0.2	0.33

Table 3.1: Representing reference uncertainty using selector attributes and partition functions.

of a reference link between an object and an object in one partition. In each partition, objects are considered equally eligible to receive the reference link.

Example 3.3 *Let us consider the power surge example. To express uncertainty about the location of computers, i.e., uncertainty about the existence of reference links between objects of the* Computer *class and objects of the* Room *class, we must first partition rooms. Let us suppose that we are in a university and we have the following different rooms: secretary, class rooms and professors office; and different operating systems: Windows, Ubuntu and MacOS. By introducing a selector attribute for reference $Computer.room$, we can define a conditional probability distribution as in table 3.1 which expresses the probability of a computer being in a given type of room depending on its operating system.*

The following is a formal definition of reference uncertainty in PRMs:

Definition 3.37 (Reference uncertainty) *A PRM Π with reference uncertainty over a relational schema $\mathcal{R}$ has the same components as in definition 3.31. In addition, for each reference slot $\rho \in \mathcal{X}$ with $Range[\rho] = Y$, we have:*

- *a partition function ψ_ρ;*
- *a selector attribute $S_\rho \in \mathcal{A}(X)$ which takes on values in the range of ψ_ρ;*
- *a set of parents and a conditional probability distribution for S_ρ.*

We can include reference uncertainty in equation 3.1 as follow:

$$P(I|\sigma_o, \Pi) = \prod_{X \in \mathcal{X}} \prod_{x \in \sigma_o(X)} \prod_{A \in \mathcal{A}(X)} P(x.A|\pi(x.A)) \prod_{\rho \in \mathcal{R}(X), y = x.\rho} \frac{P(x.S_\rho = \psi[y]|\pi(x.S_\rho)}{|\mathcal{I}(\mathbf{Y}_{\psi[y]})|}, \tag{3.2}$$

where σ_o is an object skeleton (an incomplete relation skeleton), $\psi[y]$ refers to the partition assigned to y and $|\mathcal{I}(\mathbf{Y}_{\psi[y]})|$ the number of objects in partition $\psi[y]$. Note that when using reference uncertainty, we use the selector attribute only when no reference link is defined between the two classes. Thus, in the previous example, all rooms are considered as possible candidates for a given computer. The partition function helps reducing the number of parameters since instead of defining a CPT for all existing rooms, we use the different partitions. Since we suppose that all objects in a partition are equally probable to be connected to a computer we can infer the probability for a given object to be connected to the computer. Note that the equiprobability assumption is arbitrary and nothing prevents us to choose a more subtle probability distribution (except the number of required parameters).

The second form of structural uncertainty is called existence uncertainty. Here, we refer to the entity-relationship paradigm and distinguish entity classes from reference classes. A special boolean attribute E, called the existence attribute, is added to each reference class. We differentiate determined class from undetermined ones by either setting $X.E$ to true or by assigning it a probability distribution. As for classic attribute, these specific existence attributes can have a set of parents. Given an entity skeleton σ_e, which specifies the set of objects in our domain only for entity classes, we define the induced relational skeleton $\sigma_r[\sigma_e]$ as follows:

Definition 3.38 (Induced relational skeleton) *Consider a schema with determined and undetermined classes, and let σ_e be an entity skeleton over this schema. We define the induced relational skeleton, $\sigma_r[\sigma_e]$, to be the relational skeleton that contains the following objects:*

- *If X is a determined class, then $\sigma_r[\sigma_e](X) = \sigma_e(X)$.*
- *Let X be an undetermined class with reference slots $\rho_1, \cdots, \rho_k$ whose range types are $Y_1, \cdots, Y_k$ respectively. Then $\sigma_r[\sigma_e](X)$ contains an object $X[y_1, \cdots, y_k]$ for all tuples $\langle y_1, \cdots, y_k\rangle \in \sigma_r[\sigma_e](Y_1) \times \cdots \times \sigma_r[\sigma_e](Y_k)$.*

The relations in $\sigma_r[\sigma_e]$ are defined in the obvious way: slots of objects of determined classes are taken from the entity skeleton; slots of objects of undetermined classes are induced from the object definition: $X[y_1, \cdots, y_k].\rho_i$ is y_i.

Simply put, an induced relational skeleton is a relational skeleton in which we added all combinations of possible references represented by undetermined classes. To define the probability distribution of PRM using existence uncertainty, we must define two constraints over undetermined classes. If for one slots ρ of X, $x.\rho.E = false$ then $x.E = false$. To implement such constraint we redefine the conditional probability distribution of $X.E$ such as:

$$P^*(X.E|\text{Pa}^*(X.E)) = \begin{cases} P(X.E|\pi(X.E)) & \text{if} X.\rho_i.E = true, \forall i = 1, \cdots, k, \\ 0 & \text{otherwise} \end{cases}$$

where $\text{Pa}^*(X.E) = \pi(X.E) \cup \{X.\rho_1.E, \cdots, X.\rho_k.E\}$. The second constraint prevents from taking into account attributes of nonexistence objects. If $X.A$ is an attribute and $\text{Pa}^*(X.A) = \pi(X.A) \cup \{X.E\}$ we redefine $X.A$ conditional probability distribution as follow:

$$P^*(X.A|\text{Pa}^*(X.A)) = \begin{cases} P(X.A|\pi(X.A)) & \text{if} X.E = true, \\ \frac{1}{|\text{Val}(X.A)|} & \text{otherwise.} \end{cases}$$

Given an entity skeleton σ_e, a PRM Π with existence uncertainty specifies a distribution over a set of instantiations $\mathcal{I}$ consistent with $\sigma_r[\sigma_e]$:

$$P(\mathcal{I}|\sigma_e, \Pi) = P(\mathcal{I}|\sigma_r[\sigma_e], \Pi^*) = \prod_{X \in \mathcal{X}} \prod_{x \in \sigma_r[\sigma_e](X)} \prod_{A \in \mathcal{A}(x)} P^*(x.A|\text{Pa}^*(x.A)).$$

To end our presentation of structural uncertainty in PRMs we will remark that these features can be simulated in BNs by assigning specific CPTs. However, this can be difficult to model such systems and by introducing such features, PRMs reduce the modeling costs of such systems.

Inference in Probabilistic Relational Models

Inference in PRMs is closely related to inference in OOBNs *à la Pfeffer*. The SVE algorithm has been adapted to PRMs (Pfeffer, 1999) and exploits open worlds to reduce inference cost. The concept is simple, each time the algorithm encounters an unnamed branch of the system it caches the factors obtained after its elimination and reuses it each time an equivalent branch is found. This is possible solely due to the generation algorithm of unnamed objects that must guarantee the equivalence among branches spanning from objects of the same class. As we will see later, we will not concern ourselves with open world systems, we will not exploit this feature of SVE.

Probabilistic Relational Models as an object-oriented framework

It can be surprising to consider PRMs as an extension of OOBNs *à la Pfeffer* because they are not described as an object-oriented PGM. It is even more surprising to see that notions such as class inheritance, abstraction and subtype polymorphism are not extended in PRMs. However, the differences between OOBNs *à la Pfeffer* and PRMs are scarce. An important change is the simplification of the framework, making it much more accessible, but the most important change is the introduction of reference slots. In OOBNs *à la Pfeffer*, classes were connected using input attributes, thus it was necessary to declare as output nodes each node that had children in other classes. In PRMs this constraint disappears since a class can reach the totality of a class' attributes using reference slots. From a modeling perspective, such representation is more flexible and drastically simplifies the creation of complex systems. A side effect is to reduce the structure in PRMs, indeed objects are no longer encapsulated in one another and exploiting hierarchical inference is harder. The fact that PRMs are not labeled by their creators as an object-oriented PGM does not prevent us from considering it as one. The purpose of chapter 4 and one contribution of this thesis, is to complete the PRM framework to make it a fully object-oriented PGM that is well suited for modeling complex systems.

3.4 First-Order Probabilistic Models

First-Order Probabilistic Models (FOPMs) represent relations among BN fragments using first-order logic. The use of first-order logic helps defining complex relationships and considerably increases PGMs expressive power. FOPMs are undeniably at the core of most recent contributions and regroup many extensions. Among them we can find Markov logic networks (Domingos et al., 2006), Bayesian Logic Programs (Kersting and Raedt., 2001), Multi Entity Bayesian Networks (Laskey, 2008), Relational Bayesian Networks (Jaeger, 1997), parfactors (Poole, 2003) and most frameworks presented in (Getoor and Taskar, 2007). At first sight, we could think that FOPMs do share much in common with object-oriented PGMs, but we will see that it is not the case. Indeed, Multi-Entity Bayesian Network (MEBN), the first FOPM we present was at first thought as a BN object-oriented extension (Mahoney and Laskey, 1996). But is has evolved into FOPMs after many years of refinement. The other framework that we will present, called parfactors, is at the core of new inference techniques called first-order inference. Although our domain does no require such inference technique, it is not incompatible with object-oriented PGMs (see chapter 4).

Multi-Entity Bayesian Networks

MEBNs are presented as a first-order language for specifying probabilistic knowledge base using parametrized BN fragments (Mahoney and Laskey, 1996, 1997, 1998a,b; Laskey, 2008). MEBNs have then been developed for different applications, mostly military (Laskey et al., 2001, 2004; da Costa et al., 2005; da Costa and Laskey, 2006). Finally, the framework is fully formalized in the following publications (da Costa and Laskey, 2005a,b; Laskey, 2008).

The substitute of a class in the MEBN framework is called a *MFrag* and can be described as a BN fragment with several enhancements. As for OOBNs, variables in a *MFrag* are divided in three categories: resident random variables, i.e., classic variables in BN, input random variables, similar to reference nodes in OOBNs *à la Bangsø & Wuillemin* and context variables, variables defining constraints over relations using first-order predicates. These last variables share the same purpose of reference slots in PRMs, i.e., they define the relations among *MFrags*.

Definition 3.39 *An* MFrag $\mathcal{F} = (\mathcal{C}, \mathcal{I}, \mathcal{R}, \mathcal{G}, \mathcal{D})$ *consists of a finite set* $\mathcal{C}$ *of context value assignment terms; a finite set* $\mathcal{I}$ *of input random variable terms; a finite set* $\mathcal{R}$ *of resident random variable terms; a fragment graph* $\mathcal{G}$*; and a set* $\mathcal{D}$ *of local distributions, one for each member of* $\mathcal{R}$*. The sets* $\mathcal{C}$*,* $\mathcal{I}$ *and* $\mathcal{R}$ *are pairwise disjoint. The fragment graph* $\mathcal{G}$ *is an acyclic directed graph whose nodes are in one-to-one correspondence with the random variables in* $\mathcal{I} \cup \mathcal{R}$*, such that random variables in* $\mathcal{I}$ *correspond to root nodes in* $\mathcal{G}$*. Local distributions specify conditional probability distributions for the resident random variables [...].*

Figure **??** illustrates two *MFrags* from the *starships and Klingons* example (da Costa and Laskey, 2005b). Dashed nodes are context nodes, light gray nodes are input nodes

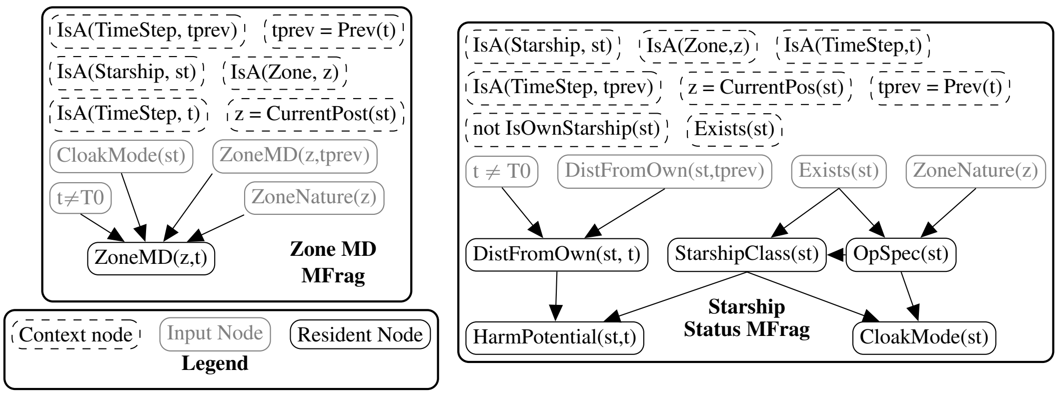


Figure 3.16: Two MFrags from the *starships and Klingons* MTheory (da Costa and Laskey, 2005b).

and plain nodes are resident nodes. We can see that context variables are first-order predicates over logical variables. Logical variables can be seen as instantiations of *MFrags* and define relations among them. For example, in the *MFrag* **Zone MD** the four *IsA* predicates enforce types over variables *tprev*, *t*, *st* and *z*. The remaining two context nodes constrain *tprev* as being the previous time step of *t* and force *z* to be the current position of *st* (representing a starship). Sometimes a resident variable can have several parents dependent of the context. This requires multiple conditional probability distributions for each given set of parents. To avoid enumerating all situations, different solutions are proposed. In da Costa and Laskey (2005b) a script language defines CPTs on the fly. In Laskey (2008) a generic approach using influence counts and combination functions is proposed. Unfortunately, these solutions lack any practical use and prevent a standard implementation of MEBNs. However, existing applications based on MEBNs suggest that practical solutions have been found to express probabilities in this framework. Unfortunately, they are either highly prototypical (no such features) or restricted to military use (not publicly available).

Probabilistic inference in MEBNs is done by generating a situation specific BN (Mahoney and Laskey, 1998a). Which is equivalent to ground inference with sensitivity analysis to prevent unnecessary groundings. In some cases the ground network can be infinite, it is then necessary to approximate it (Laskey, 2008).

Parfactors

Adding parameters to BNs has been the topic of several publications (Horsch and Poole, 1990; Buntime, 1990; Jaeger, 1997). However, parfactors are remarkable as they have been introduced for the sole purpose of first-order inference (Poole, 2003). They also offer a generic and concise formalization of First-Order PGMs. Parfactors enabled first-order inference in several frameworks such as Markov Logic Networks or Bayesian Logic Programs (Domingos and Richardson, 2007; Milch et al., 2005). A parfactor is a parameterized factor and the parameters are logical variables on which constraints are defined. They usually represent undirected PGMs, however they can represent directed models if each parfactor defines a CPT. Parfactors are well suited in representing large population of individuals for which we have very little information, i.e., how they are related to one another. The following definition is taken from Poole (2003).

Definition 3.40 (Parfactors) *A parfactor is a triple $\langle C, V, t \rangle$ where C is a set of constraints on parameters, V is a set of parametrized random variables and t is a table representing a factor from the random variables to the non-negative reals.*

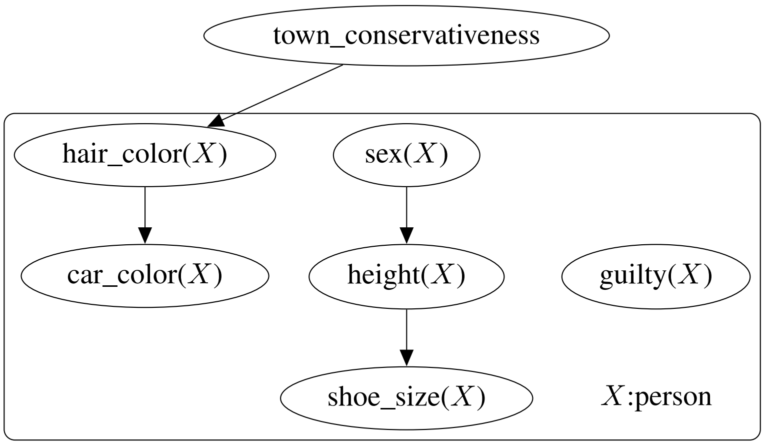


Figure 3.17: A parameterized BN from Poole (2003). Each node can be represented using a parfactor.

Note that the logical variable X can be interpreted as a class and each parameterized random variable with constraints over X as an attribute of that class. Where the example of figure 3.17 is simple (it has no constraints), parfactors can easily represent complex relationships among random variables with the use of first-order logic. However this *low-level* representation makes parfactors a difficult framework for modeling purposes. Usually, richer languages are used to model a PGM and then the network is transformed into a parfactor representation for inference purpose (Milch et al., 2005). Constraints over parameters are used to define the relations among parfactors and it can be any valid first-order proposition. Figure 3.17 illustrates a parameterized BN for the robbery example (Poole, 2003). Each node can be defined using parfactors, for example $\langle$ $\{\}$, $\{hair\_color(X), conservativeness\}$, $t\rangle$ is a valid parfactor for the parameterized node *hair_color(X)*. With t defined as follows:

hair_color	*conservativeness*	Val
purple	conservative	0.001
purple	liberal	0.01
blue	conservative	0.1
blue	liberal	0.05
	...	

Regarding inference, parfactors inference scheme consists in grounding the network into a MN and applying classic inference algorithms (variable elimination or junction trees techniques). However, parfactors purpose is to enable first-order inference, which has been a long sought objective of the PGM community (Halpern, 1990; Jaeger, 1999; Pasula and Russell, 2001; Poole, 2003). First-order inference exploits the lack of information about objects to lift elimination of worlds seemingly identical. For example, suppose we have a population of five individuals and that we only gave information about a given John Doe and that the two worlds $\{$ *friends(X_1, John Doe)*, *friends(X_1, X_2)*, *friends(X_3, X_4)* $\}$ and $\{$ *friends(X_2, John Doe)*, *friends(X_1, X_2)*, *friends(X_3, X_4)* $\}$ both induce the exact same probability distributions for *guilty(John Doe)*. First-order inference prevents inferring over both of these words by *lifting* the parameterized variables and by eliminating groups of equivalent worlds in one pass. From there, several enhancements have been proposed either to reinforce the lifting process or to extend first-order inference to handle specific graphical structures (de Salvo Braz et al., 2005; de Salvo Braz, 2007; Milch et al., 2008; Kisynski and Poole, 2009).

3.5 Discussion

About representing relations in PGMs. The idea of representing relations is a fundamental challenge for computer sciences, especially in AI. The existence of knowledge representation paradigms such as entity-relationship, object-oriented, first-order or frames prove the complexity of such a task. For the PGM community the necessity to deal with different paradigms appeared with the notion of BN fragments. Once the idea of fragmenting a BN emerged, connecting the fragments have been the focus of many new frameworks. The goal of this chapter was to give a snapshot of this trend and show how, in almost twenty years, BN have slowly evolved into several sophisticated frameworks dedicated to this task. The purpose of this discussion is to place this thesis' contributions within that trend. The main contributions of this thesis concern specifying an object-oriented framework for PGMs and developing inference algorithms exploiting structural information provided by the object-oriented paradigm. To understand why we chose to reinforce the object-oriented paradigm instead of focusing on first-order models, it is important to understand how BN extensions have evolved. MSBNs and DBNs were the first models where the notions of interfaces, structural repetitions and fragmentations have appeared. Following those frameworks, OOBNs were proposed using input and output nodes to describe relations between fragments. Relational models completed the work by borrowing the notion of slots from plate models and by focusing the description over the existence of relations rather than the existence of probabilistic links. Finally, first-order models put the focus on describing relations using first-order predicates and almost ignoring the conditional probability distribution specification: MEBNs offer high-level tools laking any practical use and parfactors require to specify all parents as parameters: the node and all of its parents must be explicitly defined. This remark is generalizable to most first-order probabilistic models since the probability distributions are specified using classic tabular representations. In some cases log-probabilities are even used. All these frameworks lack modeling aspects, which is undoubtedly because they are mostly used for machine learning tasks or for academic purposes.

Being practical about the object-oriented paradigm in PGMs. We have seen that there exists an *official* object-oriented framework (OOBNs), but many frameworks that are not qualified as object-oriented still implement many object-oriented features. DBNs, MSBNs, PRMs and MEBNs are example of frameworks that exhibit an object-oriented aspect: they all use BN fragments at the core of their formalization. The fact that each of these fragments is a class and that they define an interface is, for most frameworks, ignored (purposely or not). If the object-oriented aspect of these frameworks is hidden, it is a consequence of the unpopularity of OOBNs among the PGMs community. OOBNs have been quickly replaced by PRMs by Pfeffer and Koller and have only known few contributions from Bangsø *et al.*. We can only make guesses on the reasons why OOBNs have been dropped: not because they are a failure but because adding object-oriented features to PGMs may be perceived as a software feature more than a fundamental aspect of the theory. Another cause would be the community's interest for FOPMs and first-order inference: the number of contributions in these two domains speaks for itself. Or maybe that academic use of PGMs do not require object-oriented frameworks for machine learning or probabilistic inference. If we look at (Koller and Friedman, 2009) object-oriented frameworks (DBNs, PRMs) are described as *template-based models*, i.e., macro languages enabling richer representations and more complex queries. The point is that PGMs lack a properly defined object-oriented framework as many fundamental features of the object-oriented paradigm are missing or incomplete, mainly inheritance, abstraction and polymorphism. Moreover, many features of *template-based models* can be described or extended by considering them under the scope of the object-oriented paradigm (see chapter 4.1). An important result of this thesis is to prove that theoretical research is not superfluous, neither for the framework formalization nor for inference algorithms.

FOPMs and the object-oriented paradigm. Interestingly, most contributions concerning PRMs exhibit what is considered as their major feature: structural uncertainty. In fact, by introducing such feature PRMs opened the way to FOPMs as it is their key feature to represent complex relationships between random variables. In that matter, parfactors reduce structural uncertainty to its minimalistic form, i.e., a set of first-order predicates over parameterized random variables. On the other hand, MEBNs are a rich and complex FOPM which connects BN fragments (MFrags) using context nodes, i.e., first-order predicates. Most importantly and against intuition, the object-oriented paradigm is not incompatible with first-order representation of relations, MEBNs are a perfect example since BN fragments are a simplistic application of the object-oriented paradigm. This confusion is at the core of many misunderstandings concerning the work presented in this thesis and object-oriented extensions of PGMs. PRMs are seen as extensions of OOBNs and FOPMs as extensions of PRMs, thus discarding the object-oriented paradigm as an improper solution to represent complex relations between random variables. It is untrue since the object-oriented paradigm helps modeling complex systems and not complex relations between objects. In chapter 4.1 we show how our enhanced PRMs can be represented using parfactors, discarding the assumption that both representations are incompatible.

The need for a formal object-oriented PGM. At the start of this thesis our goal was to create a new tool for modeling large scale and complex networks using PGMs. The SKOOB ANR project was created for this purpose and is at the core of most contributions of this thesis. Most importantly, it enabled us to conclude with the help of experts that there were missing features in the current theories. We chose to build a new object-oriented framework from an existing one: PRMs. At the current time, that framework was (and is still if we put aside the work presented here) the only one including a well thought object-oriented aspect. PRMs are also a direct extension of OOBNs *à la Pfeffer* and thus has legitimacy as a starting point to enhance its object-oriented features. From there we created the SKOOL language, a declarative and object-oriented language for representing object-oriented PRMs. We also enhanced PRMs with object-oriented features to formally define class inheritance, typing, type inheritance, interface implementations. The SKOOL language is presented in Appendix A and the enhancement of PRMs in chapter 4.1.

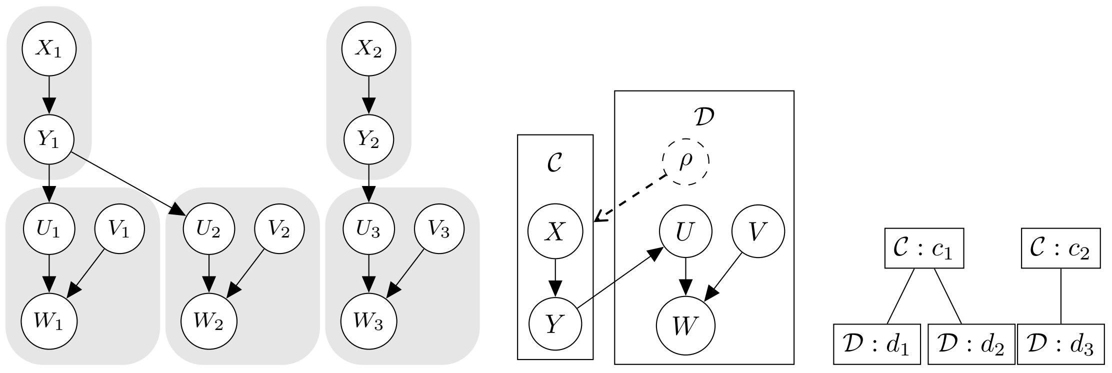
Chapter 4

Reinforcing Probabilistic Relational Model's Object-Oriented Aspect

PGMs are a general purpose framework for dealing with uncertainty. Their applications to many different domains stimulated an uninterrupted process of creation of new frameworks based on probability theory. In recent years, the Statistical Learning community proposed new probabilistic frameworks, closing the gap between first-order logic and probability theory. New models such as OOBNs, MSBNs, PRMs and MEBNs have extended BNs and widen their range of applications. In many situations, these new FOPMs can be efficiently learned from databases and used for answering probabilistic queries. However, there are situations where the scarcity of available data prevents learning. For such problems, oriented graphical models such as PRMs are more suitable than the aforementioned FOPMs, because they can be modeled by experts.

Despite being an OOBN extension, PRMs lack fundamental object-oriented features: class inheritance, subtype polymorphisms, abstraction... In software engineering, such object-oriented designs are useful for creating complex software. In this chapter, we illustrate why these mechanisms are necessary for practical design of large-scale systems and we show how light extensions can enforce strong object-oriented features in PRMs. Furthermore, we propose a representation of our extended version of PRMs using parfactors, the state-of-the-art framework for first-order probabilistic inference.

This chapter is organized as follows: after redefining the major features of *classical* PRMs, we introduce the notion of attribute's type inheritance and extend attribute overloading. We then propose a new definition for class inheritance based on attributes and reference slots overloading. Then, we offer a solution to multiple inheritance in PRMs by introducing interfaces. Finally, we provide an algorithm to transform PRMs into parfactors.



(a) A BN. (b) Two classes $\mathcal{C}$ and $\mathcal{D}$. (c) An instance diagram.

Figure 4.1: Analysis of the BN (a) reveals the use of two recurrent patterns, which are confined in two classes (b). Hence, a system equivalent to figure 4.1a may be built (c).

4.1 Probabilistic Relational Models

We give here a more formal definition of PRMs than the one given in chapter 3.3. Our objective is to clarify this framework under the scope of modeling purposes. Figure 4.1a shows a BN encoding relations between two different kinds of patterns: random variables X_i, Y_i on one hand and U_j, V_j, W_j on the other hand. We assume that the CPTs associated with random variables with the same capital names are identical. When using PRMs, the main idea is to abstract each pattern as a generic entity, called a *class*, which encapsulates all the relations between variables in the same pattern. Random variables encapsulated by a class are called *attributes*. So, in figure 4.1b, $\mathcal{C}$ encapsulates variables X_i and Y_i as well as their probabilistic relations (arc (X_i, Y_i)) and $\mathcal{D}$ encapsulates random variables U_i, V_i and W_i. Note that in figure 4.1a there exist links between different patterns, such as links $Y_1 \rightarrow U_1$ or $Y_2 \rightarrow U_3$. Classes only access to their own elements, in opposition to nodes in BNs that can be linked to other nodes in the network. To make an analogy with computer programming, we can say that BNs nodes are global variables and classes attributes are local variables, i.e., visible only in the scope of their encapsulating class. Consequently, dependencies among classes cannot be represented using BNs terminology, i.e., we cannot write $Y \notin \pi(U)$. Hence classes must have a mechanism allowing to refer to random variables outside the class. In PRMs, this mechanism is called a *reference slot*. Basically, the idea is to create some pointer ρ allowing a class to extend its scope to attributes in other classes. In figure 4.1b, reference slot ρ gives $\mathcal{D}$ access to $\mathcal{C}$'s attributes, e.g., U parent is $\rho.Y$. Reference slots can be used sequentially to create *paths*, called *slot chains*. Now, as shown in figure 4.1c, the BN from figure 4.1a can be built up from the PRM: it is sufficient to create two instances, c_1 and c_2, of class $\mathcal{C}$ as well as three instances d_1, d_2, d_3 of $\mathcal{D}$ and associate at least one instance to each reference slot of each instance. Connecting instances through reference slots can be represented using instance diagrams, as illustrated in figure 4.1c.

Classes, reference slots and attributes

We will provide a concise set of definitions that offer a formalization equivalent to those presented in chapter 3.3. The most important elements in a PRM are classes.

Definition 4.1 (Class) *A* class $\mathcal{C}$ *is defined by a Directed Acyclic Graph (DAG) over a set of attributes* $\mathcal{A}(\mathcal{C})$ *(see definition 4.5) and a set of reference slots* $\mathcal{R}(\mathcal{C})$ *(see definition 4.2). To refer to a given attribute* X *(resp. reference* ρ*) of class* $\mathcal{C}$*, we use the standard object-oriented notation* $\mathcal{C}.X$ *(resp.* $\mathcal{C}.\rho$*).*

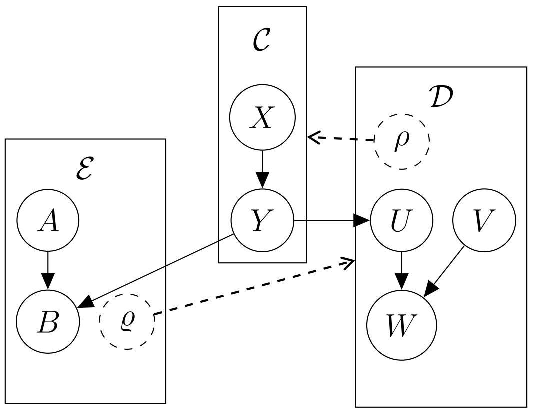
Classes are an elaborate version of BN fragments, their principal enhancement is the concept of reference slots. Reference slots indicate how classes are related to each other, not probabilistically but conceptually. This adds another dimension to modeling with PGMs: groups of random variables are clustered together in classes and these clusters are related to each other using reference slots.

Definition 4.2 (Reference slots) *Let* $\mathcal{C}$ *and* $\mathcal{D}$ *be two classes. A reference slot* $\mathcal{C}.\rho = \mathcal{D}$ *is a pointer towards a class* $\mathcal{D}$ *such that we can access elements of* $\mathcal{D}$ *through* $\mathcal{C}$*, e.g.,* $\mathcal{A}(\mathcal{C}.\rho) = \mathcal{A}(\mathcal{D})$ *and* $\mathcal{R}(\mathcal{C}.\rho) = \mathcal{R}(\mathcal{D})$*. We say that* $\mathcal{C}$ *is the domain of* $\mathcal{C}.\rho$*, denoted* $domain(\mathcal{C}.\rho)$*, and* $\mathcal{D}$ *is the range of* $\mathcal{C}.\rho$*, denoted* $range(\mathcal{C}.\rho)$*. A reference slot* $\mathcal{C}.\rho$ *is simple if it defines a 1-to-1 relation and is complex if it defines a 1-to-n relation.*

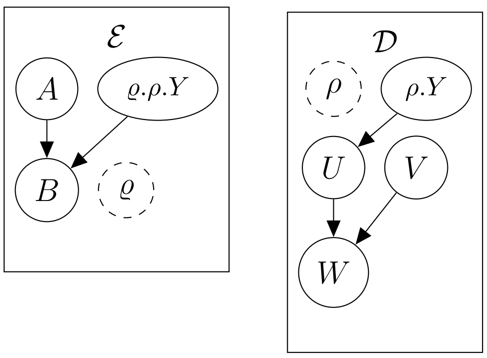
If there is no ambiguity about a reference slot's domain, we may write ρ instead of $\mathcal{C}.\rho$. The previous definition is a direct analogy with object-oriented programming languages, where objects are accessed through pointers. Traditionally, PRMs see reference slots through the lens of the entity-relationship paradigm. Both views are compatible, but an object-oriented approach is preferable since we are reinforcing the object-oriented aspect of PRMs. In figure 4.1 we can see that $\mathcal{D}.\rho$ is a reference slot pointing on class $\mathcal{C}$ and that such reference slot is used to define a probabilistic dependency between $\mathcal{D}.U$ and $\mathcal{C}.Y$. Such use of a reference slot is called a slot chain.

Definition 4.3 (Slot chains) *A* slot chain $\mathcal{C}.\mathbf{K}$ *is a sequence* $\{\rho_1, \cdots, \rho_n\}$ *of reference slots such that* $domain(\rho_1) = \mathcal{C}$ *and* $range(\rho_i) = domain(\rho_{i+1})$ *for* $1 \leqslant i < n - 1$*. We denote by* $range(\mathcal{C}.\mathbf{K}) = range(\rho_n)$ *its range and by* $domain(\mathcal{C}.\mathbf{K}) = domain(\rho_1)$ *its domain.*

As for reference slots, we may write $\mathbf{K}$ instead of $\mathcal{C}.\mathbf{K}$ when there is no ambiguity about $\mathbf{K}$'s domain. Slot chains are the practical use of reference slots to connect attributes from different classes together. Indeed, since a reference slot gives access to elements in a class, we can chain them to connect classes not directly related. Consequently, slot chains are commonly used when defining attributes dependencies. For example, in figure 4.2a attribute $\mathcal{D}.U$ is a child of attribute $\mathcal{C}.Y$, both of different classes. In such case, $\mathcal{C}.Y$ is accessed using a slot chain of one reference slot ($\mathcal{D}.\rho$) and we say that $\mathcal{D}.\rho.Y$ is the parent of $\mathcal{D}.U$. In figure 4.2a there is another slot chain connecting $\mathcal{E}.B$ to $\mathcal{C}.Y$: $\mathcal{E}.\varrho.\rho$. In the classic PRMs formalism, there is no graphical representation of slot chains and they can only be deduced by existing dependencies between attributes. Such representation is not well suited when modeling systems with many classes and complex probabilistic dependencies. A possible solution is illustrated in figure 4.2b where attributes accessed through slot chains are represented using an node labeled by the path leading to the desired attribute.



(a) Three classes $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{E}$ with a slot chain connecting $\mathcal{E}.B$ with $\mathcal{C}.Y$.



(b) An alternative representation of the slot chains in figure 4.2a.

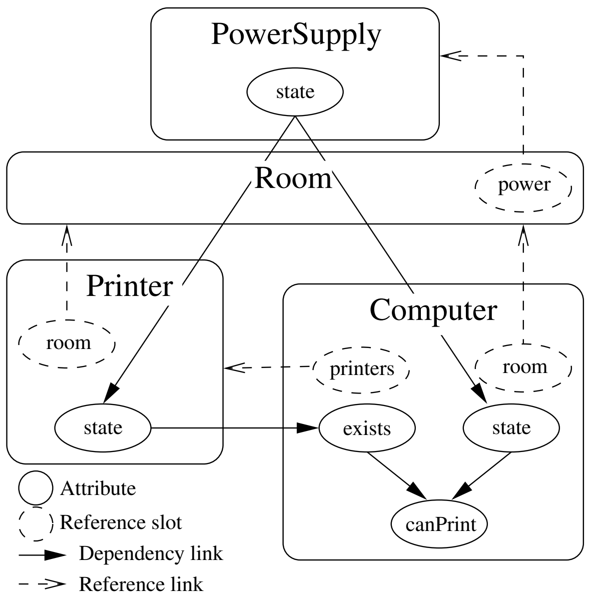
Figure 4.2: Reference slots and slot chains are used to define probabilistic dependencies among classes. Dashed nodes are reference slots, dashed links represent reference slot's range, solid line nodes represent attributes and solid line arcs probabilistic dependencies between two attributes.

Definition 4.4 (Attribute's type) *An attribute's type τ describes a family of distinct discrete random variables sharing the same domain $\tau = \{l_1, \cdots, l_n\}$, where n is the domain size of τ.*

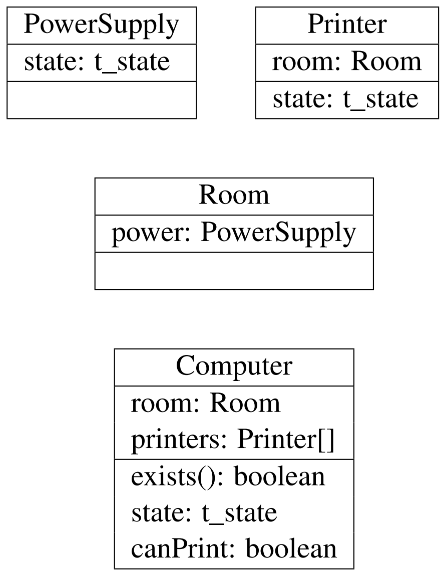
The notion of Attribute's types was introduced in OOBNs *à la Pfeffer*, however we will use it differently. Indeed, types in OOBNs *à la Pfeffer* are inferred once an object is defined. Here, types are defined before being used. In complex systems there are often many random variables sharing the same domain, thus by defining once an attribute's type we reduce the amount of redundant information that must be specified when modeling the system. Furthermore, we will see that attribute's types are a fundamental aspect of class inheritance (see section 4.2).

Definition 4.5 (Attributes) *Let $\mathcal{C}$ be a class. An attribute $\mathcal{C}.X$ is a triplet $\langle \tau_{\mathcal{C}.X}, \pi(\mathcal{C}.X), \phi_{\mathcal{C}.X} \rangle$, where $\tau_{\mathcal{C}.X}$ is $\mathcal{C}.X$'s type, $\pi(\mathcal{C}.X)$ a set of attributes called the parents of $\mathcal{C}.X$ and $\phi_{\mathcal{C}.X}$ a factor encoding the conditional probability distribution $P(\mathcal{C}.X|\pi(\mathcal{C}.X))$.*

Again, we will write X instead of $\mathcal{C}.X$ when there is no ambiguity about X's class. Attributes are equivalent to nodes in BNs. However, at class level, attributes do not define random variables, but a more generic pattern from which many *identical* random



(a) The class dependency diagram of the power surge example.



(b) UML diagrams offer a more compact representation of classes.

Figure 4.3: The power surge example. Theses classes are meant to model systems in which several computers and printers are in different rooms.

variables are created. They are closely related to the notion of parametric random variables (see section 4.3). Defining a factor over attributes is an abuse of notation, since factors are defined over random variables. However, since attributes are patterns for random variables, the link between attributes and the random variables they represent is sufficiently obvious to define unambiguously factors. Classes are meant to be used as generic entities, instantiated as many time they are used in a system. It is only after being instantiated that classes attributes becomes random variables.

Figure 4.3 illustrates the power surge example. We can see that four classes are modeled: *Power Supply*, *Room*, *Printer* and *Computer*. Except for the *Power Supply* class, each class has at least one reference slot (dashed nodes): a room is connected to a power supply, computers and printers are in a room from which they have access to a power supply. Attributes (nodes in solid lines) are present in all classes except class *Room* which is only a relational class, i.e., its purpose is to define a relation among class *Power supply* on one hand and among classes *Printer* and *Computer* on the other. Attribute *Computer.state* (respectively *Printer.state*) is connected to attribute *PowerSupply.state* through the slot chain *Computer.room.power.state* (respectively *Printer.room.power.state*) and attribute *Computer.exists* is connected to attribute *Printer.state* trough *Computer.printers.state*. Finally, note that in figure 4.3b attribute *Computer.exists* and reference slot *Computer.printers* declarations differ from other attributes. Simply because reference slot *printer* is complex and attribute *exists* is an aggregator (see section 4.1).

Instances, relational skeletons and PRMs

Definition 4.6 (Instance) *An* instance c *of class* $\mathcal{C}$ *is a BN fragment whose attributes are classic BN nodes generated from their class level counterparts and where reference slots refer to sets of their range's instances. Thus, instantiated slot chains can be used to access instances attributes parents (if necessary). As for classes, we use an object-oriented notation to access instances elements and use the same set of notations available for classes, i.e.,* $\mathcal{A}(c)$ *refers to c's attributes and* $\mathcal{R}(c)$ *refers to c's reference slots.*

It is important to differentiate attributes at class level from attributes at instance level, even if they are tightly related there is an important difference: class level attributes are not random variables and instance level attributes are random variables. Furthermore, the family of attributes spawned from the same class are all distinct random variables. Instances in a system are connected together to form a relational skeleton. Given an instance i and a reference slot $\rho \in \mathcal{R}(i)$, we denote by $range(i.\rho)$ the set of instances connected to i through ρ.

Definition 4.7 (Relational Skeleton) *A relational Skeleton* $\mathcal{S}$ *is a set of instances such that for any instance* i *of class* $\mathcal{C}$ *and any reference slot* $\mathcal{C}.\rho = \mathcal{E}$*, there exists at least one instance* $j \in \mathcal{S}$ *such that* j *is an instance of* $\mathcal{E}$ *and* $j \in range(i.\rho)$*. Finally, we denote by* $\mathbb{I}_{\mathcal{S}}(\mathcal{C})$ *the set of instances of class* $\mathcal{C}$ *in* $\mathcal{S}$*.*

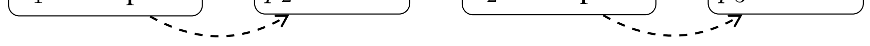


Figure 4.4: A relational skeleton using instances from the power surge example.

Relational skeletons enforce the instantiation of all reference slots, consequently any relational skeleton models a valid probability distribution. We will usually refer to the relational skeleton as the system, since it is the closest representation of a system we can have using PRMs. The graphical representation of a relational skeleton is called an instance diagram. It is a directed graph in which each node is an instance and edges indicate that two instances are connected by a reference slot. We will often say that a reference slot is instantiated, i.e., it is linked to one or more instances of the correct class in a given system.

Definition 4.8 (PRM) *A PRM Π is defined by a set of classes $\mathscr{C}$ and a relational skeleton $\mathcal{S}$ and factorizes the following joint probability distribution:*

$$P(\mathcal{A}(\mathcal{S})) = \prod_{\mathcal{C}\in\mathscr{C}} \prod_{c\in\mathbb{I}_{\mathcal{S}}(\mathcal{C}))} \prod_{c.X\in\mathcal{A}(c)} P(c.X|\pi(c.X)),$$

where $\mathcal{A}(\mathcal{S})$ is the set of all attributes in the relational skeleton $\mathcal{S}$.

Figure 4.4 illustrates a relational skeleton with instances from the power surge example. Dashed arcs represent reference slots assignments, i.e., $c \dashrightarrow d$ means c has a reference slot referencing d. In the power surge example, reference slot assignments are explicit, but for systems more complex it would be necessary to label the arcs to prevent ambiguities. For example, we could label the arc $c_1 \dashrightarrow r_1$ with $room$ and the arcs $c_1 \dashrightarrow p_1$ with $printers$.

Inverse reference slots, aggregates and output attributes

We will now define several useful notions, either for modeling or for inference. The first of them combines two definitions: inverse reference slots and inverse slot chains.

Definition 4.9 (Inverse reference slot) *Let two classes $\mathcal{C}$, $\mathcal{D}$ and a reference slot $\mathcal{C}.\rho \in \mathcal{R}(\mathcal{C})$ with $range(\mathcal{C}.\rho) = \mathcal{D}$. The inverse of $\mathcal{C}.\rho$, denoted $\mathcal{D}.\rho^{-1}$ is a reference slot such that $range(\mathcal{D}.\rho^{-1}) = domain(\mathcal{C}.\rho)$ and $domain(\mathcal{D}.\rho^{-1}) = range(\mathcal{C}.\rho)$.*

Obviously, if we can define an inverse reference slot, we can define an inverse slot chain.

Definition 4.10 (Inverse slot chains) *Let $\mathcal{C}.\mathbf{K} = \{\rho_1, \cdots, \rho_n\}$ be a slot chain. Its inverse is the slot chain $\mathcal{D}.\mathbf{K}^{-1} = \{\rho_n^{-1}, \cdots, \rho_1^{-1}\}$ where $range(\mathcal{C}.\mathbf{K}) = \mathcal{D}$.*

Inverse slot chains are useful to retrieve attributes children defined in other classes. For example, in figure 4.3 attribute *PowerSupply.state* has two children *Printer.state* and *Computer.state* that can be accesed using the inverse slot chains of $Printer.room.power$ (denoted $\mathbf{K}$) and $Computer.room.power$ (denoted $\mathbf{L}$): $PowerSupply.\mathbf{K}^{-1}.state$ and $PowerSupply.\mathbf{L}^{-1}.state$. Similarly, the inverse slot chains of a given instance i give access to the set of instances that have dependencies on i. For example, in figure 4.4 the instances in sets $p.\mathbf{K}^1 = \{p_1, p_2, p_3\}$ and $p.\mathbf{L}^{-1} = \{c_1, c_2\}$ are dependent on one of p's attributes ($p.state$). We will now re-introduce a notion that existed in OOBN but that was dropped in the classic PRM formalization: output attributes.

Definition 4.11 (Output attributes) *An output attribute $\mathcal{C}.X$ is an attribute such that there exists some attribute $\mathcal{D}.Y$ that is a child of X accessing $\mathcal{C}.X$ through a slot chain.*

We cannot define output attributes as attributes with children outside of their class, since such definition would discard recursive classes. Indeed, for such classes some attributes would access to attributes of their classes using a recursive reference slot and we would have $\mathcal{D} = \mathcal{C}$. Consequently, we must define an output attribute as an attribute being accessed using a slot chain.

The final notion concerns aggregators. We have seen that reference slots are sets of instances when instantiated. In most cases, reference slots model unary relations, i.e., an instance will be connected to a single instance. But in some cases, an instance will be connected to many instances. CPTs are not a reasonable choice to encode conditional probability distributions for attributes with a variable number of parents: since we want to keep the probabilistic semantics declared at class level, we would have to declare CPTs for each possible number of parents. Obviously, such solution suffers from the possible large number of different configurations. A solution proposed in the classic PRM framework is to aggregate parents using a special arc called an aggregator. The concept comes from database theory, where several operators have been defined to handle large amounts of data. In practice we have noticed that such arcs are often modeled as attributes with specific conditional probability distributions, usually deterministic functions. This leads to the following definition of aggregators that is different from the one proposed in Getoor et al. (2007).

Definition 4.12 (Aggregators) *Aggregators are attributes with a conditional probability distribution defined by a set of rules that can be used to generate the attribute's CPT once it is instantiated and connected to its parents.*

Classic aggregators are *min*, *max*, *for all*, *exist* and *k-gates*. They often enable to implement optimized data-structures preventing high memory consumptions. See appendix A for the specification of the aggregators we implemented in the SKOOL language.

Definition 4.13 (Ground Baysian Network) *A ground Bayesian Network is a BN $\mathcal{B}$ constructed from a PRM $\Pi = (\mathscr{C}, \mathcal{S})$ using the following steps (Getoor et al. (2007)):*

1. *There is a node for every attribute $i.X$ of every instance $i \in \mathcal{S}$, named $i.X$.*
2. *Each $i.X$ depends probabilistically on parents of the form $i.Y$ or $j.Y$ such that there exists a slot chain $\mathbf{K}$ with $j \in range(i.\mathbf{K})$.*
3. *The conditional probability distribution for $i.X$ is a CPT generated from $\phi_{C.X}$, where C is the i's class.*

Ground BNs purpose is twofold. They provide formal justification to PRMs and enable classic BN inference algorithms. The second point has already been discussed in chapter 3: ground BNs is the most direct option to perform probabilistic inference in PRMs. Regarding the first point, ground BNs provide justification because we can see PRMs as some high-level macro language to model BNs. Even if in practice they help modeling systems too complex to model using classic BNs, the fact that any PRM is a high level representation of a BN proves the soundness of this framework. Moreover, the underlying BN of a PRM is relatively obvious, as PRMs offer an intuitive framework for anyone familiar with BNs.

4.2 Class inheritance

In section 3.2 we have seen two different definitions for class inheritance in OOBNs. In OOBNs *à la Pfeffer*, the set of input nodes of a subclass must be a subset of the superclass input nodes set. For OOBNs *à la Bangsø & Wuillemin*, new input nodes can be added with specialization but none can be removed. Both solutions are restrictive (see the discussion in section 3.2). We will see that slot chains help defining a flexible class inheritance mechanism, but we must first understand why slot chains make PRMs different from OOBNs.

In section 4.1, we have seen that there only exists one kind of attribute, i.e., attributes are not declared as input, encapsulated or output attributes. In OOBNs, the user must declare what attributes in a class can be accessed by other classes. In PRMs an attribute becomes an output attribute when the user adds a child from another class to that attribute. OOBN's input attributes are replaced in PRMs by slot chains. When the user adds an attribute to a class, he must declare the attribute's parents. If some of them are defined in another class, he simply accesses them using a slot chain (supposing that it is possible to defined such a slot chain). To make an analogy with programming languages, declaring input attributes is similar to declaring variables at the beginning of a function (like in C) in opposition to declaring them where they are used (like in C++). In OOBNs, we must *push* out of a class its output attribute, i.e., we must know when modeling the class which attributes will be accessed by other classes. In PRMs, classes *pull* from other classes any required attribute, i.e., we delay the dependencies definitions when we are modeling them, not when we are modeling the class that will be used by other classes. Discussions with experts in the ANR project SKOOB showed us that the second solution was often simpler and helped experts in the modeling process of complex systems.

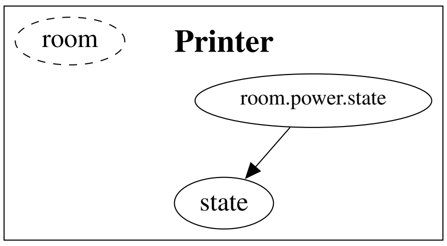
Slot chains and reference slots detail explicitly how probabilistic dependencies can be defined between classes, thus it is easier to define precisely how such feature is inherited. Indeed, OOBNs limited relational mechanisms prevent from defining a complete class inheritance mechanism and we will show in this chapter how PRMs can be used to define a complete class inheritance scheme. We will start by adapting OOBN's class inheritance scheme to PRMs and we will complete this definition throughout this section.

Definition 4.14 (Simple class inheritance) *A class $\mathcal{D}$ is a subclass of class $\mathcal{C}$, denoted $\mathcal{C} \rhd \mathcal{D}$, if:*

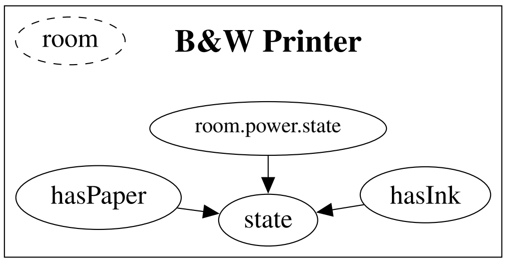
- *for each attribute $\mathcal{C}.X = \langle \tau_{\mathcal{C}.X}, \pi(\mathcal{C}.X), \phi_{\mathcal{C}.X} \rangle$, there exists an attribute $\mathcal{D}.X = \langle \tau_{\mathcal{D}.X}, \pi(\mathcal{D}.X), \phi_{\mathcal{D}.X} \rangle \in \mathcal{A}(\mathcal{D})$ such that $\tau_{\mathcal{C}.X} = \tau_{\mathcal{D}.X}$;*

- *for each reference slot $\mathcal{C}.\rho = \mathcal{E}$, there exists a reference slot $\mathcal{D}.\rho = \mathcal{F}$ such that $\mathcal{E} = \mathcal{F}$.*

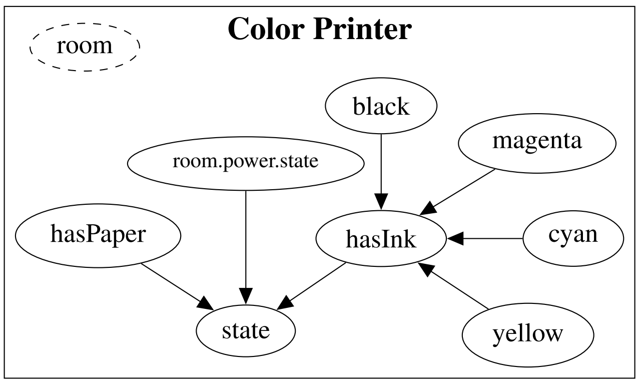
Definition 4.14 is a simple definition of class inheritance and we will improve it with each new notion that we will add to PRMs. Figure 4.5 illustrates three classes: *Printer* (figure 4.5a), *B&W Printer* (figure 4.5b) and *Color Printer* (figure 4.5c). *Color Printer* is a subclass of *B&W Printer*, which is a subclass of *Printer*. We can see that any attribute existing in *Printer* is present in either of its subclasses. Note that definition 4.14 does not preserves attributes parents nor conditional probability distributions. Indeed, the set of parents of attribute *state* changes. Consequently its conditional probability distribution changes from class *Printer* to class *B&W Printer*. The same happens for attribute *hasInk* in class *Color Printer*, but not for attribute *hasPaper* which is identical to its version in class *B&W Printer*.



(a) The printer class



(b) Dependencies of the *Printer* class.



(c) Dependencies of the *ColorPrinter* class, which is a subclass of *Printer*.

Figure 4.5: Example of class inheritance. Dashed arcs represent dependencies with attributes in another class.

Attribute's type specialization

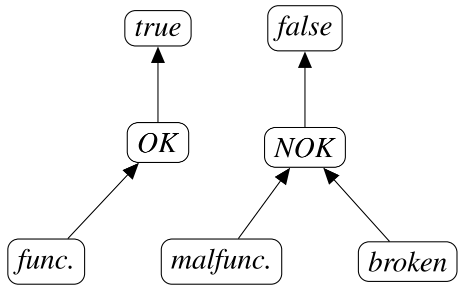
Attribute's types arise naturally when using PRMs and they are closely related to the notion of hierarchically structured variables (Sharma and Poole (2005)). There often exist several, if not many, random variables sharing a common domain in BNs, especially in BNs modeled by experts. Attribute's types inheritance purpose is to provide an additional behavior specialization to class inheritance. We have seen that definition 4.14 only allows to change inherited attributes conditional probability distributions, i.e., we can add and/or remove parents or just change their conditional probability distributions. This feature, yet useful, does not provide total satisfaction. Indeed, when specializing a class we often want to model new states that are specific to the subclass. For example, in the power surge example we could consider the following type for *Printer*'s *state* attribute: {*functional*, *dysfunctional*, *broken*}. The two values *dysfunctional* and *broken* are specific to printers: *dysfunctional* indicates that the printer prints but badly or is suffering some minor issue, e.g., a color is missing or the paper tray is empty. *Broken* states that the printer does not print anymore, there could be a paper jam or an internal component broken. In some cases we can cope with such specialization by adding new attributes, but in others we may want to overload attributes with specialized types.

Attribute's type inheritance is the process of decomposing types into more specific and precise domains. This includes partitioning types values into specific substates, e.g., *rainy* could be decomposed into *light rain*, *heavy rain*, *storm* and *typhoon*. Another possibility is to add intermediate values to model subtle differences, e.g., *OK* and *NOK* can be decomposed into *functional*, *malfunctioning* and *broken* to model the intermediate state between a functioning printer and a broken one (no more cyan ink does not prevent from printing black & white). Attribute subtyping can be modeled using a function mapping a subtype values to a subset of the super type values. Such functions are called domain generalization functions, since they generalize concepts present in subtypes.

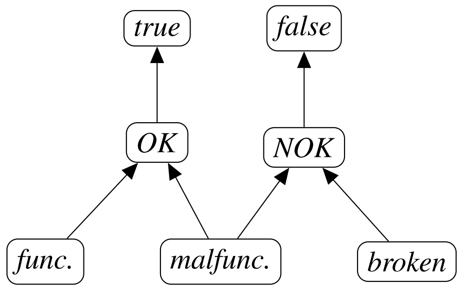
Definition 4.15 (Domain generalization function) *A domain generalization function is a function $\Phi : \tau \rightarrow \mathcal{P}(\lambda)$ where τ and λ are two distinct attribute types and $\mathcal{P}(\lambda)$ is subset of λ's values.*

Generally a subtype will have more values that its super type. However, there can be situations where we may not want to specialize a type in such manner. For example, let us consider a disease whith different variations. The set of all possible symptoms is $\{S_1, S_2, S_3\}$. Each variation may not present all three symptoms, but only a subset of them. While specializing the disease for each form, we may want to specialize the symptom's types into subtypes matching each variation. Furthermore, a domain generalization function is not necessarily a bijection, it can be a surjective or injective function. Indeed, in the previous example we define injective domain generalization functions: not all symptoms are present in each variation. In some cases, a surjective function will help define substates that overlap over several general concepts, as the *broken* state of a color printer illustrated in figure 4.6b. Attribute's type inheritance definition is straightforward now that we have introduced domain generalization functions.

Definition 4.16 (Attribute typing inheritance) *An attribute type τ inherits from another attribute type λ, denoted $\lambda \rhd \tau$, if it is defined using a domain generalization function $\Phi : \tau \rightarrow \mathcal{P}(\lambda)$. We say that λ generalizes τ or that τ specializes λ.*



(a) A subtype hierarchy where each domain generalization function returns singletons. In such situation cast descendant distributions are deterministic.



(b) A subtype hierarchy in which *malfunc.* either is state *OK* or *NOK*.

Figure 4.6: A type hierarchy with attribute types *boolean*, *t_state* and *t_malfunction*.

Given two types, we can always define a domain generalization function from one type to another. Consequently, definition 4.16 relies on how a type is defined. Such feature is, unfortunately, very implementation specific and the reader should see appendix A for a possible use of attribute's type inheritance. Figure 4.6 illustrates type inheritance. Arcs represent concepts specialization. For example, *t_malfunction* has three values: *functional* (*func.*), *malfunction* (*malfunc.*) and *broken*. In figure 4.6a, *finctional* and *malfunction* are specialization of *NOK*. In figure 4.6b an alternative representation is illustrated and *malfunctional* is also a specialization of *OK*. As is, attribute's type inheritance is only a semantic relation: *t_state*'s label *OK* is a sort of *true*, *broken* is a sort of *false*, etc. In the next section, we will show how to exploit such concepts probabilistically.

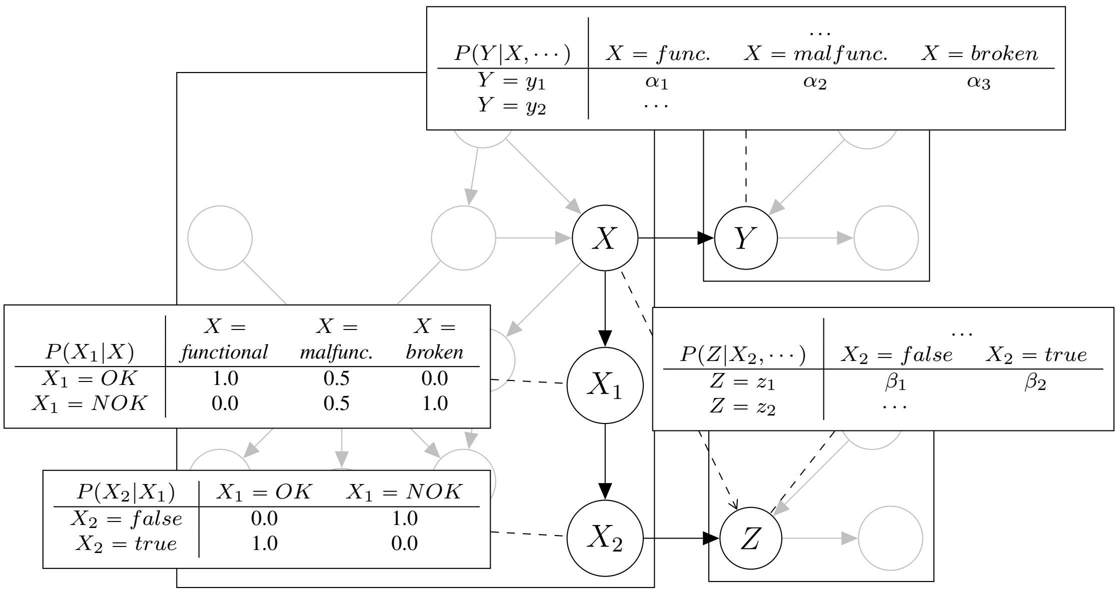
Attribute overloading

In the previous section we introduced attribute's type inheritance as a specialization mechanism for overloading attributes. If we overload attributes with different types, we will obviously break probabilistic dependencies: if a class $\mathcal{C}$ depends on $\mathcal{D}$'s attributes and that we connect an instance of $\mathcal{C}$ with an instance of $\mathcal{E}$, such that $\mathcal{D} \rhd \mathcal{E}$ and that some of $\mathcal{E}$'s attributes specialize $\mathcal{D}$'s, then $\mathcal{C}$'s instance dependencies will be broken. Indeed, by changing the domain of the underlying random variables, all conditional probability distributions specifications become erroneous. To allow subtype polymorphism, we must provide an efficient solution to preserve dependencies. To do so, we will present in this section cast descendants, a solution that allows attribute overloading using type inheritance and that preserves dependencies. We will first define the notion of type genealogy.

Definition 4.17 (Attribute type genealogy) *The genealogy of an attribute's type τ is a set of attribute's types $\{\lambda_1, \cdots, \lambda_n\}$ such that $\lambda_1 = \tau$ and $\lambda_i \rhd \lambda_{i+1}$ for $1 \leqslant i < n$ with λ_n is a type with no super type, called τ's ancestral type.*

A type's genealogy is an ordered set of all the type's super types. We will exploit types genealogies to automatically generate cast descendant attributes. These cast descendant attributes purpose is to *cast* an attribute into one of its super types. Cast descendants are automatically added, such that any attribute can be casted in any of its super types. Suppose that we have two classes $\mathcal{C}$ and $\mathcal{D}$ such that some attribute in $\mathcal{C}$ is overloaded in $\mathcal{D}$ using type specialization. If a third class, name it $\mathcal{E}$ has an attribute with a parent $\mathbf{K}.X = \mathcal{C}.X$, then the slot chain $\mathbf{K}$ can reference an instance of $\mathcal{D}$ by changing $\mathbf{K}.X$ to $\mathbf{K}.X_{\lambda_i}$. Consequently, slot chains will automatically be plugged to attributes of the correct type, see figure 4.7 for an example. The cost of adding cast descendants is negligible because if they are not used, i.e., neither queried nor referenced by a slot chain, we can discard them before inference as they will be barren nodes.

Definition 4.18 (Cast descendant) *Let $\mathcal{C}.X = \langle \tau, \pi(\mathcal{C}.X), \phi_{\mathcal{C}.X} \rangle$ be an attribute with τ's genealogy being $\{\lambda_1, \cdots, \lambda_n\}$. $\mathcal{C}.X$ cast descendants are the set of attributes $\{\mathcal{C}.X_{\lambda_i} = \langle \lambda_i, \{\mathcal{C}.X_{\lambda_{i-1}}\}, \phi_{\mathcal{C}.X_{\lambda_i}} \rangle\}$, $1 \leqslant i \leqslant n$ with $\mathcal{C}.X_{\lambda_0} = \mathcal{C}.X$. Each ϕ_{λ_i} is either user defined or the default cast distribution (see definition 4.19). We say that $\mathcal{C}.X_i$ is the direct cast descendant of $\mathcal{C}.X_{i-1}$.*



$P(Y \mid X, \cdots)$	$X = func.$	$X = malfunc.$	$X = broken$	$\cdots$
$Y = y_1$	α_1	α_2	α_3	
$Y = y_2$	$\cdots$			

$P(X_1 \mid X)$	$X =$ *functional*	$X =$ *malfunc.*	$X =$ *broken*
$X_1 = OK$	1.0	0.5	0.0
$X_1 = NOK$	0.0	0.5	1.0

$P(X_2 \mid X_1)$	$X_1 = OK$	$X_1 = NOK$
$X_2 = false$	0.0	1.0
$X_2 = true$	1.0	0.0

$P(Z \mid X_2, \cdots)$	$X_2 = false$	$X_2 = true$	$\cdots$
$Z = z_1$	β_1	β_2	
$Z = z_2$	$\cdots$		

Figure 4.7: X is an attribute with two cast descendant (X_1 and X_2). Attributes Y and Z both depends on X, however Z expects a boolean while Y expects an attribute of type *t_malfunction*.

Definition 4.19 (Default cast distribution) *Let $\tau = \{l_1^\tau, \cdots, l_n^\tau\}$ and $\lambda = \{l_1^\lambda, \cdots, l_m^\lambda\}$ be two attribute types of respective domain sizes n and m such that $\lambda \rhd \tau$. Given an attribute $\mathcal{C}.X = \langle \tau, \pi(\mathcal{C}.X), \phi_{\mathcal{C}.X} \rangle$, its direct cast descendant $\mathcal{C}.X_\lambda = \langle \lambda, \{\mathcal{C}.X\}, \phi_{\mathcal{C}.X_\lambda} \rangle$ and the domain generalization function $\Phi : \tau \mapsto \mathcal{P}(\lambda)$, the default cast distribution for $P(\mathcal{C}.X_\lambda|\mathcal{C}.X)$ is:*

$$P(\mathcal{C}.X_\lambda = l_i^\lambda | \mathcal{C}.X = l_j^\tau) = \begin{cases} 1/|\Phi(l_j^\tau)| & \text{if } l_i^\lambda \in \Phi(l_j^\tau), \\ 0 & \text{otherwise.} \end{cases}$$

Figure 4.7 illustrates an example of cast descendants using the default cast distribution with the domain generalization functions of figure 4.6b. Attribute X is of type *t_malfunction*, X_1 of type *t_state* and X_2 of type *boolean*. Figure 4.7 shows a partial representation of a system in which three instances are connected through dependency links $X \rightarrow Y$ and $X_2 \rightarrow Z$. Attributes Y and Z are both connected to X using a slot chain, but where Y expects an attribute of type *t_malfunction*, Z expects type *boolean*. Z's class must be connected to some super class of X's one and X must be an overloading using type specialization. The framework can easily detect that Z requires a subtype of X, and automatically connects it to the correct cast descendant.

Subtype polymorphism and dependencies preservation

In the precedent section we have mentioned subtype polymorphisms and we will now explain how such feature can be added to PRM's class inheritance scheme. To do so we will change definition 4.7 as follows:

Definition 4.20 (Relational Skeleton with subtype polymorphism) *A relational Skeleton $\mathcal{S}$ is a set of instances such that for any instance i of class $\mathcal{C}$ and any reference slot $\mathcal{C}.\rho = \mathcal{E}$, there exists at least one instance j in $\mathcal{S}$ such that j is an instance of $\mathcal{E}$ or of a subclass of $\mathcal{E}$ and $j \in range(i.\rho)$.*

This definition is almost identical to definition 4.7, except that definition 4.20 allows a wider range of instances to be referenced by reference slots. Another important feature regarding reference slot and class inheritance is the possibility of overloading reference slots. Reference slot overloading consists in specializing the range of an inherited reference slot. This is a useful feature when modeling complex systems as we will often first define classes using abstract concepts and specialize them using inheritance. In many cases we want to increase the complexity of the relationships between classes. We can now provide a proper definition of class inheritance in PRMs.

Definition 4.21 (Class inheritance) *A class $\mathcal{D}$ is a subclass of class $\mathcal{C}$, denoted $\mathcal{C} \rhd \mathcal{D}$, if:*

- *for each attribute $\mathcal{C}.X = \langle \tau_{\mathcal{C}.X}, \pi(\mathcal{C}.X), \phi_{\mathcal{C}.X} \rangle$, there exists an attribute $\mathcal{D}.X = \langle \tau_{\mathcal{D}.X}, \pi(\mathcal{D}.X), \phi_{\mathcal{D}.X} \rangle$ such that $\tau_{\mathcal{C}.X} = \tau_{\mathcal{D}.X}$ or $\tau_{\mathcal{C}.X} \rhd \tau_{\mathcal{D}.X}$;*

- *for each reference slot $\mathcal{C}.\rho = \mathcal{E}$, there exists a reference slot $\mathcal{D}.\rho = \mathcal{F}$ such that $\mathcal{E} = \mathcal{F}$ or $\mathcal{E} \rhd \mathcal{F}$.*

The fundamental property that must be present in any class inheritance mechanism is dependencies preservation. Preserving dependencies guarantees subtype polymorphism as it states that wherever a class is used in defining a probabilistic dependency, we can replace it by one of its subclasses without requiring to change any attribute or reference slot definition.

Property 4.1 (Dependencies preservation) *Let $\mathcal{C}$ and $\mathcal{D}$ be two classes such that $\mathcal{C} \rhd \mathcal{D}$. For each reference slot $\rho^{\mathcal{C}} \in \mathcal{R}(\mathcal{C})$, we denote by $\rho^{\mathcal{D}}$ its inherited version in $\mathcal{D}$ and for each attribute $X^{\mathcal{C}} \in \mathcal{A}(\mathcal{C})$, we denote by $X^{\mathcal{D}}$ its inherited version in $\mathcal{D}$. A class inheritance mechanism preserves dependencies if:*

- *for any slot chain $\mathbf{K} = \{\rho_1, \cdots, \rho_i^{\mathcal{C}}, \cdots, \rho_n\}$ we can substitute $\mathcal{C}$ referenced by ρ_{i-1} by $\mathcal{D}$ such that $\mathbf{K}' = \{\rho_1, \cdots, \rho_i^{\mathcal{D}}, \cdots, \rho_n\}$ is a legal slot chain;*
- *for any parent $\mathbf{K}.X$ with $\mathbf{K} = \{\rho_1, \cdots, \rho_n^{\mathcal{C}}\}$, we can substitute $\mathcal{C}$ referenced by ρ_{n-1} by $\mathcal{D}$ such that $\mathbf{K}' = \{\rho_1, \cdots, \rho_n^{\mathcal{D}}\}$ is a legal slot chain and $\mathbf{K}'.X$'s type equal or is a subtype of $\mathbf{K}.X$'s type.*

An illegal slot chain would be a slot chain $\mathcal{C}.\mathbf{K} = \{\rho_1, \cdots, \rho_n\}$ such that $\exists \rho_i \in \mathcal{C}.\mathbf{K}$, $\rho_i \notin \mathcal{R}(\mathcal{C}.\rho_1.\cdots.\rho_{i-1})$.

Theorem 4.1 *The class inheritance scheme defined by definition 4.21 preserves dependencies.*

Proof. Let us start by the first point of property 4.1. Let $\mathcal{C}$ and $\mathcal{D}$ be two classes such that $\mathcal{C} \rhd \mathcal{D}$. Suppose that there exists some legal slot chain $\mathcal{E}.\mathbf{K} = \{\rho_1, \cdots, \rho_i^{\mathcal{C}}, \cdots, \rho_n\}$ such that $\mathcal{E}.\mathbf{K}' = \{\rho_1, \cdots, \rho_i^{\mathcal{D}}, \cdots, \rho_n\}$ is not legal. Either $\rho_i^{\mathcal{D}} \notin \mathcal{R}(\mathcal{E}.\rho_1.\cdots.\rho_{i-1})$ or $\rho_{i+1} \notin \mathcal{R}(\mathcal{E}.\rho_1.\cdots.\rho_i^{\mathcal{D}})$.

The first case is trivial: since $\mathcal{D}$ is a subclass of $\mathcal{C}$, any reference slot referencing $\mathcal{C}$ can also refer $\mathcal{D}$ (see definition 4.20). The only situation where $\rho_i^{\mathcal{D}} \notin \mathcal{R}(\mathcal{E}.\rho_1.\cdots.\rho_{i-1})$ would be when $\mathcal{D}$ is not a subclass of $\mathcal{C}$ or that $\mathcal{E}.\mathbf{K}$ is not a legal slot chain, both propositions are contradictory with our hypothesis. In the second case, definition 4.21 guarantees that all reference slots in $\mathcal{C}$ are defined in $\mathcal{D}$, either as is or using reference slot overloading. Thus, if $\rho_{i+1} \notin \mathcal{R}(\mathcal{E}.\rho_1.\cdots.\rho_i^{\mathcal{D}})$ then $\mathcal{D}$ is not a subclass of $\mathcal{C}$ or $\mathcal{E}.\mathbf{K}$ was not a legal slot chain.

Now let us prove the second point of property 4.1. Suppose that attribute $\mathcal{E}.Y$ depends on $\mathcal{C}.X$ through the slot chain $\mathcal{E}.\mathbf{K}$ with $range(\mathcal{E}.\mathbf{K}) = \mathcal{C}$, i.e., $\mathcal{E}.\mathbf{K}.X \in \pi(\mathcal{E}.Y)$. Suppose that $\mathcal{E}.\mathbf{K}'.X$, with $\mathbf{K}' = \{\rho_1, \cdots, \rho_n^{\mathcal{D}}\}$, is not a legal parent of $\mathcal{E}.Y$. This implies either that $\mathcal{D}.X \notin \mathcal{A}(\mathcal{D})$ or that $\mathcal{D}.X$'s type differs from $\mathcal{C}.X$'s. However, by definition 4.21 there exists an attribute $\mathcal{D}.X$ such that its type equals $\mathcal{C}.X$'s or is a sybtype of $\mathcal{C}.X$'s type. If not, then $\mathcal{D}$ is not a subclass of $\mathcal{C}$ which is contradictory with our hypothesis. ■

Interfaces and multiple inheritance

Multiple inheritance is a key feature in most object-oriented programming languages and is essential for a strong object-oriented framework. Unfortunately, a problem arises when confronted with *diamond-shaped* inheritance, as illustrated in figure 4.8. An ambiguity results from how class $\mathcal{A}$'s properties are inherited by class $\mathcal{D}$, since two distinct paths exist from $\mathcal{A}$ to $\mathcal{D}$ (through $\mathcal{B}$ or $\mathcal{C}$). In programming languages there exist two solutions to handle multiple inheritance. The first one is straightforward as it forces subclasses to declare explicitly from whom each property is inherited. For PRMs, this implies that we should detail for each attribute which set of parents or CPT is used, e.g., from superclass $\mathcal{C}$ or $\mathcal{D}$. The second solution forbids multiple inheritance among classes and introduces new objects called interfaces.

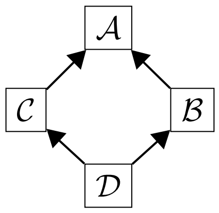


Figure 4.8: A diamond-shaped inheritance graph.

In object-oriented programming languages interfaces are a set of methods signatures, i.e., they do not provide any body for their methods. Consequently, interfaces cannot be instantiated but are used as an abstraction of concrete classes: from a programming perspective, a method's body is unnecessary to call it since only its signature is required. When a class implements an interface, the programmer must define a method and its code for each corresponding method in the implemented interface. Interfaces have been used in PGMs (DBNs, MSBNs, OOBNs, MEBNs). For PRMs, an interface would be an object with the minimal amount of information required to define probabilistic dependencies. This leads us to the following definition for an interface in PRMs.

Definition 4.22 (Interface) *An interface $\mathcal{I}$ is defined by a set of labeled types called abstract attributes, denoted $\mathcal{A}(\mathcal{I})$, and by a set of reference slots, denoted $\mathcal{R}(\mathcal{I})$. Interfaces cannot be instantiated.*

Interfaces are used to define dependencies among classes using abstraction. Given two classes $\mathcal{C}$ and $\mathcal{D}$, if an attribute $\mathcal{D}.Y$ depends on attribute $\mathcal{C}.X$, then the only information required for defining $\mathcal{D}.Y$'s conditional probability distribution is the type of $\mathcal{C}.X$. Abstract attributes represent the minimal amount of information required to define probabilistic dependencies among classes. However interfaces cannot be instantiated, but we know that any class implementing an interface will possess attributes and reference slots declared in that interface, thus we can use instances of such classes as substitute of the interface in a system. The same is possible with classes and their subclass and provides an efficient implementation of subtype polymorphism. A class can implement any number of interfaces as long as there is no name conflict. A name conflict appears if a class implements two interfaces that each possess an attribute or a reference slot with the same name but with incompatible types. In such a case, it is impossible for the class to satisfy both interfaces.

Definition 4.23 (Interface implementation) *A class $\mathcal{C}$ implements an interface $\mathcal{I}$ if and only if:*

- *for each abstract attribute $\mathcal{I}.X$ of type λ, there exists an attribute $\mathcal{C}.X = \langle \tau, \pi(\mathcal{C}.X), \phi_{\mathcal{C}.X} \rangle$ such that $\lambda = \tau$ or $\lambda \rhd \tau$;*
- *for each reference slot $\mathcal{I}.\rho = \mathcal{E}$, there exists a reference slot $\mathcal{C}.\rho = \mathcal{F}$ such that $\mathcal{E} = \mathcal{F}$ or $\mathcal{E} \rhd \mathcal{F}$.*

A class can implement any number of interfaces as long as it does not create any name conflict.

As for classes, we can define inheritance among interfaces. The principle is similar to definition 4.21 and offers a simple mechanism to specialize interfaces. Since there is no probabilistic semantic in interfaces, we do not have any limitation on the number of interfaces an interface can specialize.

Definition 4.24 (Interface inheritance) *Let $\mathcal{I}$ and $\mathcal{J}$ be two interfaces. We say that $\mathcal{J}$ is a sub interface of $\mathcal{I}$, denoted $\mathcal{I} \rhd \mathcal{J}$, if:*

- *for each abstract attribute $\mathcal{I}.X$ of type λ there exists an attribute $\mathcal{J}.X$ of type τ such that $\lambda = \tau$ or $\lambda \rhd \tau$;*
- *for each reference slot $\mathcal{I}.\rho = \mathcal{E}$ there exists a reference slot $\mathcal{J}.\rho = \mathcal{F}$ such that $\mathcal{E} = \mathcal{F}$ or $\mathcal{E} \rhd \mathcal{F}$.*

Figure 4.9 shows an example of an interface implementation, where the two classes *B&W Printer* and *Color Printer* implement interface *Printer* (which is no longer a class for this example). The *Printer* interface defines the minimal set of attributes and reference slots any printer must declare: a reference to its room, attributes to represent its ink and paper status and attribute to represent if it the printer is functional. Figure 4.9 is also an alternative representation of classes using a UML syntax. Such syntax is necessary to point out attributes and reference slots types and prevents from overloading the class dependency graphs with attribute types (long names often do not fit within in graphical user interfaces or UML diagrams).

We can also derive from theorem 4.1 the following corollary.

Corollary 4.1 *The interface inheritance scheme defined by definitions 6.1 and 4.24 preserves dependencies.*

Proof is omitted as it is almost identical to theorem 4.1 proof.

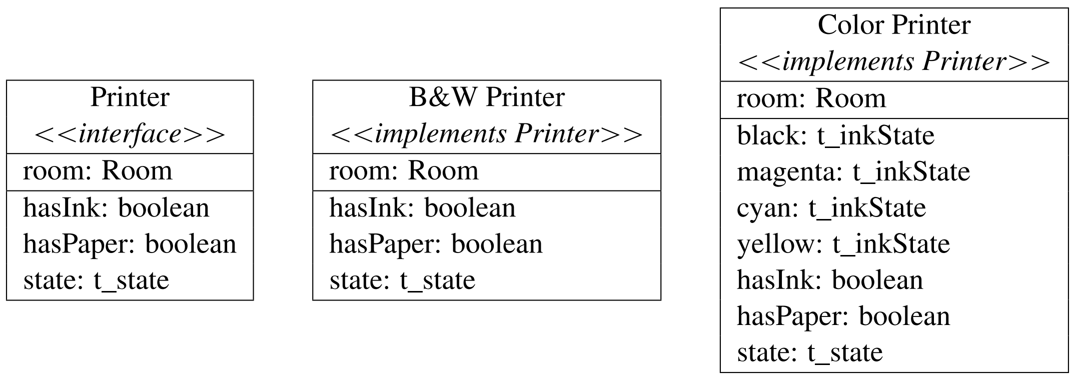


Figure 4.9: An interface and two classes implementing it.

PRMs and visibility

We will conclude section 4.2 with a discussion about the use of visibility in PRMs. Visibility in object-oriented programming languages is composed of three different visibility descriptors: public, private and protected. They rule which elements are publicly visible, i.e., what elements are accessible outside of their class; which elements are only visible to elements of the same class and element only accessible to subclasses. Table 4.1 resumes these informations.

Keyword	scope
Public	Attributes and reference slots can be accessed by all other classes.
Private	Attributes and reference slots can only be accessed by elements of the same class.
Protected	Attributes and reference slots can only be accessed by elements of the same class and by elements of subclasses.

Table 4.1: The different visibilities and their associated scope.

4.3 Parfactor representation of PRMs

FOPMs are a popular framework and have received a considerable amount of contributions, mostly on extending BNs and MNs to the first-order paradigm. Furthermore, lifted inference is a dedicated probabilistic inference algorithm that exploits FOPMs (see section 3.4). Lifted inference algorithms use the parfactor framework, a minimalistic FOPM. This section purpose is to demonstrate how an object-oriented framework such as PRMs can be expressed using parfactors. We recall here the definition of parfactors.

Definition 4.25 *A parfactor is a triple $\langle C, V, t \rangle$ where C is a set of constraints on parameters, V is a set of parametrized random variables and t is a table representing a* *factor from the random variable to the non-negative reals.*

Algorithm 4 details formally how an attribute can be converted into a set of parfactors. We will detail this algorithm using attribute *state* of class *Color Printer* from figure 4.5c. When we consider parfactors, we cannot miss the link between random variables parameters and classes. Indeed, both notions fulfill the same goals: defining relations among random variables. If we consider all factors with the same parameter, we are considering a group of random variables sharing some common semantics. Parfactors with several parameters exhibit the relations between different groups of random variables. Consequently, we represent classes as random variables parameters. To ensure the exact representation of the structure encoded by the classes of a PRM, it is necessary to use two different types of constraints.

To represent classes, class inheritance and interface implementation we will use *isA()*-like constraints (e.g. *isAPrinter(X)*). Relations can be expressed as binary constraints in which each parameter has a *isA()* constraint. For example, attribute state of class *Color Printer* can be represented by the following parfactor:

$$\begin{aligned}\langle\ &\{isAColorPrinter(X) \wedge isARoom(Y) \wedge \\ &\ isAPowerSupply(Z) \wedge room(X,Y) \wedge power(Y,Z)\}, \\ &\{malfunction\_works(X),\ paperType\_hasPaper(X), \\ &\ Boolean\_hasInk(X),\ state\_works(Z)\},\ t\ \rangle.\end{aligned}$$

The first part of this parfactor is composed of type constraints (*isAColorPrinter()*, *isARoom()* and *isAPowerSupply()*) and relational constraints (*room()* and *power()*). The second part contains the dependencies of the parfactor, which are the parametrized random variables *malfunction_works(X)*, *paperType_hasPaper(X)*, *Boolean_hasInk(X)* and *state_works(Z)*. The Cartesian product of their values is mapped to the values in *t*, which represent the CPT of *ColorPrinter.works*.

Algorithm 4: Parfactor generation of an attribute.

```
   Input: Class c, Attribute attr
   Output: Parfactor fctr
1  Parfactor fctr;
2  Add a isA() constraint over c's type;
3  Add a parametrized variable named by attr and prefixed by attr's type;
4  foreach parent prt of attr do
5      if prt not in $\mathbf{A}(c)$ then
6          Add a isA() constraint over prt's class type;
7          foreach reference $\rho$ in the slot chain from attr to prt do
8              Add a relational constraints in fctr matching $\rho$;
9              Add a isA() constraint over $\rho$ range type;
10     Add a parametrized variable named by prt and prefixed by prt's type;
11 Copy in fctr's table attr's CPT;
12 return fctr
```

The *isA()* constraints encode the inheritance scheme of a PRM if, for each instance i of a system, a ground variable is declared for each type of i, i.e., for all of its super classes and implemented interfaces. For example, an instance *coloria* of the *Color Printer* class will be represented with the following ground variables: *isAColorPrinter(coloria)* and *isAPrinter(coloria).*

Finally, cast descendants can be represented by including types names in the parametric variables declarations. For example *malfunction_works(X)* stands for the attribute *ColorPrinter.works* of type *malfunction*. Then, by generating parfactors for each cast descendant, the constraints names will ensure the correct structure. For example, the cast descendant *ColorPrinter.works* will be declared as:

$$\begin{aligned}&\langle\ \{isAColorPrinter(X)\}\\ &\ \ \{state\_works(X),\ malfunction\_works(X)\},\ t\ \rangle\end{aligned}$$

At first sight, such a representation seems cumbersome but it illustrates the expressive power of parfactors and of FOPMs. First-order logic can be used to express very complex relations: only two types of constraints are necessary to represent all the notions presented in this paper. However such expressive power has a major flaw as semantics and relations are hidden in the mass of constraints declarations. When dealing with large-scale systems, creating and maintaining such knowledge base can be extremely difficult. Reinforced object-oriented PRMs are a proposition to manage such knowledge with a formalism less expressive but much more scalable.

Chapter 5

Structured Probabilistic Inference

Probabilistic inference is often considered the prime task of PGMs and an efficient inference scheme is essential. For most BN extensions, probabilistic inference reduces to ground inference: given a model, a BN is generated and used for inference. In some cases, specific inference algorithms are designed: time slices repetition is exploited to compute an efficient elimination order for 2-TBNs, hyper-tree triangulation enables distributed inference for MSBNs and first-order logic is used to lift identical worlds for parfactors. OOBNs, and by extension PRMs, also benefit from a dedicated inference scheme: structured inference.

Structured probabilistic inference have already been explored in (Pfeffer, 1999). Pfeffer's preliminary work set the basis of the contributions presented in this chapter. We have found extremely difficult to reuse the algorithms proposed in (Pfeffer, 1999) and our first approach was to redefine SVE, an algorithm that exploits structural information encoded in OOBNs and PRMs. SVE exploits hierarchical and structural inference to prevent redundant computations. It uses the information encoded by classes to eliminate attributes at class level, the resulting computations can then be applied to each class' instances. However, SVE suffers from several shortcomings which have lead us to develop a new approach to structured inference. Another unanswered issue concerns d-separation analysis and structured inference. Indeed, both approaches seem at first incompatible. Where d-separation analysis breaks the structure encoded by classes to only consider relevant attributes, structured inference needs that structure to detect repetitions and prevent redundant computations. We propose a solution that couples both approaches and offers a substantial performance gain.

In this chapter, we will first discuss the limitation of ground inference. We will then introduce our version of SVE and expose its inefficiency to deal with certain families of systems. We will then present a new inference algorithm that generalizes SVE. From there, we will expose our adaptation of the BayesBall (BB) algorithm to PRMs and conclude with experimental results.

5.1 Ground inference limits

Ground inference's main limitation is its lack of scalability. When using an object-oriented framework, specifying large scale networks is simple. Let us consider the power surge example, which is a simple network. It includes four classes, one with zero attribute, two classes with one attribute and one class with three attributes (one of them is an aggregator). Each attribute's domain size is binary, but the overall tree-width of the system can be large depending on the number of printers connected to computers. Figure 5.1 is a sample ground network in which we have four rooms, each with twenty computers and two printers. While this shows the benefits from using object-oriented frameworks to model large scale systems: only the four classes in figure 4.3 are necessary to specify the ground network in figure 5.1; it also gives a good indication of the memory consumption of generating ground BNs.

Definition 4.13 gives us a general outline of the grounding algorithm. Grounding a PRM is pretty straightforward: each attribute becomes a node in the ground BN and each CPT is copied for each ground attribute. A naive approach would build a BN by allocating memory for each CPT copied that way. This can cause high memory consumptions and will restrain from grounding large relational skeletons. For example, using a PRM to represent figure 5.1 five CPTs are required (four attributes and one aggregator), the ground BN of figure 5.1 requires $1 + 4 * (20 * 3 + 2) = 249$ CPTs.

However, we can imagine *smart* ground networks preventing redundant CPTs copies. To do so, we would need to dereference each CPT to a unique memory location, thus each unique CPT is stored once in memory. Unfortunately, such solution is inconvenient for several reasons. Suppose we decide to use a junction tree algorithm: creating the junction tree and initializing each clique breaks the memory gain, unless we design a specific junction tree algorithm to exploit structural repetition. By doing so, we lose the main advantage of ground inference (to use existing inference algorithms) and we are creating a new inference algorithm that should be specified for PRMs rather for ground BNs. If we choose to use existing softwares for inference over the ground BN, we are confronted to another problem. Indeed, if we do so we will limit ourselves to small networks since such software do not take advantage of structural repetition. Commercial softwares often require some file format, e.g., BIF, and dumping the ground BN into such format would invalidate any benefits from a smart implementation of ground BNs.

Figure 5.2 illustrates the limitation of ground inference. We have performed inference over a system generated from the power surge example, with three different inference algorithms: VE, VE coupled with BayesBall (VEBB) and Shafer-Shenoy. The queried variable is the state attribute of a randomly chosen computer and there is no evidence. The system had 1 power supply, 40 rooms and 6 printers per room. We increased the number of computers per room from 10 to 200 and we see that we reached the memory limit of the computer[1] around 130 computers per room, which explains why the graphs of figure 5.2 ends around 130 computers per room. Finally we can see that inference time increases drastically with the number of computers per room and yet the

[1]The computer was a i686 quad core @1998.0 MHz and with 3 Go of memory.

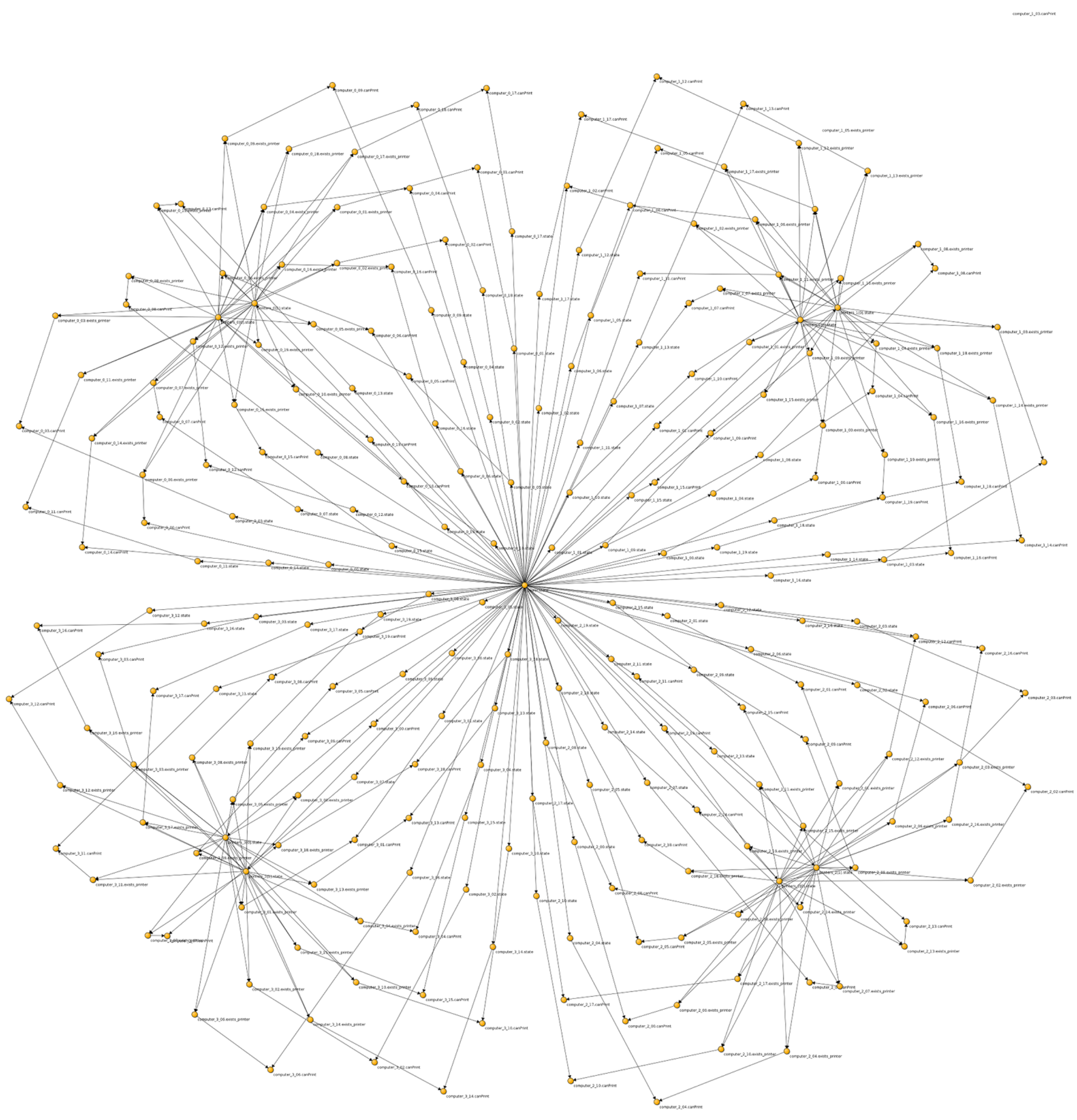


Figure 5.1: Ground BNs does not encode structural repetition. This application of the power surge example contains 249 nodes, but there are only five distinct CPTs.

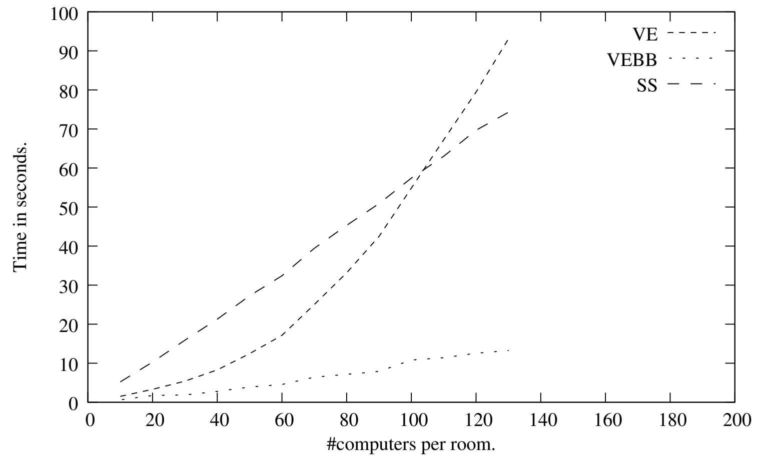


Figure 5.2: The limits of ground inference.

PRMs we used here is simple and does not offer many challenges except for its size.

Inference in PRMs is hard both for memory consumption and execution time, as even a simple set of classes can be used to model challengingly large systems. If we consider exploiting smart ground BNs, we obviously need to develop dedicated inference algorithms and by doing so there are few reasonable arguments to not develop such algorithms directly for PRMs. We have seen that PRMs can reduce memory consumption, but we can optimize other criteria by reasoning at an object-oriented level. This is precisely the purpose of SVE.

5.2 Structured Variable Elimination

Structured Variable Elimination (SVE) is a probabilistic inference algorithm proposed in Pfeffer (1999). Its main feature is to offer a dedicated inference algorithm for OOBNs and PRMs. Unfortunately, SVE has several shortcomings. From an experimental perspective, SVE lacks solid experimentations, since the experiments in Pfeffer (1999) do not detail precisely the networks used and the experimental environment. For example tree-widths are not given. From a formal perspective, the use of PRMs in Pfeffer (1999) is profoundly different from the use of PRMs in this thesis. Indeed, in this thesis we focus on closed world systems. In closed world systems, all instances and references are known, i.e., there is no structural uncertainty. Furthermore, the use of SVE to OOBNs is incompatible with PRMs as most of the structural constraints present in OOBNs are not present in PRMs. In this chapter we will focus on theses issues and explain why we chose to adapt SVE to be efficient on PRMs used to model closed world systems.

Hierarchical inference in Probabilistic Relational Models

We will first detail the specificities of OOBNs exploited by SVE. SVE's principle is similar to specific triangulation algorithms used for DBNs: it exploits structure and a

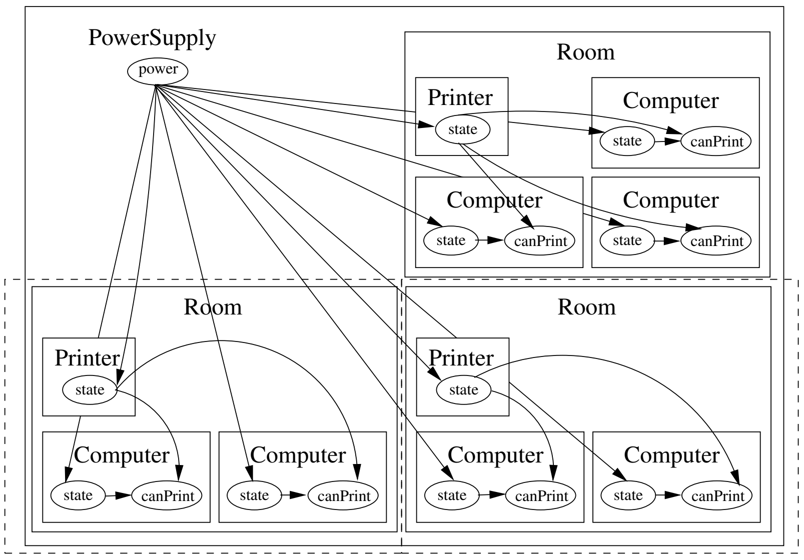


Figure 5.3: Illustrating the power surge example using an OOBN representation. Aggregators are not represented since such notion does not exist in OOBNs.

heuristic to build good elimination orders. This is possible because of OOBNs hierarchical nature, i.e., that relations between objects entail that an object will always be encapsulated into another one. In chapter 3.2 we have seen an example where the object *Hard Drive* is encapsulated by the object *Computer*. Consequently, we can define an interface for each object that will d-separate its internal components from the outer ones. This feature is called hierarchical inference. Figure 5.3 illustrates a system instantiated from the power surge example, in which we have one power supply, two rooms with two computers and a third room with three computers. All rooms have a single printer connected to all other computers in the same room. Hierarchical inference will exploit encapsulation and eliminate inner objects first. In figure 5.3 this leads to the elimination of each computers and printers before eliminating rooms and the power supply.

Such elimination is still possible in PRMs, however probabilistic dependencies defined using slot chains do not allow to infer this elimination from the sole information encoded in the relational skeleton. Consequently, there is no viable solution to exploit hierarchical inference in PRMs (either in Pfeffer's version or in ours) as it is done in OOBNs. However, we can extend hierarchical inference to PRMs using a simple statement: if we cannot infer an encapsulation order between instances, we can do so with attributes. Indeed, we can circumvent the absence of hierarchy by defining three sets of attributes: inner, output and external attributes. We have already defined the notion of output attributes (see definition 4.11). Inner attributes are any attribute that is not an output attribute. For example, attributes *canPrint* and *state* of the *Computer* class are inner attributes. Output attributes are any attribute with at least one child outside of its class, e.g., attribute *power* of class *PowerSupply*. Note that class level output attributes can be inner attributes at instance level. Consider an instance of *Printer* that is not connected to a computer, then its *state* attribute is no longer an output attribute. External attributes of a class $\mathcal{C}$ are attributes not in $\mathcal{A}(\mathcal{C})$ but present in CPTs of $\mathcal{C}$'s attributes, i.e., they are accessed from $\mathcal{C}$ using slot chains.

Eliminated attributes	Potentials
$\{\}$	$P(C.state \mid Pow.state)$, $P(C.exists \mid Prnt.state)$, $P(C.canPrint \mid C.state, C.exists)$
$\{C.canPrint\}$	$P(C.state \mid Pow.state)$, $P(C.exits \mid Prnt.state)$, $\Phi_1(C.state, C.exist)$
$\{C.canPrint, C.state\}$	$P(C.exits \mid Prnt.state)$, $\Phi_2(C.exist, Pow.state)$
$\{C.canPrint, C.state, C.exists\}$	$\Phi_3(Pow.state, Prnt)$

Table 5.1: Resulting factor after eliminating inner attributes of the *Computer* class. Remaining random variables belong to attributes defined in other classes.

We can exploit differences between inner, output and external attributes to recreate a hierarchical structure in PRMs. Table 5.1 illustrates factors obtained after eliminating *Computer*'s inner attributes. Note that the *Computer* class only has inner attributes. Factor Φ_2 only contains external attributes, i.e., attributes that are not part of *Computer* set of attributes. Now, let us generalize this principle. Given a class $\mathcal{C}$, the CPTs associated with $\mathcal{C}$'s attributes can either contain inner attributes, denoted $\mathcal{A}_{in}(\mathcal{C})$, output attributes, denoted $\mathcal{A}_{out}(\mathcal{C})$, or external attributes, denoted $\mathcal{A}_{ext}(\mathcal{C})$. Note that $\mathcal{A}_{in}(\mathcal{C}) \cup \mathcal{A}_{out}(\mathcal{C}) = \mathcal{A}(\mathcal{C})$ and $\forall X \in \mathcal{A}_{ext}(\mathcal{C})$, $X \notin \mathcal{A}(\mathcal{C})$. The factor obtained after eliminating all inner attributes will be:

$$\phi(\mathcal{A}_{out}(\mathcal{C}), \mathcal{A}_{ext}(\mathcal{C})) = \sum_{X \in \mathcal{A}_{in}(\mathcal{C})} \prod_{X \in \mathcal{A}(\mathcal{C})} P(X|\pi(X)). \tag{5.1}$$

There is however a category of attributes that cannot be eliminated along with inner attributes at class level: aggregators. Indeed, these attributes are used to aggregate information of an unknown number of parents. Consequently, aggregators conditional probability distributions at class level are not representative of their instantiated versions. In some cases, aggregators are defined using simple reference slots, but in the general case there is no possible way to know what their conditional probability distributions will be once instantiated. Fortunately, we only need to consider aggregators as output attributes as we will see that output attributes are also dependent of their instance's context.

Structured inference in PRMs

Hierarchical inference only provides a predefined elimination order and, unfortunately, we cannot guarantee that such elimination order is efficient in the general case. It can even be, in some situations, particularly inefficient. To increase hierarchical inference's performance, we must also use structural inference. Structural inference exploits structure repetition to prevent redundant computations. For example, in figure 5.3 we can see two different kind of repetitions. The first repetition concerns classes and their instances, in figure 5.3 we can see that the computer class has seven instantiations, the *Printer* class has three and so does class *Room*. Hierarchical inference can be exploited to reuse factors obtained after eliminating inner attributes. For instance, the factors resulting from *Computer*'s inner attributes elimination can be reused for each of its instantiation. The second repetition is patterns repetition. Patterns are sets of connected instances repeated throughout a system. In figure 5.3 we can see that both outlined rooms share an identical pattern. Eliminating attributes that are encapsulated in such pattern results in factors that can be reused in each of the pattern's occurrences. However, exploiting these patterns is a remarkably difficult task and is the topic of chapter 6.

In this chapter, we will focus on exploiting structured inference over classes. Our goal is to reuse inner attribute's class level elimination for each suitable instance. We say suitable instance because not all instances are eligible to receive the factor obtained after class level eliminations. Indeed, evidence on inner attributes breaks structure by locally changing an instance's conditional probability distributions. This prevents any reuse of class level elimination and could be a major flaw as evidence update is one of the main use of PGMs. We will see that it is not that much a burden as exploiting d-separation analysis helps optimizing inference in systems with evidence. Coupling d-separation analysis and structural inference is the topic of section 5.4.

Adapting SVE to object-oriented PRMs

We will now adapt the SVE algorithm of Pfeffer (1999) to object-oriented PRMs. In its traditional form, SVE does not exploit hierarchical or structural inference on named instances (see chapter 3.3) and handles them not differently than ground inference does. Since we only consider systems in which all instances are named, we cannot use SVE in its classic form. However, Pfeffer introduces several notions that can be used in our context. We have adapted two of them: hierarchical and structural inference. The final notion we will exploit is a specific elimination order of instances called a bottom-up elimination. Bottom-up eliminations of instances consist in eliminating instances with no output attribute first.

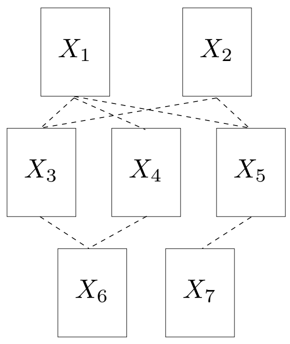
Definition 5.1 (Leaf instances) *A leaf instance is an instance with no output attribute. An instance has no output attribute either because of the relational skeleton topology or because all instances that had dependencies on that instance have been eliminated.*

To understand the importance of bottom-up eliminations and leaf instances, we will consider the example illustrated by figure 5.4. Figure 5.4 represents a system containing seven instances. We can see that two of them are leaf instances: X_6 and X_7 (we suppose that all instances are of the same class, name it $\mathcal{C}$). The black node of instance X_4 is the queried attribute. A bottom-up elimination eliminates first leaf instances and to find them we follow inverse slot chains starting from the queried instance. This is illustrated throughout figures 5.4b to 5.4c: we start from X_4 (figure 5.4b), then go down in the system (figure 5.4c). When a leaf instance is reached, we eliminate it and obtain a factor over its external attributes. For example, the external attributes of X_6 are attributes of instances X_3 and X_4. Now that we have reached a leaf instance, we follow up its slot chains to reach instances containing its external attributes. If these instances are leaves, we proceed with their elimination, and so on.

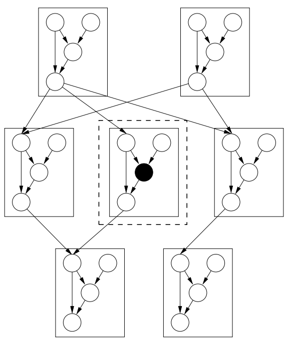
Let us focus on the elimination of instance X_6. When we eliminate X_6's attributes, we obtain factors over its external attributes. Such factors are stored in a global set called the pool of factors (referred as the pool). Later on, when X_3 is chosen for elimination, we eliminate its inner attributes, resulting in factors over its output and external attributes. Since all attributes depending on its output attributes have been eliminated, we can eliminate them if we take into account the factors in pool. Without a bottom-up elimination, we could not guarantee that all dependencies have been dealt with. The method we described can be applied on X_6 (figure 5.4c), then X_4 (figure 5.4d), then X_3 (figure 5.4e).

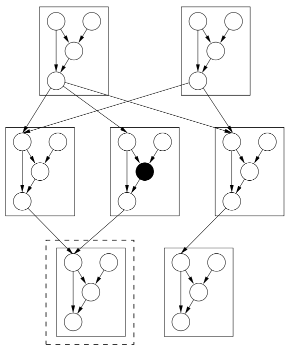
Now let us consider step 5, illustrated by figure 5.4f. In step 5 we have reached instance X_1 that still has existing dependencies. Thus, before eliminating X_1 we must eliminate instances that have dependencies on its output attributes. This results in a recursive call over inverse slot chains illustrated on figures 5.4g and 5.4h. On the latter we reached a leaf where the preceding paragraph can be applied, hence resulting in the elimination of X_7 (figure 5.4h), and then of X_5 (figure 5.4i), as eliminating X_7 makes X_5 a leaf instance and eligible to elimination. The remaining instances, X_1 and X_2, become leaf instances after X_5 elimination and can be eliminated.

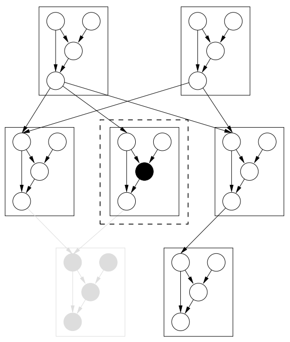
Algorithm 5 is our adaptation of Pfeffer's SVE to close world systems. SVE requires as inputs a PRM Π, a relational skeleton $\mathcal{S}$, a query attribute Q, its instance q and a set of evidence $\mathbf{e}$. Evidence is encoded by factors over the observed attributes, as for algorithm 1 (see chapter 2.2). SVE's output is a factor over q encoding $P(q|\mathbf{e})$. Algorithm 5 also uses three different calls to the procedure VE (line 9, 11 and 13). How algorithm 1 is used is crucial to algorithm 5 and each call will be detailed later on. The first step of algorithm 5 is to initialize a list and a set of instances with q (lines 2-3), lst's purpose is to proceed with a depth-first search in $\mathcal{S}$ and $visited$ purpose is to prevent from visiting instances twice. Until lst is empty (line 4), algorithm 5 picks the first element of the list, name it i, and if it is a leaf it proceeds with its elimination (lines 6-13). Three possible eliminations can occur. If i is the queried instance we proceed with a call of VE over i's attributes and takes into account the queried attribute Q and factors in $pool$ and in $\mathbf{e}$ (lines 8-9). If i has evidence, VE is applied to i's attribute and takes into account factors in $pool$ and in $\mathbf{e}$ (lines 10-11). If i has no evidence, VE is applied first on i's class to eliminate inner attributes at class level. The resulting factors are cached to be reused at the next occurrence of an instance of i's class. After the class level elimination, VE proceeds with the elimination of i's output attributes and takes into account factors in $pool$ and in $\mathbf{e}$ (lines 12-13). The next step of algorithm 5 is to add i's dependencies to lst. Such dependencies include any instance in which attributes are parent of some of i's attribute (lines 14-18). If i is not a leaf, then we parse its

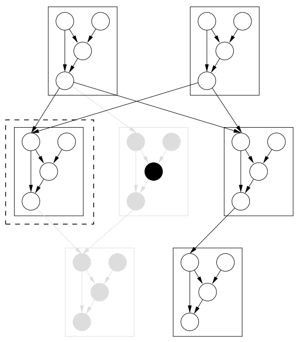


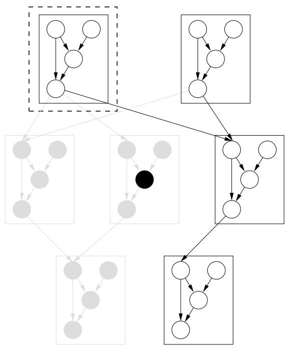
(a) A relational skeleton.

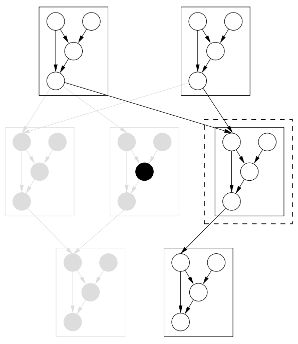

(b) Step 1.

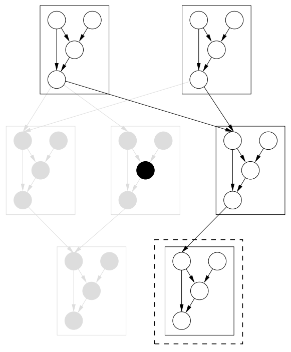

(c) Step 2.

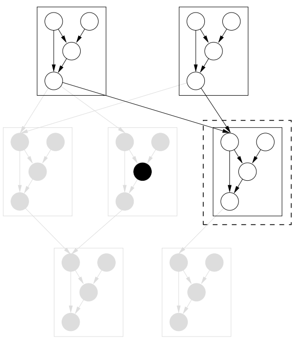

(d) Step 3.

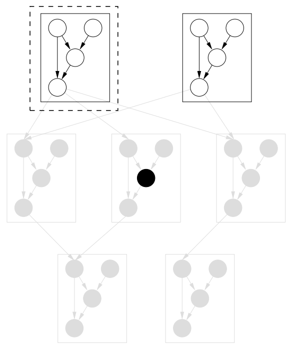

(e) Step 4.

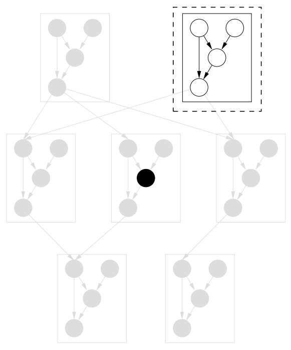

(f) Step 5.


(g) Step 6.


(h) Step 7.


(i) Step 8.


(j) Step 9.


(k) Step 10.

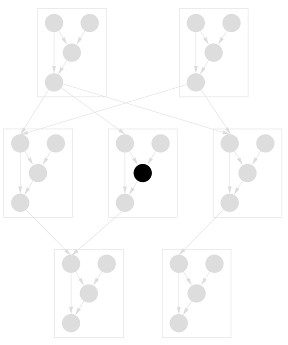

(l) Step 11.

Figure 5.4: Illustrating SVE's bottom-up elimination order. Outlined instances are considered for elimination and light gray instances are eliminated. The black node is the query.

Algorithm 5: Structured Variable Elimination

```
   Input: a PRM $\Pi$, a relational skeleton $\mathcal{S}$, a query $q.Q$, a set of evidence $\mathbf{e}$
   Output: a factor encoding $P(q|\mathbf{e})$
 1 let $pool = \varnothing$ be an empty set of factors;
 2 let $lst = [q]$ be a list of instances;
 3 let $visited = \{q\}$ be a set of instances;
 4 while $lst$ is not empty do
 5     $i = lst.front()$;
 6     if $i$ is a leaf instance then
 7         $lst.pop\_front()$;
 8         if $i$ equals $q$ then
 9             VE1$(q, Q,\ \mathbf{e},\ pool)$;
10         else if an inner attribute of $i$ has evidence then
11             VE2$(i,\ \mathbf{e},\ pool)$;
12         else
13             VE3$(Class(i),\ i,\ pool)$;
14         foreach slot chain $\mathrm{K}$ of $i$ do
15             foreach $j \in i.\mathrm{K}$ do
16                 if $j \notin visited$ then
17                     $visited = visited \cup \{j\}$;
18                     $lst.push\_front(j)$;
19     else
20         foreach inverse slot chain $\mathrm{K}$ of $i$ do
21             foreach $j \in i.\mathrm{K}$ do
22                 if $j \notin visited$ then
23                     $visited = visited \cup \{j\}$;
24                     $lst.push\_front(j)$;
25 Let $\phi$ be a factor over $q$ initialized with ones;
26 foreach factor $\psi$ in pool do $\phi = \phi \times \psi$;
27 $normalize(\phi)$;
28 return $\phi$;
```

inverse slot chains to add all instances that depend on one of i's output attributes (lines 19-24). Note that in such case, we do not remove i from lst. Finally, once lst is empty and all instances have been eliminated, algorithm 5 retrieves all factors containing the queried attribute Q in $pool$, factorizes them, normalizes the result and returns the factor encoding $P(q|\mathrm{e})$ (lines 25-28).

Procedure VE1 eliminates queried instance's attributes. We need a different treatment for this instance to not eliminate the queried attribute. Procedure VE1 is simple:

```
Procedure VE1($q$, $Q$, e, $pool$)
   Input: An instance $q$, an attribute $Q$, a set of evidence e, a set of factor $pool$
 1 Compute an elimination order $t$ over $\mathcal{A}(q)\backslash\{Q\}$;
 2 foreach Attribute $A \in \mathcal{A}(q)$ do
 3     add $q.A$'s CPT to $pool$;
 4     if $q.A$ is observed then add evidence in e over $q.A$ to $pool$;
 5 Apply algorithm 1 using $t$ over $pool$;
```

```
Procedure VE2($i$, e, $pool$)
   Input: An instance $i$, a set of evidence e, a set of factor $pool$
 1 Compute an elimination order $t$ over $\mathcal{A}(i)$;
 2 foreach Attribute $A \in \mathcal{A}(i)$ do
 3     add $i.A$'s CPT to $pool$;
 4     if $i.A$ is observed then add evidence in e over $i.A$ to $pool$;
 5 Apply algorithm 1 using $t$ over $pool$;
```

after computing an elimination order over the instance's attributes (except $q.Q$), it adds each CPT to $pool$ and each evidence of an attribute in q to $pool$. Finally, it applies algorithm 1, which proceeds with the elimination of q's attributes. Procedure VE2 is almost identical to procedure VE1, the difference lies in the fact that all attributes of i are eliminated, where the queried attribute is preserved in procedure VE1.

Procedure VE3 proceeds with the elimination of i's attributes in two steps. The first step eliminates the inner attributes at class level (line 2-9) and copies the obtained factors into their instance level counterparts (lines 10-15). When the instance level factors are obtained, procedure VE3 eliminates output attributes (line 17). The $Cache(\mathcal{C})$ procedure returns a set of factors associated with class $\mathcal{C}$.

We will now prove that algorithm 5 correctly computes the value $P(q|\mathrm{e})$ under certain conditions. To do so we must first define a category of classes that prevents algorithm 5 convergence.

Definition 5.2 (Mutually dependent classes) *Let $\{\mathcal{C}_1, \cdots, \mathcal{C}_n\}$ be a set of classes such that $\mathcal{C}_1 = \mathcal{C}_n$. Classes $\mathcal{C}_1$ to $\mathcal{C}_n$ are said to be mutually dependent if for $1 \leqslant i < n-1$, $\mathcal{C}_i$ is dependent of $\mathcal{C}_{i+1}$, i.e., one of $\mathcal{C}_i$'s attributes is the child of one of $\mathcal{C}_{i+1}$'s attributes.*

Mutually dependent classes appear once a list of classes creates a reference cycle in the dependency class diagram. Such mutually dependent classes can be instantiated into a set of mutually dependent instances.

Definition 5.3 (Mutually dependent instances) *Let $\{c_1, \cdots, c_n\}$ be a set of instances such that $c_1 = c_n$. Instances c_1 to c_n are said to be mutually dependent if for $1 \leqslant i < n-1$, c_i is dependent of c_{i+1}, i.e., one of c_i's attributes is the child of one of c_{i+1}'s attributes.*

Procedure VE3($\mathcal{C}$, i, **e**, $pool$)

```
Input: A class $\mathcal{C}$, an instance $i$, a set of factor $pool$
1  let $bucket = \emptyset$ be an empty set of factors;
2  if $Cache(\mathcal{C})$ exists then
3      $bucket = Cache(\mathcal{C})$;
4  else
5      Compute an elimination order $t$ over $\mathcal{A}_{in}(\mathcal{C})$;
6      foreach Attribute $A \in \mathcal{A}(\mathcal{C})$ do
7          add $\mathcal{C}.A$'s CPT to $bucket$;
8      Apply algorithm 1 using $t$ over $bucket$;
9      $Cache(\mathcal{C}) = bucket$;
10 foreach factor $\phi \in bucket$ do
11     let $\{\mathcal{C}.X_1, \cdots, \mathcal{C}.X_n\}$ = Scope($\phi$);
12     let $\psi$ be a factor such that Scope($\psi$) $= \{i.X_1, \cdots, i.X_n\}$;
13     foreach Value $\{\mathcal{C}.X_1 = x_1, \cdots, \mathcal{C}.X_n = x_n\}$ do
14         $\psi(i.X_1 = x_1, \cdots, i.X_n = x_n) = \phi(\mathcal{C}.X_1 = x_1, \cdots, \mathcal{C}.X_n = x_n)$;
15     $pool = pool \cup \{\psi\}$;
16 Compute an elimination order $t'$ over $\mathcal{A}_{out}(\mathcal{C})$;
17 Apply algorithm 1 using $t'$ over $pool$;
```

When there are mutually dependent instances, SVE cannot converge since it will endlessly iterates over mutually dependent instances. Such configurations can be encountered but we have found it particularly counter-intuitive to model such systems in the classic PRM framework. However, the notion of interface we introduced in chapter 4 can easily create configurations where mutually dependent instances appear. SVE's inability to deal with mutually dependent instances is due to the bottom-up elimination ordering. This led us to its abandon and its replacement is the topic of section 5.3. Thus, to prove algorithm 5 convergence we must limit its use to systems with no mutually dependent instances.

Theorem 5.1 (SVE convergences) *Let Π be a PRM and $\mathcal{S}$ a relational skeleton. In the absence of mutually dependent instances in $\mathcal{S}$, SVE will proceed with the elimination of all instances in $\mathcal{S}$.*

Proof. Algorithm 5 proceeds with a depth-first search in the relational skeleton, following slot chains and inverse slot chains. If there does not exist any set of mutually dependent instances in $\mathcal{S}$ we are guaranteed to find at least one leaf instance. Eliminating that leaf instance creates at least one other leaf instance and, by induction, we can eliminate all instances of the relational skeleton. If it is not the case, there is obviously a set of mutually dependent instances, which contradicts our assumption that there is none. ■

Now that we have shown that algorithm 5 converges, we will prove that the attribute elimination scheme of SVE matches the elimination performed by VE with a specific elimination order.

Theorem 5.2 (SVE correctness) *Let Π be a PRM, $\mathcal{S}$ a relational skeleton, a set of evidence e and a query $Q \in \mathcal{A}(q)$, with q an instance in $\mathcal{S}$. If SVE converges, then the returned factor encodes the probability $P(q|\mathrm{e})$.*

Proof. Without loss of generality, we will suppose that there is no evidence[2]. Let Π be a PRM, $\mathcal{S}$ a relational skeleton and $\{c_1, \cdots, c_n\}$ be a set of instances such that c_i is eliminated before c_j only if $i < j$ and we denote by $\mathcal{C}_i$ the class of instance c_i.

Let us consider c_1's elimination. Since c_1 is the first instance to be eliminated, it is necessarily a leaf node (theorem 5.1). This entails that either $\mathcal{C}_1$ has no output attribute or that none of $\mathcal{C}_1$ output attribute instantiations in c_1 have children outside of c_1. Then by applying equation 5.1 we eliminate inner attributes at class level and then if there are class level output attributes, we must eliminate them from the factors obtained through equation 5.1. Since, none of these output attributes have children outside of c_1, all information required for their elimination is encapsulated in c_1. Consequently c_1 is eliminated correctly and results in the creation of factors over $\mathcal{A}_{ext}(c_1)$ (supposing that $\mathcal{A}_{ext}(c_1) \neq \varnothing$).

Now let us consider the elimination of instance c_i. We first apply equation 5.1 to eliminate c_i's inner attributes at class level, i.e., we obtain a factor over c_i's output and external attributes. If c_i has no output attribute, then it has been eliminated and remains a factor over $\mathcal{A}_{ext}(c_i)$ (supposing that $\mathcal{A}_{ext}(c_i) \neq \varnothing$ and that the factor's scope is not limited to $\mathcal{A}_{ext}(c_i)$ but is a subset of $\bigcup_{1 \leqslant k \leqslant i} \mathcal{A}_{ext}(c_k)$). If $\mathcal{C}_i$ has output attributes but none of their instantiations have children outside of c_i, we apply the same reasoning used for c_1. Finally, in the case where c_i has attributes with children outside of it, the bottom-up elimination order ensures that all instances that had dependencies over c_i have been eliminated. Consequently, we can eliminate c_i's output attributes if we take into account the factors created by previous instance eliminations (as in VE, we assume that such factors are used only once and removed from the pool of factors after they have been used to create a new one). As a result, c_i is correctly eliminated and its elimination creates factors over $\mathcal{A}_{ext}(c_i)$ (assuming that $\mathcal{A}_{ext}(c_i) \neq \varnothing$ and that the factor's scope is not limited to $\mathcal{A}_{ext}(c_i)$).

When instance c_n is reached, all previous instances have been eliminated. Consequently, existing factor's scopes are only over attributes in c_n (and the queried attributes). Then, eliminating c_n can be done as explained previously, except that the resulting factors will be over the queried attribute.

Eliminating instances c_1 to c_n using SVE is equivalent to applying algorithm 1 over the ground BN of $\mathcal{S}$ using an elimination order induced by the partial ordering $\{c_1, \cdots, c_n\}$. Indeed, c_1's attributes elimination only requires attributes that are in c_1 and c_i's attributes elimination requires attributes that have been eliminated (theorem

[2]When confronted to evidence, we can replace instances with evidence by classes that directly encode the desired evidence.

5.1). Thus algorithm 5 is equivalent to algorithm 1 using a specific elimination ordering. ■

We will conclude this section by discussing algorithm 5 complexity. Obviously, algorithm 5 applies at most VE n times, where n is the number of instances in $\mathcal{S}$. Since VE complexity is exponential in the size of the largest clique, we can infer that algorithm SVE is of the same order of magnitude. However, VE elimination orders are defined using algorithms that are not constrained by the specific ordering induced by SVE's bottom-up elimination ordering. Thus, even if the theoretical complexity of both algorithms is identical, the quality of the elimination ordering induced by the bottom-up elimination of instances can considerably burden or favor SVE. However, estimating the quality of SVE's elimination ordering can be done easily: we simply need to compute the size of each factor after each attribute elimination. Doing so will give us the size of the largest factor created by SVE. We can then proceed with the same operation using VE and compare each elimination order's quality. If the size of the largest factor created by SVE is by order of magnitude bigger than VE's largest factor, then it would be preferable to use ground inference or the algorithm presented in the next section.

5.3 Structured Probabilistic Inference

We present here a new probabilistic inference algorithm called Structured Probabilistic Inference (SPI). SPI is a generalization of SVE in which we drop bottom-up elimination of instances for a more generic approach. We have already pointed out SVE's limits when dealing with mutually dependent instances, yet there are other issues, notably about SVE's performances on specific systems. We will first present these specific systems for which SVE is counter-performing and we will then present a solution well suited for them. We will then detail the SPI algorithm and provide its complexity analysis.

Limits of a bottom-up elimination

We have seen in section 5.2 that SVE is based on a bottom-up elimination of instances. Doing so offers the advantage of dealing recursively with instances elimination (the original SVE is recursive, unlike algorithm 5). However, we remarked that some systems, with mutually dependent instances, do not allow SVE to converge (this is also true with the original SVE). We will see that there is another important flaw in SVE regarding inference performances. Indeed, the bottom-up elimination order used by SVE enforces elimination orders of attributes that can be particularly counter-performing for a wide range of systems.

Figure 5.5 illustrates a simple system for which SVE is remarkably not well suited. In figure 5.5 all instances only contain one attribute, thus we represent instances and attributes with only one graphical element. We can immediately notice that figure 5.5 is a polytree, consequently inference should be easy. Let us consider how SVE will

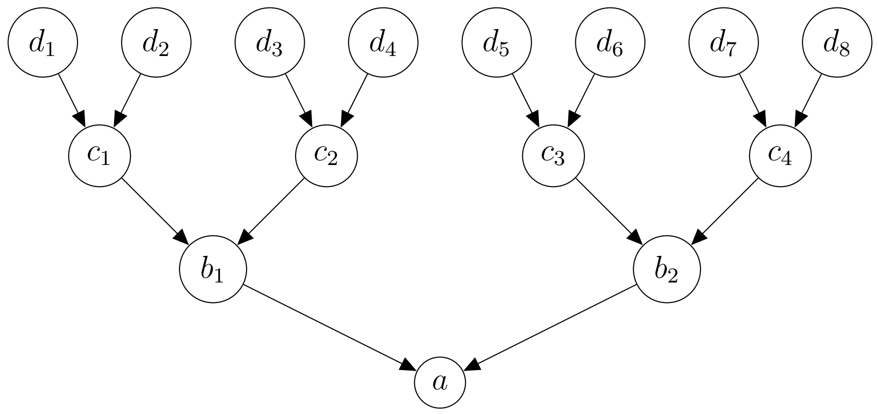


Figure 5.5: A worse case scenario for SVE: each node is the only attribute of its instance (instances are not illustrated for the sake of clarity).

proceed with the system of figure 5.5. Let us suppose that the query is c_4 and let us focus on the factors created after instances are eliminated:

Instance	Factors in the pool after the instance elimination
a	$\phi_1(b_1, b_2)$
b_1	$\phi_2(c_1, c_2, b_2)$
c_1	$\phi_3(d_1, d_2, c_2, b_2)$
d_1	$\phi_3(d_2, c_2, b_2)$

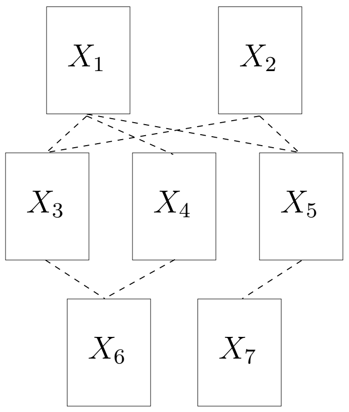
Note that even changing the order in which leaf instances are eliminated does not improve the quality of SVE elimination of attributes:

Instance	Factors in the pool after the instance elimination
a	$\phi_1(b_1, b_2)$
b_1	$\phi_2(c_1, c_2, b_2)$
b_2	$\phi_3(c_1, c_2, c_3, c_4)$
c_1	$\phi_3(d_1, d_2, c_2, c_3, c_4)$

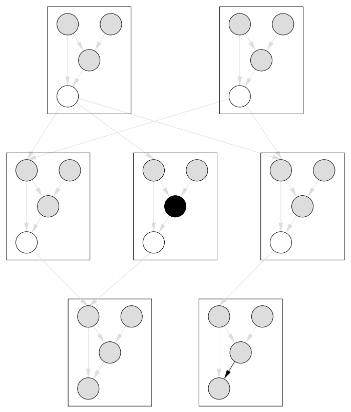
Globally, when we are confronted with an inverse pyramidal topology, i.e., many parents and few children, SVE performs badly. This is crucial as we are often confronted with aggregators having many parents. The bottom-up elimination of instances forces SVE to eliminate aggregators before their parents, creating large factors that would have been prevented with other elimination orders.

A generic scheme for structural inference

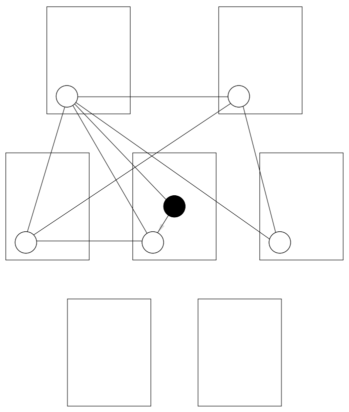
Our proposition to enhance inference in PRMs while exploiting hierarchical and structural inference is to proceed with a two steps inference. The first step of our algorithm eliminates inner attributes from all instances, reusing as many times as possible class level eliminations. The second step generates the Markov Network (MN) induced by the factors obtained after eliminating inner attributes. We then apply VE or SS over the newly created MN.



(a) The system used in figure 5.4.



(b) Gray nodes are eliminated attributes.



(c) The MN induced by the factors resulting inner attributes elimination.

Figure 5.6: If we eliminated inner attributes from all instances, we can create a MN over the remaining factors.

Figure 5.6 illustrates the two steps with the same system of figure 5.4 in which we eliminated all inner attributes (figure 5.6b). The factors created by the elimination of all inner attributes can be used to create the MN illustrated in figure 5.6c. SPI fixes several of SVE flaws. First of all, dropping the bottom-up elimination order allows more optimal elimination orders. Secondly, SPI is not bothered by mutually dependent instances, as there is no recursive elimination of instances. SPI offers a generic framework for exploiting structural information as the second phase of the algorithm can use any probabilistic inference for MNs such as junction tree inference algorithms or approximate inference algorithms.

Algorithm 6 requires a PRM Π, a system $\mathcal{S}$, a query and a set of evidence $\mathbf{e}$. It returns a MN constructed from the factors obtained after eliminating inner attributes. Algorithm 6 processes each instance one by one and either: (i) eliminates inner attributes at instance level, except of the queried attribute if the instance is the query (lines 2-3); (ii) eliminates inner attributes at instance level and takes into account evidence over the inner attributes (lines 4-5); (iii) proceeds with a class level elimination and reuses if possible previous eliminations (lines 6-7). The final step of algorithm 6 is to create the induced MN from the factors in *pool* (see chapter 2.2).

Algorithm 6: Structured Probabilistic Inference

```
   Input: PRM $\Pi$, system $\mathcal{S}$, a query $q.Q$, a set of evidence $\mathbf{e}$
   Output: a MN
 1 let $pool = \varnothing$ be an empty set of factors;
 2 foreach instance $i$ in $S$ do
 3     if $i$ equals $q$ then
 4         $VE(q, Q,\ \mathbf{e},\ pool)$;
 5     else if an inner attribute of $i$ has evidence then
 6         $VE(i,\ \mathbf{e},\ pool)$;
 7     else
 8         $VE(Class(i),\ i,\ pool)$;
 9 Create the induced MN $\mathcal{H}$ over the factors in $pool$;
10 return $\mathcal{H}$;
```

Theorem 5.3 *Let* Π *be a PRM and* $\mathcal{S}$ *a system. The MN returned by algorithm 6 matches the induced MN obtained by applying VE over the ground BN of* $\mathcal{S}$ *after eliminating ground inner attributes.*

Proof of theorem 5.3 is trivial and is omitted. Theorem 5.3 is useful as it tells us that SPI is in fact VE applied on a subset of the nodes in the ground BN of $\mathcal{S}$. The only difference with VE is that SPI prevents redundant computations by caching class level eliminations.

5.4 Structured BayesBall

As stated by Shachter (1998), d-separation analysis (with respect to structure and evidence) may induce another kind of optimization: only a subset of the BN may be necessary to answer a specific request. D-separation exploits the graphical properties of BNs to prune irrelevant nodes with respect to a query and evidence. A classic example is to see nodes as valves that block or let the flow of information pass as shown in figure 5.7 (Pearl, 1988).

Figure 5.7 illustrates how hard evidence influences the flow of information in BNs. We can see that chains ($X \rightarrow Y \rightarrow Z$) and divergent arcs ($X \leftarrow Y \rightarrow Z$) are both blocked by hard evidence. However, v-structures ($X \rightarrow Y \leftarrow Z$) do not block information when there is evidence and block it if not. The reasons of these behaviors have been detailed in chapter 2.1. The BB algorithm (Shachter, 1998) exploits d-separation to determine the set of requisite nodes given a query and evidence. It uses an imaginary ball that bounces the BN's nodes, following the flow of information. In this section, our aim is to adapt such algorithm to PRMs. A d-separation analysis will help reducing the number of computations, which can be seen as a low level exploitation of the structural information encoded in PRMs.

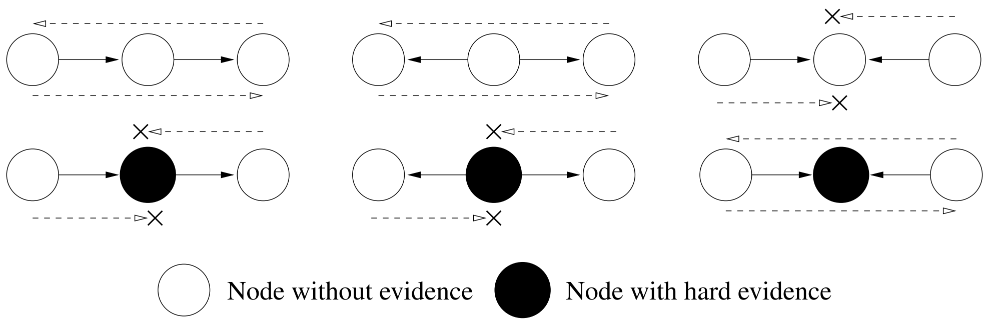
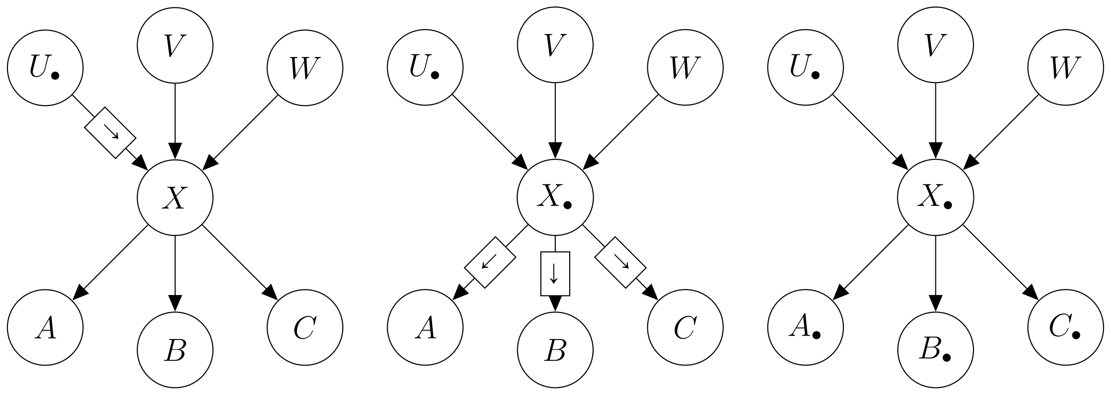


Figure 5.7: The different influences of hard evidence on BNs.

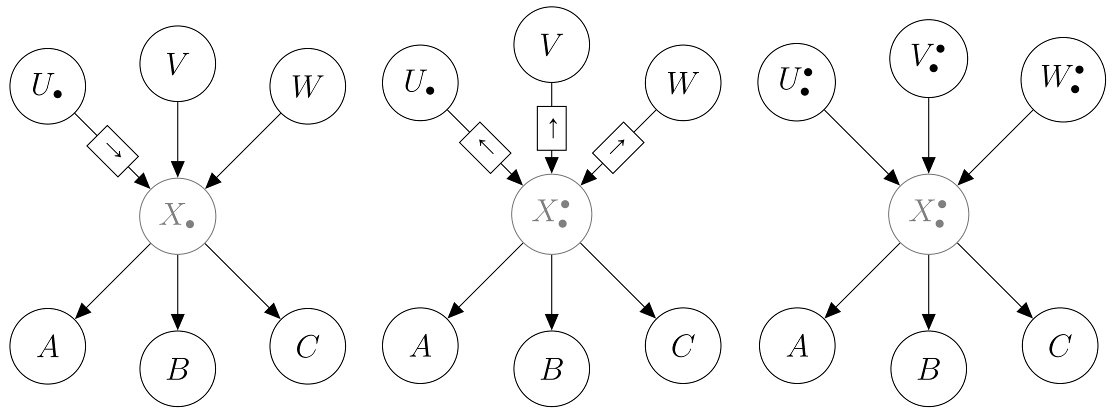
The BB algorithm marks BN's nodes on the top and the bottom. The marks purposes are twofold: they define the set of required nodes (nodes with the top marked) and how a node have been reached (marked on the top when reached by a child and marked on the bottom when reached by a parent). We know that barren nodes are unobserved leaves in the BN's DAG. We also know that when all the children of a node are barren, then that node is also barren. The BB algorithm exploits this fact to find the set of barren nodes in a BN: starting from the query, it parses the BN's DAG in all directions (the parsing is illustrated by a ball, giving the algorithm's name). The different situations are illustrated in figures 5.8, 5.9, 5.10 and 5.11. In the following examples, we denote by X the node that receives the ball either from a parent (figures 5.8 and 5.9) or from a child (figures 5.10 and 5.11).



(a) U received the ball from a parent and sends it to X.
(b) X is not observed: he forwards the ball to its children and marks itself on the bottom.
(c) X and its children are marked on the bottom: they are not required nodes.

Figure 5.8: In the absence of evidence, the ball is forwarded to children until an observed node or a leaf node is reached.

Figures 5.8 and 5.9 illustrate how the ball is forwarded after it was received by a parent. How the marking works is important to understand the algorithm we will present in this section: a node is marked on the top when it is required. In most cases, this happens when the node received the ball from a children. Indeed, this occurs only



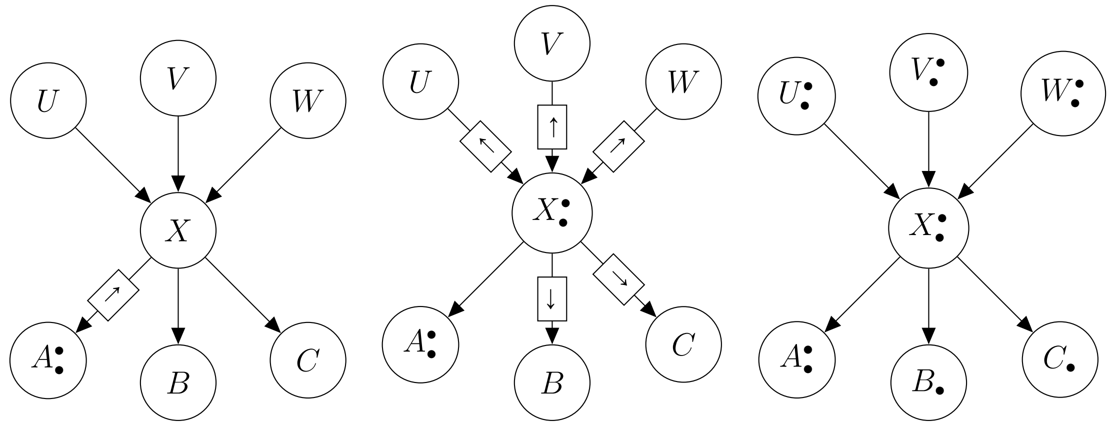
(a) U received the ball from a parent and forwards it to X.

(b) X is observed: he forwards the ball to his parents but not to his children.

(c) X and his parents are required nodes.

Figure 5.9: An observed node *connects* its parents, this is illustrated by the ball being forwarded to all of X's parents. When the ball is received from a child, it entails that the child have some relevant information with respect to the query. Consequently, the parents of such nodes are required as well.

when a child has relevant information with respect to the query. Nodes are also marked on the top when they are observed and received the ball from a parent, since in such situations the evidence will be relevant given the query.



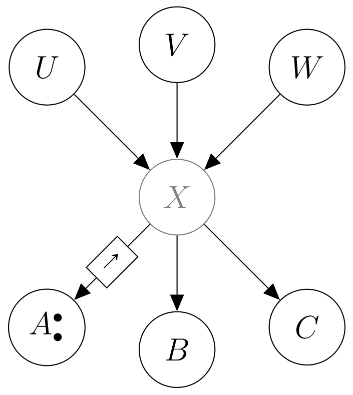
(a) A received the ball from a child and sends it to X.

(b) X is not observed: he forwards the ball to his neighbors and become a required node.

(c) Parents of X are marked as required nodes and X's descendant are visited.

Figure 5.10: When a node is unobserved and receives the ball from a child, it becomes a required node. Consequently, its parents must also be added (if they are unobserved) and its children must be visited to find relevant information, i.e., evidence.

Figures 5.10 and 5.11 illustrate how the ball is forwarded after it was received by a


(a) A received the ball from of its children and forwards it to X.

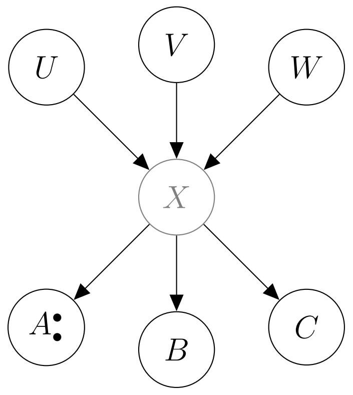

(b) Since X is observed: he ignores the ball received from A.

Figure 5.11: When observed, a node d-separates its children from its parents. Here, node X is observed and ignore the ball when it is received by one of its children.

child. When X is unobserved the rule is simple: X is marked both on top and bottom and sends the ball to his neighbors. Since its parents will received the ball from a child, they will also be marked doubly and forward the ball in a similar manner. Children on the other side will be required only if they are observed, indeed if unobserved they would be barren nodes. To summarize the purpose of marks, we must remember that nodes marked on the top are required and nodes marked on the bottom have been visited but are not (yet) required.

Now that we have explained how the BB algorithm works, we will discuss how d-separation analysis and structural inference are not incompatible inference optimizations. Indeed, structural inference uses repetitions to prevent redundant computation and d-separation analysis extracts the minimal set of nodes required to answer a query with respect to a set of evidence. Both approaches enter in conflict since d-separation breaks the structure used by structured inference. Fortunately, there is a trade-off between both approaches.

Figure 5.12 illustrates an experiment in which we randomly observed different numbers of attributes in the same system. We used a system generated from the power surge example with fifty rooms, ten printers and forty computers (for a total of 6501 nodes). Queries were randomly generated and each point is the result of several hundred of runs. The curve of figure 5.12 illustrates how much d-separation analysis is sensitive to evidence and queries (we did not allow configurations where the query was observed). The curve is uneven because we voluntarily chose to not run enough experiments to even it. Doing so points out the high variance of the size of the set of requisite nodes with respect to the query and the evidence. What we must learn from figure 5.12 is that there are cases in which repetition and evidence occur. We will present a scheme to detect and exploit such repetitions, while exploiting d-separation analysis to prune irrelevant attributes.

To exploit d-separation and structural information we must first consider how in-

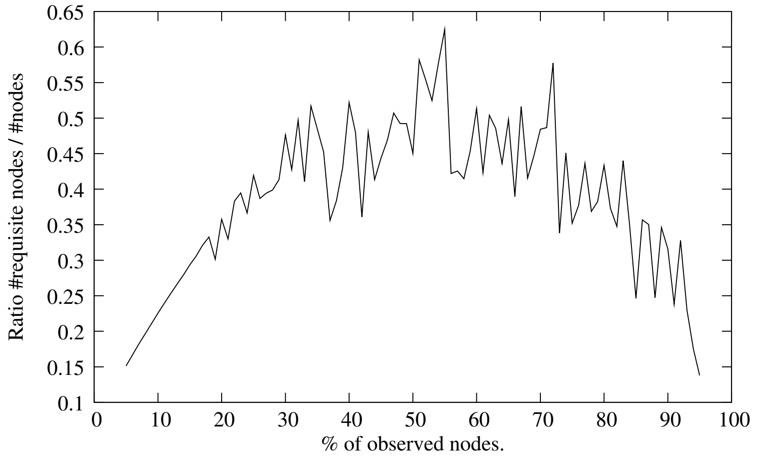
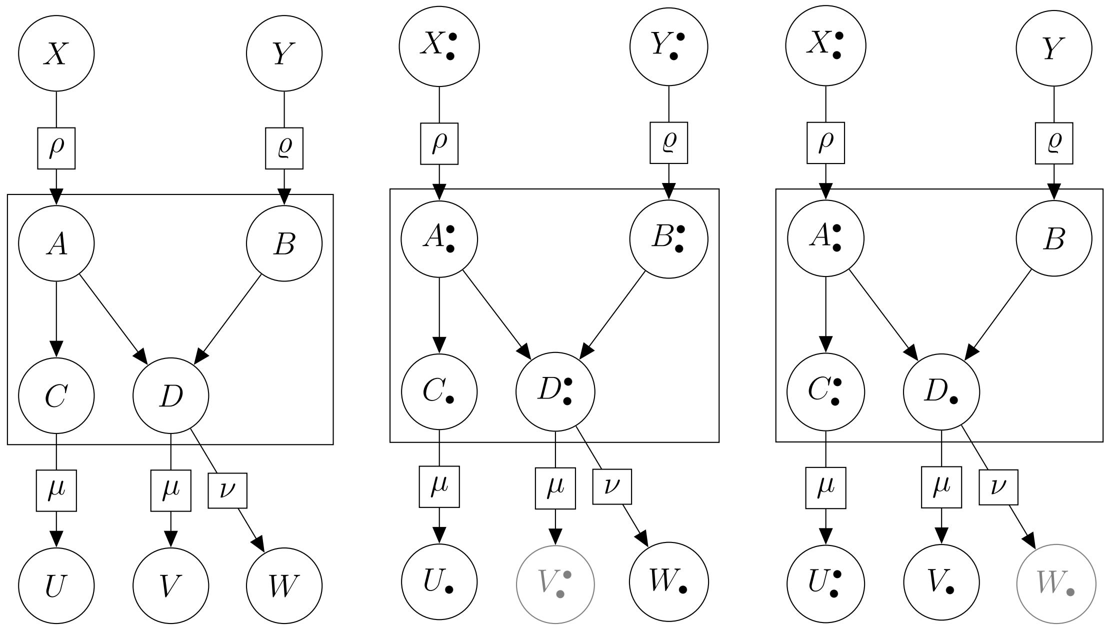


Figure 5.12: D-separation analysis with BB.

formation flows between instances in a system. We know that attributes of different instances are connected through slot chains (inverse or not). Then, we can consider each slot chain as an indicator of active paths between instances.



(a) Slot chains and inverse slot chains are used to define a class' dependencies.

(b) The ball is sent from X and V (or one of his descendant) is observed. Here, attributes A, B, and D are required.

(c) The ball is sent by X and U (or one of his descendants) is observed. Here, attributes A and C are required.

Figure 5.13: A class with attributes A, B, C, D and its dependencies: square labels are slot chains (ρ, ϱ) or inverse slot chains (μ, ν). The other attributes belong to unrepresented classes in figure 5.13a and instances in figures 5.13b and 5.13c.

Figure 5.13 illustrates how the set of required attributes can differ depending on the instance's context. In figures 5.13b and 5.13c two instances of the class represented in figure 5.13a are placed in different contexts during the BB algorithm (supposing we have such algorithm for PRMs). Both instances receive the ball from X but in figure 5.13b V (or one of his descendants) is observed and in figure 5.13c U (or one of his descendants) is observed. We can see that both instances required attribute sets differ, thus if we want to use d-separation analysis and structured inference we must dissociate instances with different set of required attributes. From there, adapting BB to PRMs is trivial.

Algorithm 7: Structured BayesBall

```
   Input: a PRM $\Pi$, a relational skeleton $\mathcal{S}$, a query $q.Q$, a set of evidence $\mathbf{e}$
   Output: a set of required attributes
1  Add $q.Q$ to be visited from a child;
2  while there is an attribute $X$ to be visited do
3      if $X$ is visited from a parent and is not marked on the bottom then
4          Mark the bottom of $X$;
5          if $X$ is observed then
6              Mark the top of $X$;
7              foreach $Y \in \pi(X)$ do
8                  Add $Y$ to be visited from a child;
9          else
10             foreach $Y \in Ch(X)$ do
11                 Add $Y$ to be visited from a parent;
12     else if $X$ is visited from a child and is not marked on the top then
13         if $X$ is not observed then
14             Mark the top and bottom of $X$;
15             foreach $Y \in \pi(X)$ do
16                 Add $Y$ to be visited from a child;
17             foreach $Y \in Ch(X)$ do
18                 Add $X$ to be visited from a parent;
19 Return the set of attributes marked on the top;
```

Algorithm 7 is almost identical to the BB algorithm. Algorithm 7 takes as inputs a PRM, a relational skeleton a query (q is an instance and Q an attribute) and a set of evidence. Algorithm 7 supposes that $\pi(X)$ and $\mathrm{Ch}(X)$ return respectively the parents and children that are in X's instance and in other instances. We could choose to make the slot chains and inverse slot chains part of the algorithm's specification but it adds unnecessary complexity. Algorithm 7 parses the relation skeleton from attribute to attribute, updating the markings each time it is necessary. Complexity analysis and convergence proof are identical to the BN version of this algorithm and can be found in Shachter

(1998).

5.5 Experimental results

PRMs are a difficult framework for experimenting as there are no standard system to compare probabilistic inference. Furthermore generating random PRMs is difficult since there is a considerable amount of parameters: class, inheritance, reference slots, reference chains, etc. To experiment our inference algorithms we chose to exploit the power surge example described in figure 4.3. For the following experiments we have used different systems generated from the power surge example (see figure 4.3. The examples we used for our experiments are composed of a fixed number of rooms (fifty) and a random number of printers and computers per room. Despite the simplicity of the power surge example, we can generate challenging systems. Furthermore, each experiment requires a small set of parameters, making them more understandable and help understand our algorithms behavior. One parameter is the number of computers per room, that have an influence on the size of the network. Another parameter is the number of printers, which are connected to all the computers in their rooms, that have a direct impact on the size of the largest clique.

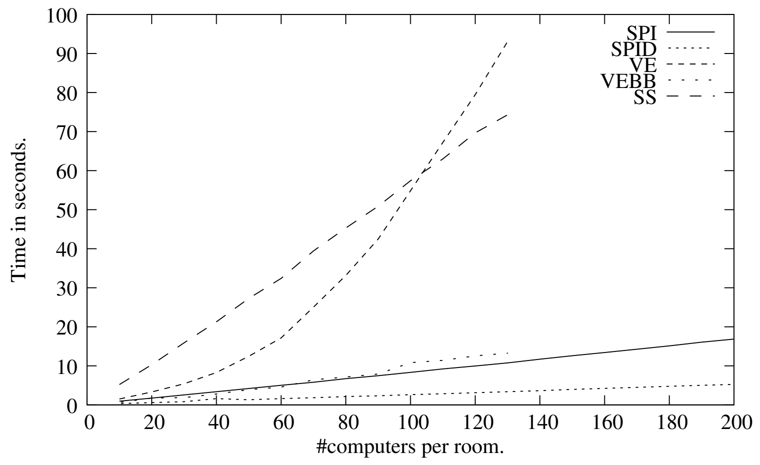


Figure 5.14: Structured and ground inference.

Figure 5.14 exhibits the behavior of several inference algorithms in the absence of observations. For this experiment we have raised the number of computers per room from 10 to 200, with 6 printers and 40 rooms (for a total of 1 441 to 24 241 nodes). The curves which stops after 130 computers represent classic inference algorithms used on ground BNs. The following ground inference algorithms were used: VE, SS (SS) and VE coupled with BB (VEBB). We used two versions of structured inference: SPI (SPI) and SPI coupled with Structured BayesBall (SPID). The results show the limitation of reasoning on BNs, since the computer used for the tests could not handle the size of the ground BNs. Furthermore SPI and SPID give good results.

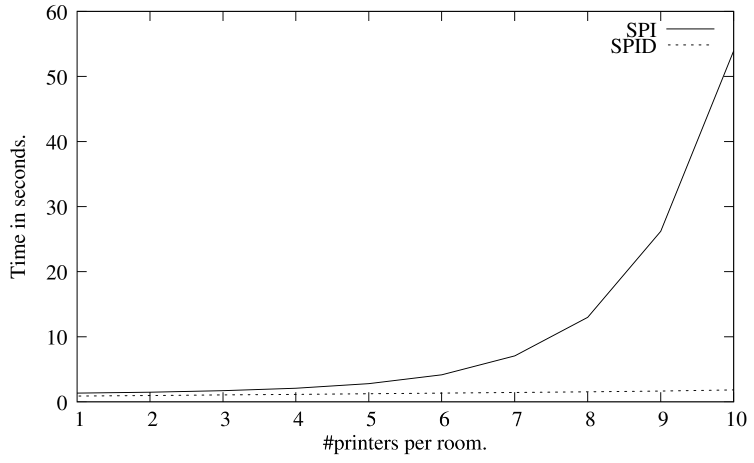


Figure 5.15: Comparaison between SPI and SPID on networks with high tree-width.

Figure 5.15 shows the behavior of SPI and SPID in the absence of evidence. The system contained 40 rooms with 50 computers and the number of printers varied from 1 to 10 (for a total of 6 041 to 6 401 nodes). The curves clearly show the possible gain d-separation analysis can give to inference. However we must point out that the example used for these test gives very good performance to algorithms exploiting d-separation and they should be considered as best case situations. Although in the worst case situation, it is easy to show that SPI and SPID differ neither in complexity nor in time.

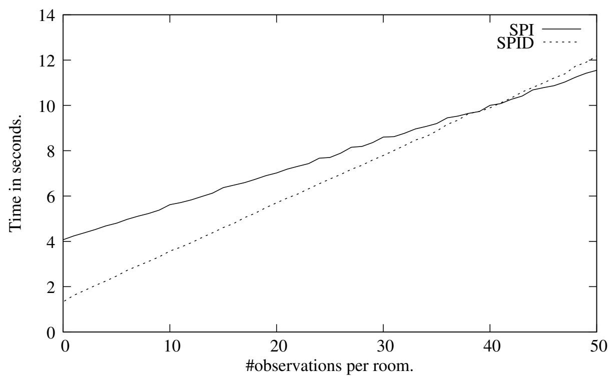


Figure 5.16: Inference performance under evidence (6 printers per room).

Figure 5.16 shows the impact of observations on SPID and SPI in a system with 6 printers, 50 computers per room and 40 rooms. It gives the inference time when the power supply's state attribute is queried under different amounts of evidence. The number of observed attributes in each room grows from 0 observed computers to 50. These results give us the insights of SPID performances under heavy observations: we can expect slower inference, even slower then SPI in the worst case. The worst case for SPID would be systems with small cliques.

Figure 5.17 illustrates an experiment slightly different from the one represented in

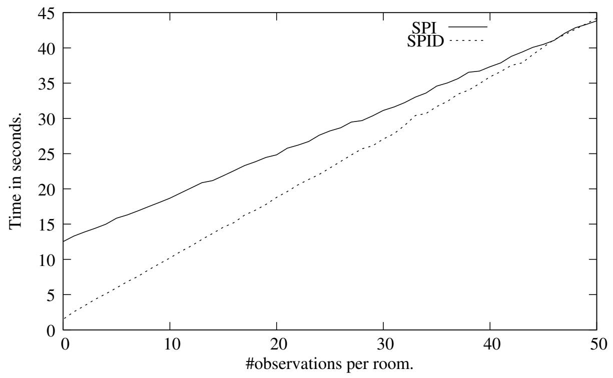


Figure 5.17: Inference performance under evidence (8 printers per room).

figure 5.16. The difference lies in the number of printers per room, which was raised to 8. This illustrates the importance of tree-width in inference complexity: in figure 5.16 the tree-width equaled 129 and in figure 5.17 257. We can see that the cost of d-separation analysis is quickly negligible when inferring in systems with large cliques.

Chapter 6

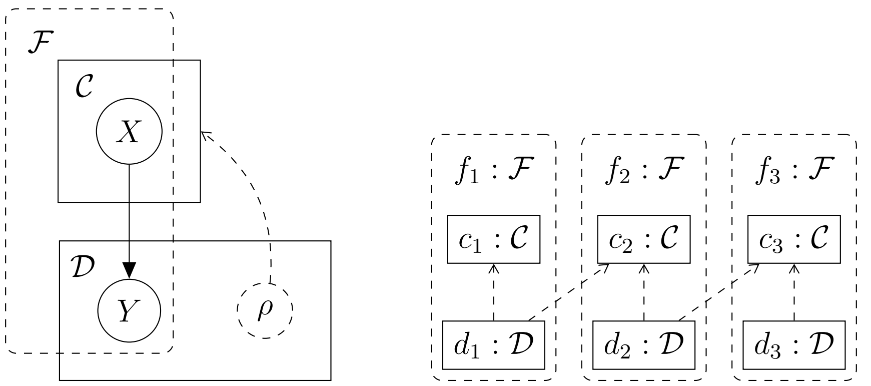
Pattern Discovery for Structured Probabilistic Inference

In this chapter, we propose an enhancement of SPI based on discovering repeated patterns of instances that are used to speed-up structured inference. To understand why such patterns emerge in complex systems modeled using PRMs, we must consider abstraction. Abstraction is one of the most important feature in the object-oriented paradigm: it is used to decompose knowledge into classes. The more complex the domain, the more classes necessary to lower its complexity. Indeed, it is easier to model and reuse classes made of few attributes, e.g., less than ten and one or two reference slots, than classes composed of hundred of attributes and tens of reference slots. This decomposition of knowledge leads to a large number of instances and since experts will naturally minimize the number of attributes[1] the benefits of using structured inference can be limited. However, in many real world applications, instances often form repeated patterns throughout the system. By using a frequent subgraph pattern mining algorithm, it is possible to discover such patterns and exploit them to speed-up SPI. Frequent subgraph pattern mining is related to the subgraph isomorphism problem, which is a NP-complete problem (Cook, 1971). Consequently, mining optimally such patterns is hard. In this chapter, we both provide a structured inference algorithm for PRMs exploiting patterns and a mining heuristic fast enough for efficient on-line inference.

Marginalizing-out internal attributes at class level is the key to structured inference efficiency as it reduces significantly redundant computations. However, not all redundancies can be identified by this scheme. Figure 6.1 illustrates the idea of finding patterns to infer new classes and use them to reduce the number of output attributes in the relational skeleton. In figure 6.1 $\mathcal{C}$ and $\mathcal{D}$ are two classes with $X \in \mathcal{A}(\mathcal{C})$ and $Y \in \mathcal{A}(\mathcal{D})$ two attributes such that $\mathcal{C}.X$ is the parent of $\mathcal{D}.Y$ and $\mathcal{D}.Y$ is the only child of $\mathcal{C}.X$ not in $\mathcal{A}(\mathcal{C})$. In this configuration, $\mathcal{C}.X$ is an output attribute and $\mathcal{D}.Y$ is an inner attribute. Consequently, we will be able to eliminate $\mathcal{D}.Y$ at class level but not $\mathcal{C}.X$. However, if we consider a *new* class, name it $\mathcal{F}$, defined by the union of $\mathcal{C}$ and $\mathcal{D}$, attribute $\mathcal{F}.X$ is no longer an output attribute since $\mathcal{F}.Y \in \mathcal{A}(\mathcal{F})$. Hence, for any pair of instances (c, d)

[1]We have been confronted to systems where each class contained a single attribute.

of classes $\mathcal{C}$ and $\mathcal{D}$ that match the definition of $\mathcal{F}$, we can replace them by an instance of $\mathcal{F}$ in which $\mathcal{F}.X$ is an inner attribute, thus eligible to class level elimination.



(a) Classes $\mathcal{C}$ and $\mathcal{D}$ can be grouped into a *dynamic* class $\mathcal{F}$.

(b) Patterns matching $\mathcal{F}$ can be replaced by instances of $\mathcal{F}$.

Figure 6.1: We can search for repeated patterns of instances in the relational skeleton, doing so enables class level elimination for output attributes encapsulated by patterns. Here, attribute $c_1.X$ and can be eliminated at class level if we consider (c_1, d_1) as an instance of $\mathcal{F}$.

Note however that not all pairs (c, d) can be used: in figure 6.1, pairs (c_1, d_1) cannot be used conjointly with (c_2, d_1). Indeed, if we replace (c_1, d_1) by an instance of $\mathcal{F}$ we cannot replace (c_2, d_1) by another instance of $\mathcal{F}$ without changing the joint probability distribution. Another important notion is the efficiency of a pattern. In figure 6.1a we can see that pattern (c, d) has a total of five possible matches, but at most three of them can be used at the same time. If we substitute (c_1, d_1), (c_2, d_2) and (c_3, d_3) with instances f_1, f_2 and f_3 of $\mathcal{F}$ see that $f_1.X$ is an inner attribute, but $f_2.X$ and $f_3.X$ are not. Furthermore attribute $f_3.Y$ does not have the same number of parents as attributes $f_1.Y$ and $f_2.Y$. Fortunately, we will see that by labeling the relational skeleton we can build a graph in which discriminating different kinds of instances of a same class becomes trivial.

This chapter is organized as follow: in section 6.1 we formalize our problem of repeated sets of instances in a relational skeleton and in section 6.2 we provide a complexity analysis of this problem. Will see that finding optimal patterns in a relational skeleton is a NP-hard problem. In section 6.3 we present an approximate algorithm for mining patterns and we detail the pruning rule we used to find good quality patterns. Experiments reported in section 6.4 show the practical efficiency of our approach.

6.1 Formalizing pattern discovery

When searching for patterns, we need to consider both output attributes and aggregators. Doing so ensures that each set of instances matching the same pattern can be encapsulated by the same class. Such instances are called interchangeable because apart from their names, they are identical in all points.

Definition 6.1 (Interchangeable instances) *Let Π be a PRM and $\mathcal{S}$ a relational skeleton. Two instances c_i and c_j of class $\mathcal{C}$ are said to be interchangeable in $\mathcal{S}$ if for all aggregators $X \in \mathcal{A}(\mathcal{C})$ we have $|\pi(c_i.X)| = |\pi(c_j.X)|$ and for all output attributes $Y \in \mathcal{A}(\mathcal{C})$ we have $|Ch(c_i.Y)| = |Ch(c_j.Y)|$.*

For example, in figure 6.1a, instances c_2 and c_3 are interchangeable, so are instances d_1 and d_2. Detecting such variations between instances is necessary to discover patterns. To do so, we will use a labeled graph called a boundary graph.

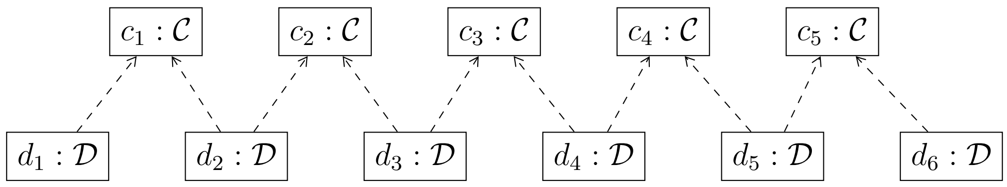
Definition 6.2 (boundary graph) *Let Π be a PRM and $\mathcal{S}$ a relational skeleton. A boundary graph is an undirected graph $\mathcal{BG} = (\mathcal{I}, \mathcal{E})$, where:*

- *$\mathcal{I}$ is a set of labeled nodes representing instances such that two nodes c and d are labeled identically if and only if c and d are interchangeable in $\mathcal{S}$;*
- *$\mathcal{E} \subseteq \mathcal{I} \times \mathcal{I}$ is a set of labeled edges such that $\exists (c, d) \in \mathcal{E}$ if and only if*

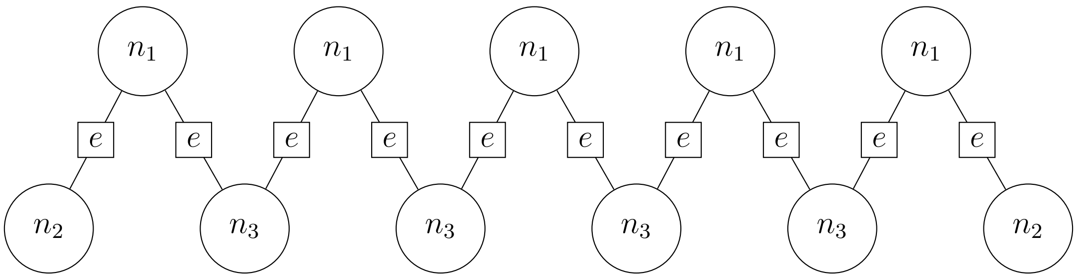
$$\mathbf{L}_{cd} = \big(\mathcal{A}_{out}(c) \cap \mathcal{A}_{ext}(d)\big) \bigcup \big(\mathcal{A}_{out}(d) \cap \mathcal{A}_{ext}(c)\big) \neq \emptyset,$$

  where $\mathcal{A}_{ext}(c)$ (respectively $\mathcal{A}_{ext}(d)$) is the set of external attributes of c (respectively d), $\mathcal{A}_{out}(c)$ (respectively $\mathcal{A}_{out}(d)$) the set of output attributes of c and $\mathbf{L}_{cd}$ is used as the label of edge (c, d) in $\mathcal{BG}$.

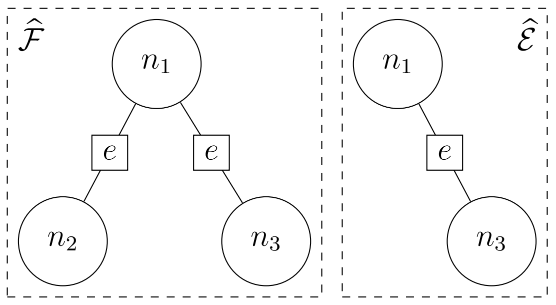
Ideally, an edge (c, d) of the boundary graph and its label define precisely what attributes that were eliminated at instance level in c and d, should be eliminated at class level, if (c, d) were considered as an instance of the fusion of $\mathcal{C}$ and $\mathcal{D}$. Unfortunately, it is not that simple. Let us illustrate this notion with an example: figure 6.2b illustrates the boundary graph of figure 6.2a in which three different sorts of instances are represented. Label n_1 represents instances of class $\mathcal{C}$, label n_2 instances of class $\mathcal{D}$ where instantiations of $\mathcal{D}.Y$ have one parent and label n_3 instances of class $\mathcal{D}$ where instantiations of $\mathcal{D}.Y$ have two parents. Edges are labeled such that two edges with identical labels encode the exact same set of dependencies. In this example, there are necessarily dependencies between some instantiation of $\mathcal{C}.X$ and some instantiation of $\mathcal{D}.Y$. In figure 6.2a, each edge represents the fact that an instantiation of $\mathcal{C}.X$ is shared by an instance of $\mathcal{C}$ and two instances of $\mathcal{D}$. Consequently all edges are labeled identically with label e. However, in the general case we can expect instances to have different dependencies given their context. In figure 6.2b this is represented by the two different labels given to instances of class $\mathcal{D}$. Instances of $\mathcal{D}$ labeled n_3 differ greatly from the



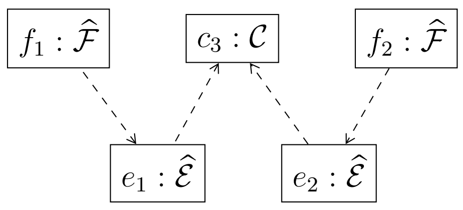
(a) A relational skeleton in which several patterns can be found.



(b) The boundary of figure 6.2a. Circles are the boundary graph nodes and squares are edges labels.



(c) Two existing patterns in figure 6.2b represented as dynamic classes.


$f_1 : \widehat{\mathcal{F}}$
$c_3 : \mathcal{C}$
$f_2 : \widehat{\mathcal{F}}$
$e_1 : \widehat{\mathcal{E}}$
$e_2 : \widehat{\mathcal{E}}$

(d) Using instances of the dynamic class in figure 6.2c on the relational skeleton of figure 6.2a.

Figure 6.2: Finding patterns in a boundary graph is more explicit than finding in a relational skeleton.

ones labeled n_2 as they have different conditional probability distributions (the number of parents of instantiations of $\mathcal{D}.Y$ varies).

As said in this chapter's introduction, our objective is to reduce the number of output attributes using patterns of instances. In figure 6.2a, only instantiations of $\mathcal{C}.X$ are output attributes and we can see that only two kinds of patterns transform it into inner attributes: pattern $n_2 - n_1 - n_3$ and pattern $n_3 - n_1 - n_3$. However, these two patterns are incompatible: if we apply the first one first, there is only a single possible occurrence left for the second and if we choose the second one, there will be no possible occurrence for the first. In figure 6.2c we choose to use the first pattern, denoted $\widehat{\mathcal{F}}$, and another one over $n_1 - n_3$ denoted $\widehat{\mathcal{E}}$. Figure 6.2d illustrates the relation skeleton obtained after substituting instances of $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{E}}$ in figure 6.2a. Such classes are called *dynamic classes*.

Definition 6.3 (dynamic class) *Let $\mathcal{BG}$ be a boundary graph. A dynamic class $\widehat{\mathcal{D}}$ in $\mathcal{BG}$ is a pair $(G_{\widehat{\mathcal{D}}}, \mathcal{D})$ where $G_{\widehat{\mathcal{D}}}$ is a subgraph of $\mathcal{BG}$ and $\mathcal{D}$ the class induced by $G_{\widehat{\mathcal{D}}}$. We say that $\widehat{\mathcal{D}}$ is* composed *of classes (one for each instance in $G_{\widehat{\mathcal{D}}}$) that are called the components of $\widehat{\mathcal{D}}$.*

By abuse of notation we will often refer to $\widehat{\mathcal{D}}$ as if it was the class $\mathcal{D}$ induced by $G_{\widehat{\mathcal{D}}}$. What is interesting about dynamic classes is to compare its output attributes $\mathcal{A}_{out}(\widehat{\mathcal{D}})$ with the unions of its components output attribute. Let us denote by $\boldsymbol{\mathcal{C}}$ the set of components of $\widehat{\mathcal{D}}$. If $|\mathcal{A}_{out}(\widehat{\mathcal{D}})| < |\bigcup_{\mathcal{C} \in \boldsymbol{\mathcal{C}}} \mathcal{A}_{out}(\mathcal{C})|$ then $\widehat{\mathcal{D}}$ as fewer output attributes than the unions of its components, i.e., it should offer better performance under SPI. However, we must also consider external attributes, as eliminating inner attributes results in a factor over the union of external and output attributes. Note that the set of output attributes of $\widehat{\mathcal{D}}$ is different from the set of output attributes of a classic class, because any instance of $\widehat{\mathcal{D}}$ is guaranteed to have the same set of output attributes than its dynamic class. Ideed, given definitions 6.1 and 6.2 two instances are labelled identically if and only if they are interchangeable, i.e., only if they have the same set of equivalent external and output attributes. So, to improve structured inference, we shall search the boundary graph for frequent subgraphs, i.e., subgraphs repeated many times, create their corresponding dynamic class, substitute each subgraph by one instance of its dynamic class and, finally, apply an inference algorithm like SPI. Thus we need to characterize the set of instances matching a dynamic class.

Definition 6.4 (Matches of a dynamic class) *Let Π be a PRM, $\mathcal{S}$ a relational skeleton, $\mathcal{BG}$ the boundary of $\mathcal{S}$ and $\widehat{\mathcal{D}}$ a dynamic class. The matches $\mathrm{M}(\widehat{\mathcal{D}})$ of $\widehat{\mathcal{D}}$ is a multiset of instances such that each set $s \in \mathrm{M}(\widehat{\mathcal{D}})$ is a set of instances matching $\widehat{\mathcal{D}}$, i.e., that each set of instances in s can be replaced by an instance of $\widehat{\mathcal{D}}$ in $\mathcal{S}$.*

For example, in figure 6.2, we can see that $\widehat{\mathcal{F}}$ has the matches $\mathrm{M}(\widehat{\mathcal{F}}) = \{\{c_1, d_1, d_2\}, \{c_5, d_5, d_6\}\}$ and matches for $\widehat{\mathcal{E}}$ are $\mathrm{M}(\widehat{\mathcal{E}}) = \{\{c_1, d_2\}, \{c_2, d_2\}, \{c_2, d_3\}, \{c_3, d_3\}, \{c_3, d_4\}, \{c_4, d_4\}, \{c_4, d_5\}, \{c_5, d_4\}\}$. However, as we said, replacing matches by instances of their dynamic classes must be performed carefully: it may actually happen that occurrences of frequent subgraphs share some nodes. In this case, only one of them can be substituted, otherwise some instances of the *original* system would be counted several times. For example, in figure 6.2 we can see that many matches of $\widehat{\mathcal{E}}$ share common instances. Hence the following rule and definition:

Rule 6.1 *In the boundary graph, substituted subgraphs cannot share any node, i.e., any instance in the relational skeleton.*

Definition 6.5 (Substitutions of a dynamic class) *Let Π be a PRM, $\mathcal{S}$ a relational skeleton, $\mathcal{BG}$ the boundary of $\mathcal{S}$, $\widehat{\mathcal{D}}$ a dynamic class and $\mathrm{M}(\widehat{\mathcal{D}})$ its matches. A substitution $\mathrm{S}(\widehat{\mathcal{D}})$ is a subset of $\mathrm{M}(\widehat{\mathcal{D}})$ such that $\mathrm{S}(\widehat{\mathcal{D}})$ respects Rule 6.1.*

6.2 Complexity of pattern discovery

Optimizing structured inference thus amounts to searching for the *best* set of dynamic classes and subgraph substitutions satisfying Rule 6.1. Unfortunately, as shown in the following proposition, this problem is NP-hard:

Proposition 6.1 *The following problem is NP-hard:*
Instance: *a PRM, a boundary graph, an integer $K \geqslant 0$.*
Question: *is there a set of dynamic classes and boundary subgraph substitutions of these classes such that the number of operations (multiplications and summations) performed by structured inference is smaller than K?*

Proof of Proposition 6.1:
Reduction from Vertex Cover for cubic graphs, which is known to be NP-complete (Garey et al., 1974):
Instance: A graph $G = (V, E)$, $V = \{v_1, ..., v_n\}$, $E = \{e_1, ..., e_m\}$, each v_i has 3 neighbors. An integer $K \geqslant 0$.
Question: Is there a set of vertices C of size at most K s.t. each e_i is incident to at least one vertex in C?

Define a PRM as follows: let X and Y be the proptotypes of two octary random variables, i.e., $|X| = |Y| = H = 8$, and let $\mathcal{I}_x = \{X\}$ and $\mathcal{I}_y = \{Y\}$ be two interfaces. Create classes $\mathcal{S}, \mathcal{T}, \mathcal{Q}, \mathcal{U}_i, \mathcal{R}_{ij}, \mathcal{Z}_{ij}$ such that:

- class $\mathcal{S}$ implements interface $\mathcal{I}_y$, contains only one attribute $\mathcal{A}(\mathcal{S}) = \{Y\}$, no reference slot $\mathcal{R}(\mathcal{S}) = \emptyset$ and a probability distribution $\mathbf{P}(\mathcal{S}) = \{P_S(Y)\}$;

- class $\mathcal{T}$ implements $\mathcal{I}_x$, has one attribute $\mathcal{A}(\mathcal{T}) = \{X\}$, one reference slot $\mathcal{R}(\mathcal{T}) = \{\rho_\mathcal{T}\}$ with $range(\rho_\mathcal{T}) = \mathcal{I}_y$, $\mathcal{T}.X$ has for parent $\pi(X) = \{\rho_\mathcal{T}.Y\}$ and distribution $\mathbf{P}(\mathcal{T}) = \{P_T(X|Y)\}$;

- class $\mathcal{Q}$ implements $\mathcal{I}_y$, has one attribute $\mathcal{A}(\mathcal{Q}) = \{Y\}$, one reference slot $\mathcal{R}(\mathcal{Q}) = \{\rho_\mathcal{Q}\}$ with $range(\rho_\mathcal{Q}) = \mathcal{I}_x$, $\mathcal{Q}.Y$ has for parent $\pi(Y) = \{\rho_\mathcal{Q}.X\}$ and distribution $\mathbf{P}(\mathcal{Q}) = \{P_Q(Y|X)\}$;

- for each node $v_i \in V$, class $\mathcal{U}_i$ implements $\mathcal{I}_y$, $\mathcal{A}(\mathcal{U}_i) = \{Y\}$, $\mathcal{R}(\mathcal{U}_i) = \{\rho_{\mathcal{U}_i}\}$ with $range(\rho_{\mathcal{U}_i}) = \mathcal{T}$, $\mathcal{U}_i.Y$ parent has for parent $\pi(Y) = \{\rho_{\mathcal{U}_i}.Y\}$ and $\mathbf{P}(\mathcal{U}_i) = \{P_{U_i}(X|Y)\}$;

- for any i, j, class $\mathcal{R}_{ij}$ implements $\mathcal{I}_y$, $\mathcal{A}(\mathcal{R}_{ij}) = \{Y\}$, $\mathcal{R}(R)(\mathcal{R}_{ij}) = \{\rho_{\mathcal{R}_{ij}}\}$ with $range(\rho_{\mathcal{R}_{ij}}) = \mathcal{T}$, $\mathcal{R}_{ij}.Y$ has for parent $\pi(Y) = \{\rho_{\mathcal{R}_{ij}}.Y\}$ and $\mathbf{P}(\mathcal{R}_{i,j}) = \{P_{R_{i,j}}(Y|X)\}$.

- for any i, j, class $\mathcal{Z}_{ij}$ implements $\mathcal{I}_x$, $\mathcal{A}(\mathcal{Z}_{ij})) = \{X\}$, $\mathcal{R}(\mathcal{Z}_{ij}) = \{\rho_{\mathcal{Z}_{ij}}\}$ with $range(\rho_{\mathcal{Z}_{ij}}) = \mathcal{I}_y$, $\mathcal{Z}_{ij}.X$ has for parent $\pi(X) = \{\rho_{\mathcal{Z}_{ij}}.Y\}$ and $\mathbf{P}(\mathcal{Z}_{i,j}) = \{P_{Z_{i,j}}(X|Y)\}$.

All the distributions $P_S, P_T, P_Q, P_{U_i}, P_{R_{ij}}, P_{Z_{ij}}$ are distinct. For any instances, say a and b, of these classes, ab and $\prod_{i=1}^n a$ are shortcuts for $a \to b$ and $a \to a \to \cdots \to a$ respectively. In addition, for any pattern $M = a_1 a_2 \cdots a_r$ of instances, $\widehat{\mathcal{N}} = \mathcal{S}(M)$ denotes a dynamic class $\widehat{\mathcal{N}}$ implementing a_r's class interface and such that $\mathcal{A}(\widehat{\mathcal{N}})$ contains only the attribute in a_r's class, say $b \in \{X, Y\}$, $\mathcal{R}(\widehat{\mathcal{N}})$ contains only the interface of the reference in a_1's class, say I_c, $c \in \{X, Y\}$, the DAG of $\widehat{\mathcal{N}}$ is $c \to b$ and $\mathbf{P}(\widehat{\mathcal{N}}) = \{P_N(b|c)\}$, where P_N is the distribution resulting from the elimination of instances $a_1, \cdots, a_{r-1}$ from M. Pattern M is thus considered as a dynamic class, we eliminate its internal nodes $a_1, \cdots, a_{r-1}$ and we insert back its interface into the boundary graph. As M is a chain, computing P_N requires $(r-1)H^3$ operations since each a_i's elimination is of the form $\sum_b P_{a_i}(b|c)P_{a_{i+1}}(c|b)$. Now, construct the PRM's ground BN: for any $v_i \in V$, let A_i and B_i represent patterns tu_it and qtu_itq respectively, where t, q, u_i are instance prototypes of classes with the same uppercase names. For any edge $e_i = (v_j, v_k) \in E$, let C_i represent pattern qtu_jtqtu_ktq. Now, consider the PRM network $BN = s\prod_{j=1}^{40}\prod_{i=1}^{n}(A_i r_{ji})z_{00}\prod_{j=1}^{15}\prod_{i=1}^{n}(B_i z_{ji})\prod_{i=1}^{n}(C_i z_{0i})$. We will show that G has a vertex cover of size at most K if and only if $|\text{Comp}(BN)| \leqslant \Delta(K) = H + H^2 + \frac{H^3}{4}[78n + K + 2m]$, where $|\text{Comp}(BN)|$ is the number of operations performed by SPI.

First, note that BN is a chain. Hence, using SPI, removing an attribute at the end of the chain induces H^2 operations. Similarly, removing attribute s requires H operations. Assume that G has a vertex cover C of size $k \leqslant K$. For each node $v_i \in V$, consider substitution $\widehat{\mathcal{M}}_i = \mathcal{S}(A_i)$ and for any $v_i \in C$, substitution $\widehat{\mathcal{N}}_i = \mathcal{S}(B_i)$. Apply substitution m_i for all patterns A_i, substitution n_i for all patterns B_i when $v_i \in C$ and substitution qm_iq for all patterns B_i when $v_i \in V\backslash C$. Finally, as C is a vertex cover, each edge of E is incident to a node in C, hence each pattern C_i corresponding to edge (v_j, v_h) can be substituted by either $n_j m_h q$ or $qm_j n_h$. After all these substitutions, BN is a chain with s plus $R = 1 + 140n + 4m - 30k$ nodes and the number of computations, including those of the P_{M_i} and P_{N_i}, is $H + RH^2 + 2nH^3 + 4kH^3 = \Delta(k) \leqslant \Delta(K)$.

Conversely, assume there exists a set $\widehat{\mathcal{M}}$ of dynamic classes enabling to substitute BN by another instance graph BN' such that the overall number of computations is lesser or equal than $\Delta(K)$. Without loss of generality, we may assume that no substituted pattern contains a node z_{ij} or r_{ij} since no two z_{ij} or r_{ij} in BN contain the same probability distribution and, hence, only one substitution would occur in BN, thus resulting in an increase of computations (hence removing this substitution from $\widehat{\mathcal{M}}$ would produce an BN'' such that $|\text{Comp}(BN'')| \leqslant |\text{Comp}(BN')|$). For the same reason, we can assume that substituted patterns do not contain u_jtqtu_k. Let i be such that pattern A_i has not been substituted by instances from $\widehat{\mathcal{M}}$. Then one can easily prove that substituting A_i by $\mathcal{S}(A_i)$ reduces the number of operations in SPI. Similarly, it is always better to use $\mathcal{S}(A_i)$ than $\mathcal{S}(tu_i)$ or $\mathcal{S}(u_it)$. So we shall consider that $\widehat{\mathcal{M}}$ contains $\{\mathcal{S}(A_i), i = 1, \cdots, n\}$. Now, it is easy to see that we shall never use $\mathcal{S}(qtu_i)$ nor $\mathcal{S}(u_itq)$. We shall also assume that $\mathcal{S}(qtu_it)$ or $\mathcal{S}(tu_itq)$ do not belong to $\widehat{\mathcal{M}}$ since they induce $45H^2 + 3H^3$ operations in $\prod_{j=1}^{15}\prod_{i=1}^{n}(B_i z_{ji})$ and can save up to $9H^2$ in the $\prod_{i=1}^{n} C_i$ part, which is never better than not applying the substitution. To summarize: all A_i are substituted by $\mathcal{S}(A_i)$, some B_i are substituted by $\mathcal{S}(B_i)$ while others are substituted by $q\mathcal{S}(A_i)q$ and C_i are substituted by $\mathcal{S}(B_j)\mathcal{S}(A_h)q$ or $q\mathcal{S}(A_j)\mathcal{S}(B_h)$ or $q\mathcal{S}(A_j)q\mathcal{S}(A_h)q$. Let f and g denote the number of B_i substituted by $\mathcal{S}(B_i)$ and the number of C_i substituted by $q\mathcal{S}(A_j)q\mathcal{S}(A_h)q$ respectively. Then the overall number of computations, including eliminating inner attributes, is equal to $H + H^2 + \frac{H^3}{4}[78n + f + g + 2m]$. By assumption, this quantity is $\leqslant \Delta(K)$, hence $f + g \leqslant K$. Now, the nodes corresponding to substitutions in $\mathcal{S}(B_i)$ are adjacent to all the edges except j edges. By adding one node from each of these edges, we construct a vertex cover of a size $f + g \leqslant K$. ■

In a sense, this proposition is not very surprising since determining the minimal number of operations in variable elimination algorithms such as SPI or VE is equivalent to determining an optimal elimination sequence, which is known to be NP-hard (Rose et al., 1976). In addition, determining all the occurrences of a given subgraph in a graph is NP-hard as well (Garey et al., 1974). Finally, given a set of dynamic classes and their subgraph occurrences in the boundary graph, determining which ones should be substituted amounts to solve an *Independent Set* problem in which each vertex represents a boundary subgraph and edges link vertices corresponding to overlapping boundary subgraphs. Again, this problem is NP-hard (Garey and Johnson, 1979b). However, the proof of Proposition 6.1 shows that finding the best dynamic classes/substitutions remains NP-hard even in cases where inference in the ground BN is polynomial (singly-connected BNs). We shall however present in the next section an efficient approximate algorithm for determining an effective set of dynamic classes.

6.3 An Approximate Algorithm for pattern discovery

The problem of finding frequent patterns in labeled graphs has received many contributions, although their aim is somewhat different from ours as they consist of finding frequent subgraphs that appear in many graphs (Inokuchi et al., 2005; Kuramochi and Karypis, 2001; Yan and Han, 2002). However, the connection with our problem is sufficiently high that techniques from this domain can be borrowed to solve our problem. In this paper, we suggest to use a variant of gSpan (Yan and Han, 2002).

The gSpan algorithm

The gSpan (graph-based Substructure pattern mining) algorithm exploits a labeled graph and depth-first search to mine subgraphs. It builds a linear order among subgraphs using a Depth-First Search (DFS) code and a DFS tree. Each node in the DFS tree matches a subgraph and is associated with a DFS code. gSpan exploits properties of the DFS code to avoid reconsidering subgraphs twice: when a subgraph is first encountered, the DFS code associated with its node in the DFS tree is minimal, i.e., that any subsequent occurrence of this subgraph in the DFS tree will be associated with a non minimal DFS code. To detect non minimal DFS code, a simple algorithm, i.e., polynomial, tests a code to detect if it is minimal or not. If not, then we are guaranteed that the current node has already been discovered. This is very useful as it avoids comparing each new subgraph with previous ones. This repetition detection scheme is used to prune the DFS tree of uninteresting branches and helps focusing on subgraphs not yet found. Finally, gSpan goal is to find repeated subgraphs, thus it prunes any subgraph that has not at least K occurrences. The details and formalizations used by gSpan to enable such coding will not be discussed and the reader should refer to Yan and Han (2002) for a complete presentation of the gSpan algorithm.

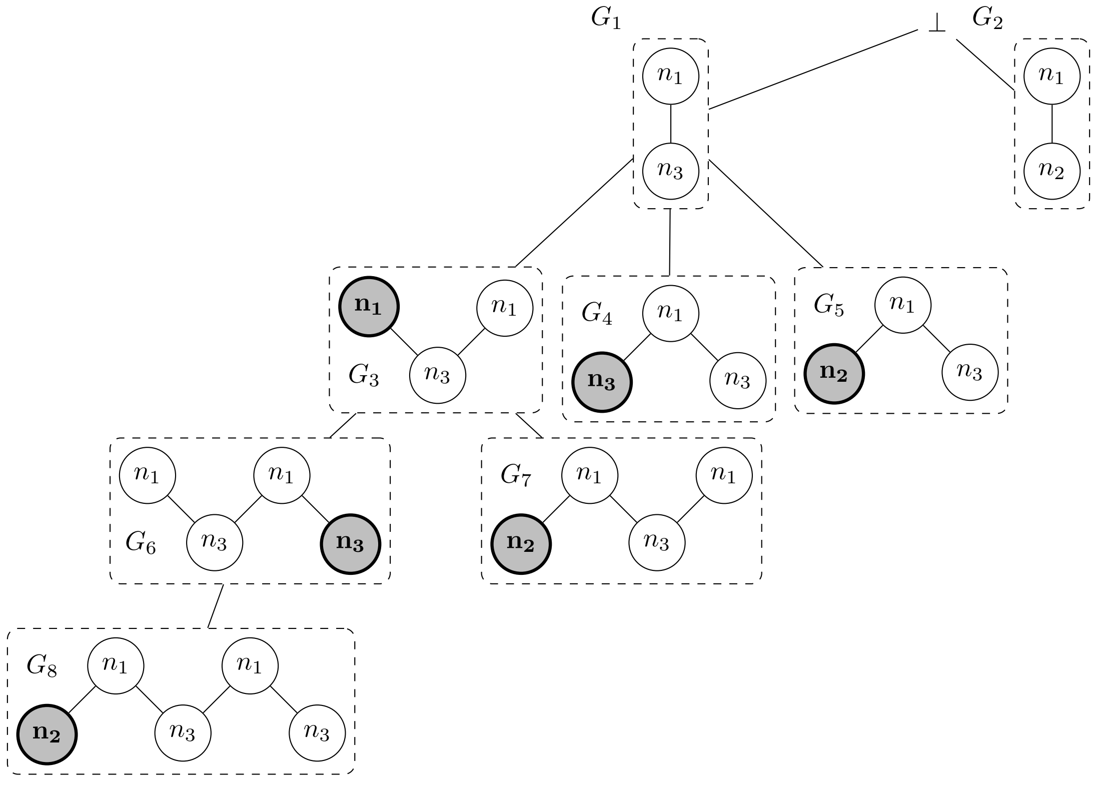
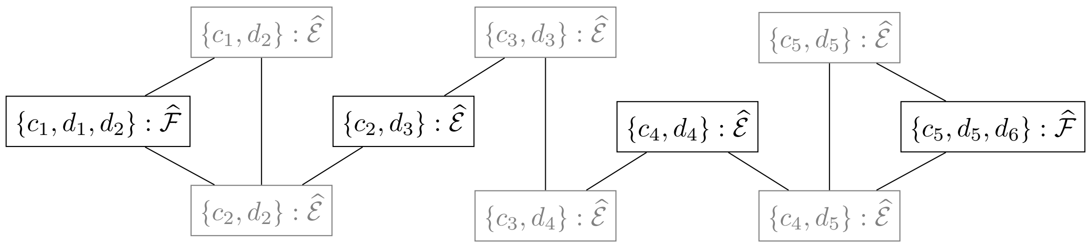


Figure 6.3: The DFS tree obtained by gSpan on the boundary graph of figure 6.2b. For any two nodes G_i and G_j, if $i < j$ then G_i was mined before G_j.

Figure 6.3 illustrates the DFS tree built by gSpan to discover subgraphs on the boundary graph of figure 6.2b. At the first level of the DFS tree we can find 1-edge subgraphs of figure 6.2b. The labeling used for each node of the DFS tree indicates the order in which each subgraph has been discovered. In this example, the most repeated 1-edge subgraph is G_1 (repeated 8 times) and the second most frequent is G_2 (repeated twice). Then, gSpan start growing G_1 to find new subgraphs: subgraphs G_3 (repeated 4 times), G_4 (repeated 3 times) and G_5 (repeated twice) are added to the DFS tree. G_3 is the most frequent subgraphs, thus gSpan searches for subgraphs by adding an edge to G_3. It finds G_6 (repeated 6 times) and G_7 (repeated twice). From G_6 only one growth is at least repeated twice, all others have only one occurence in figure 6.2b. Finally, no other node can be used to find subgraphs that are not already found and the search stops after G_8 discovery.

Adapting gSpan for mining dynamic classes

We will now explain how we can exploit a DFS tree $\mathbb{T}$ as the one in figure 6.3 for finding dynamic classes in a boundary graph. Each node G_i of the tree is mapped to a pair $(\widehat{\mathcal{D}}_i, \mathrm{M}(\widehat{\mathcal{D}}_i))$ where $\widehat{\mathcal{D}}_i$ is a dynamic class and $\mathrm{M}(\widehat{\mathcal{D}}_i)$ is the set of its matches in $\mathcal{BG}$. It can be confusing that the boundary graph's labels do not match instances names, thus it is important to recall that each node in $\mathcal{BG}$ is mapped to a single instance in the relational skeleton from which $\mathcal{BG}$ was built. $\mathbb{T}$ is initialized with all the dynamic classes corresponding to 1-edge subgraphs in $\mathcal{BG}$. In $\mathbb{T}$, nodes at level $k+1$ are derived from those at level k by extending their associated subgraph in $\mathcal{BG}$ with one of their adjacent node in $\mathcal{BG}$. As a consequence, each node of $\mathbb{T}$ represents a dynamic class whose boundary subgraph is connected and whose set of instances is nonempty. The whole tree thus reveals precisely all the possible substitutions that can be applied in the relational skeleton. We denote by $\mathbf{V} = \bigcup_{\widehat{\mathcal{D}} \in \mathbb{T}} \mathrm{M}(\widehat{\mathcal{D}})$ the set of substitutions.



$\{c_1, d_2\} : \widehat{\mathcal{E}}$
$\{c_3, d_3\} : \widehat{\mathcal{E}}$
$\{c_5, d_5\} : \widehat{\mathcal{E}}$
$\{c_1, d_1, d_2\} : \widehat{\mathcal{F}}$
$\{c_2, d_3\} : \widehat{\mathcal{E}}$
$\{c_4, d_4\} : \widehat{\mathcal{E}}$
$\{c_5, d_5, d_6\} : \widehat{\mathcal{F}}$
$\{c_2, d_2\} : \widehat{\mathcal{E}}$
$\{c_3, d_4\} : \widehat{\mathcal{E}}$
$\{c_4, d_5\} : \widehat{\mathcal{E}}$

Figure 6.4: An example of independence set for the boundary graph of figure 6.2b with patterns in figure 6.2c. Gray nodes are unchosen one.

Finally, we must select among $\mathbf{V}$ the *best* possible substitutions. To do so, we must enforce Rule 6.1. It is easily done by observing that each node of $\mathcal{BG}$ can only belong to one dynamic class and, more precisely, to one instance of this dynamic class. Hence, if we create a graph $G = (V, E)$ in which each node of V represents a given element of $\mathbf{V}$, i.e., a subgraph of $\mathcal{BG}$, and each edge $(v_1, v_2) \in E$ represents the fact v_1 and v_2 have a nonempty intersection in $\mathcal{BG}$, then any subset $W \subseteq V$ such that no pair of nodes of W are adjacent in G corresponds to a set of substitutions satisfying Rule 6.1. In other words, there is a one-to-one mapping between the *Independent Sets* of G and the sets of substitutions satisfying Rule 6.1. Of course, some substitutions are better than others because they induce higher speed-ups in Structured Inference (see the $\beta_{\widehat{\mathcal{D}}}$ and $\gamma_{\widehat{\mathcal{D}}}$ scores below). So by weighting nodes of V according to the speed-up improvements they induce, the *best* substitutions we look for correspond to solutions of a *Max Weighted Independent Set* problem (Halldórsson, 2000). Figure 6.4 illustrates the max independence set for the boundary graph of figure 6.2b using patterns in figure 6.2c. For this example we suppose that patterns $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{E}}$ have equivalent costs. The nodes labels represent which pattern is represented and above each node we can find the dynamic class' substitute in figure 6.2a.

Of course, the size of $\mathbb{T}$ is exponential and, thus, some pruning is necessary. Pruning rules will be described in the next subsection. But, to guaranty their efficiency, we shall construct $\mathbb{T}$ in such a way that the *best* dynamic classes are constructed first. This results in changing how gSpan sorts mined subgraphs. We have seen that gSpan consider frequency of each subgraphs, searching from most frequent subgraphs first. Consequently, we simply need to change the frequency of a subgraphs by a score guarantying that the *most promising* dynamic classes are mined first. Such score function is also our pruning rule and is described in the next section.

Algorithm 8: Pattern Discovery

```
   Input: A PRM $\Pi$, its boundary graph $\mathcal{BG}$ and a score function $f$
   Output: A set of dynamic classes/substitutions
1  $\mathbb{T} \leftarrow$ all dynamic classes of 1-edge subgraphs of $\mathcal{BG}$ found by gSpan
2  sort the nodes in $\mathbb{T}$ using $f$ from most promising to less promising
3  remove any unpromising 1-edge subgraphs
4  foreach new node $G_i$ found by gSpan do
5      sort $G_i$'s children that were found by gSpan using the score function
6      add promising children to $\mathbb{T}$ from most promising to less promising
7      prune unpromising children
8  sort all of $\mathbb{T}$ nodes from most promising to less promising
9  build a graph $G = (V, E)$ such that:
10     • each node $v \in V$ is a dynamic class match
11     • each edge $(v_i, v_j) \in E$ exists if $v_i$ and $v_j$ share instances
12 weight the nodes in $V$ using $f$
13 solve a Max Weighted Independent Set over $V$ and infer the substitutions of each mined
   dynamic class
14 return the set of dynamic classes with their substitutions
```

Algorithm 8 is based upon the gSpan algorithm, so we do not describe the subgraph mining part of algorithm 8. To understand algorithm 8 we simply need to know that at each iteration of gSpan will find the next most interesting subgraph with respect to the pruning rule. Algorithm 8 inputs are a PRM and a boundary graph and returns the set of dynamic classes and their substitutions. The first step of algorithm 8 is to initialize the search tree (lines 1-3). The second step of algorithm 8 is to grow the search tree using both gSpan and the pruning rule (lines 4-7). The third and final step picks the most promising dynamic classes and find possible instantiations of them in $\mathcal{BG}$ using a Max Weighted Independent Set (lines 8-14). Algorithm 8's complexity is identical to gSpan's complexity, i.e., exponential in the number of nodes in the boundary graph. This is where the pruning rule becomes primordial. Indeed, we do not want to search for all repeated subgraphs in the boundary graph, but we only need a few that are interesting for SPI. The more time we spend searching for dynamic classes, the less cost efficient they will be.

Score function

To estimate the possible gain achievable by a dynamic class in SPI, we first need to define the cost of using a dynamic class.

Definition 6.6 (Dynamic class cost) *Let $\widehat{\mathcal{D}}$ be a dynamic class, $\mathrm{M}(\widehat{\mathcal{D}})$ its matches and $\mathrm{S}(\widehat{\mathcal{D}})$ its substitutions. We denote by:*

- *$w(\widehat{\mathcal{D}})$ the number of operations necessary to eliminate $\widehat{\mathcal{D}}$'s inner attributes;*
- *$\overline{w}(\widehat{\mathcal{D}})$ the number of operations necessary to eliminate $\widehat{\mathcal{D}}$'s output attributes.*

The cost of a dynamic class $\widehat{\mathcal{D}}$ is equal to:

$$w(\widehat{\mathcal{D}}) + |\mathrm{S}(\widehat{\mathcal{D}})| \times \overline{w}(\widehat{\mathcal{D}}).$$

A dynamic class cost is the number of operations performed by SPI if we substitute instances in $\mathrm{S}(\widehat{\mathcal{D}})$ by instances of $\widehat{\mathcal{D}}$. This measure of a dynamic class efficiency is at the core of our pruning rule. Now remember that $\widehat{\mathcal{D}}$ corresponds to a 1-edge extension of its parent $\pi(\widehat{\mathcal{D}})$. So, matches of $\widehat{\mathcal{D}}$ that are not substituted can still be substituted by $\pi(\widehat{\mathcal{D}})$. Such matches are called $\widehat{\mathcal{D}}$'s left overs.

Definition 6.7 (Dynamic class left overs) *Let $\widehat{\mathcal{D}}$ be a dynamic class, $\mathrm{M}(\widehat{\mathcal{D}})$ its matches and $\mathrm{S}(\widehat{\mathcal{D}})$ its substitutions. We call $\widehat{\mathcal{D}}$ left overs the set $\mathbf{L}(\widehat{\mathcal{D}}) = \mathrm{M}(\widehat{\mathcal{D}}) \backslash \mathrm{S}(\widehat{\mathcal{D}})$.*

If we assume that the left overs of $\widehat{\mathcal{D}}$ can all be substituted as instances of $\pi(\widehat{\mathcal{D}})$, their eliminations by SPI would have cost $w(\pi(\widehat{\mathcal{D}})) + |\mathbf{L}(\widehat{\mathcal{D}})| \times \overline{\overline{w}}(\widehat{\mathcal{D}})$ where $\overline{\overline{w}}(\widehat{\mathcal{D}}) = \overline{w}_{\pi(\widehat{\mathcal{D}})} + k(\widehat{\mathcal{D}})$ and $k(\widehat{\mathcal{D}})$ corresponds to the cost of eliminating the edge added to $\pi(\widehat{\mathcal{D}})$.

Definition 6.8 (Total cost of a dynamic class) *The total cost of a dynamic class $\widehat{\mathcal{D}}$ is:*

$$\beta(\widehat{\mathcal{D}}) = w(\widehat{\mathcal{D}}) + |\mathrm{S}(\widehat{\mathcal{D}})| \times \overline{w}(\widehat{\mathcal{D}}) + w(\pi(\widehat{\mathcal{D}})) + |\mathbf{L}(\widehat{\mathcal{D}})| \times \overline{\overline{w}}(\widehat{\mathcal{D}}).$$

We will compare the total cost of $\widehat{\mathcal{D}}$ with the cost of not using it.

Definition 6.9 (Negative cost of a dynamic class) *The negative cost a dynamic class $\widehat{\mathcal{D}}$ is the cost of using its parent $\pi(\widehat{\mathcal{D}})$ over substitutions in $\mathrm{S}(\widehat{\mathcal{D}})$:*

$$\gamma(\widehat{\mathcal{D}}) = w(\pi(\widehat{\mathcal{D}})) + |\mathrm{S}(\widehat{\mathcal{D}})| \times \overline{\overline{w}}(\widehat{\mathcal{D}}).$$

We can now define the effective cost of using a dynamic class $\widehat{\mathcal{D}}$.

Definition 6.10 (Effective cost of a dynamic class) *The effective cost of a dynamic class $\widehat{\mathcal{D}}$ is the difference between the total cost $\beta(\widehat{\mathcal{D}})$ and its negative cost $\gamma(\widehat{\mathcal{D}})$:*

$$\begin{aligned} \alpha(\widehat{\mathcal{D}}) &= \beta(\widehat{\mathcal{D}}) - \gamma(\widehat{\mathcal{D}}) \\ &= w(\widehat{\mathcal{D}}) + s(\widehat{\mathcal{D}}) \times (\overline{w}(\widehat{\mathcal{D}}) - \overline{\overline{w}}(\widehat{\mathcal{D}})). \end{aligned}$$

Then we can easily express our pruning rule using the effective cost of a dynamic class.

Rule 6.2 (Pruning rule of dynamic classes) *A dynamic class $\widehat{\mathcal{D}}$ is pruned whenever the total cost of $\widehat{\mathcal{D}}$ is greater than or equal to its negative cost:*

$$\alpha(\widehat{\mathcal{D}}) > 0,$$

i.e., when the cost of using *$\widehat{\mathcal{D}}$ is higher than the cost of* not using *it.*

Finally, note that $\mathrm{S}(\widehat{\mathcal{D}})$, $w(\widehat{\mathcal{D}})$, $\overline{\overline{w}}(\widehat{\mathcal{D}})$, $k(\widehat{\mathcal{D}})$ can be estimated quickly: as shown previously, $\mathrm{S}(\widehat{\mathcal{D}})$ can be estimated by solving a *Max Independent Set* problem induced by $\mathbf{M}(\widehat{\mathcal{D}})$. To estimate $w(\widehat{\mathcal{D}})$, it is sufficient to compute an elimination order over $\widehat{\mathcal{D}}$'s inner attributes and to sum-up the sizes of the factors creating in the process. Eliminating the remaining variables provides an estimation of $\overline{w}(\widehat{\mathcal{D}})$. $k(\widehat{\mathcal{D}})$ can be estimated similarly. Figure 6.5 illustrates a dynamic class and the factors created by eliminated its inner attributes (X_1, X_2, X_3 and X_4) and its output attributes (X_5 and X_6). The results are equivalent to equation 5.1 presented in chapter 5 and we can see with table 6.1 that a dynamic class' cost effectiveness is dependent on the class' context: here the smaller $\pi(X_1)$, $\pi(X_2)$, Ch(X_5) and Ch(X_6), the more cost effective $\widehat{\mathcal{D}}$.

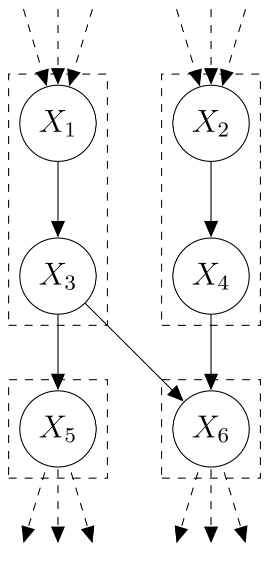


Figure 6.5: A dynamic class with four components: $\{X_1, X_3\}$, $\{X_2, X_4\}$, $\{X_5\}$ and $\{X_6\}$.

Note however that $\mathbb{T}$ is not α-decreasing, i.e., it may happen that $\alpha(\widehat{\mathcal{D}}) > 0$ for a given dynamic class $\widehat{\mathcal{D}}$, but not for some of its descendants. This property results from the fact that, in these descendants the number of inner nodes may be far higher than that in $\widehat{\mathcal{D}}$, hence decreasing $w(\widehat{\mathcal{D}})$ (dropping constraints on the junction tree's elimination order) as well as $\overline{w}(\widehat{\mathcal{D}})$ (the inner nodes do not belong to the boundary). Unfortunately, the α-non-decreasing property does not allow for a clear pruning rule. We chose to stop mining as soon as a dynamic class was found uninteresting, i.e., it was α-positive. This choice results from two observations: (i) mining subgraphs is time-expensive, in fact for large graphs it can be much more complex than inference (Yan and Han, 2002), thus we want to stop the mining as soon as possible; (ii) we observed that small patterns were more cost-effective as they require less time to be found and are more frequent than large ones. Indeed large patterns can seem interesting but finding them will require to mine all intermediate sub-graphs and they will usually be less frequent. Thus the gain of using large patterns as dynamic classes may not necessarily be high regarding the time spent finding them.

Elim. order	Created factors
X_1	$\pi(X_1)$, X_1, X_3
X_2	$\pi(X_2)$, X_2, X_4
X_4	$\pi(X_2)$, X_4, X_3, X_6
X_3	$\pi(X_1)$, $\pi(X_2)$, X_3, X_5, X_6
X_5	$\pi(X_1)$, $\pi(X_2)$, Ch(X_5), X_6
X_6	$\pi(X_1)$, $\pi(X_2)$, Ch(X_5), Ch(X_6)

Table 6.1: Factors created during a possible elimination order over figure 6.5 attributes. Summing the factors sizes gives a good estimation of the dynamic class efficiency.

6.4 Experimental Results

We now describe different set of experiments that highlight the gain in inference speed resulting from the combination of structured inference and pattern mining. In each experiment, we compared our new algorithm (subsequently denoted as PD for Pattern Discovery) with Structured Probabilistic Inference (SPI), the standard inference algorithm for structured inference and also with Variable Elimination (VE), a classic and standard probabilistic inference algorithm for Bayesian Networks. Response times reported for PD take into account both pattern mining and inference. For experiments using VE, results include both grounding and inference time. However, to allow ground inference on large networks, we exploit the information encoded by PRMs to prevent unnecessary copies of CPTs in the ground BNs. It is important to note that our experiments included no evidence. This choice was motivated by the fact that the structure of the network varies drastically given evidence. Our goal here was to show how pattern mining can improve inference when there exist repetitions in the network. Moreover, evidence is not a good indicator of repetitions as it can either be identically applied in each pattern, thus preserving repetition, or applied randomly, thus breaking the structure. Experiments 1 and 2 show the results of our new approach on networks with and without repetitions, hence providing a good insight of PD's performance. All our experiments were performed on an Intel Xeon at 2.7 Ghz.

The key to understand these experimentations lies in the generation of the benchmarked PRMs. High level frameworks such as PRMs offer a wide variety of generation methods. Here, our primary concern was the generation of PRMs in which we could control the amount of structure repetition in order to prove that, when confronted to a large amount of pattern repetitions: (i) a substantial speed gain can be achieved and (ii) our approach does not suffer from a prohibitive pattern mining cost. Our generator takes the following parameters as inputs: $domain$ is the domain size of each attribute; min_{attr} is the number of attributes common to all classes; max_{attr} is the number of attributes in each class; c is the minimal number of classes; max_{ref} is the maximal number of reference slots allowed per class; n is the number of instances in the system.

The PRM's generation process is performed as follows: first, we generate an interface with min_{attr} attributes which will be implemented by all classes and will be the slot type of each reference slot in each class. Next, for all $k \in [0, \cdots, \max_{ref}]$, a class with precisely k reference slots is created. Then, if $max_{ref} < c$, we generate new classes until exactly c classes have been created. For those new classes, the number of reference slots is chosen randomly between 0 and max_{ref}. Finally, we generate a DAG S representing the relational skeleton of our generated system: each node represents an instance and an arc $i \rightarrow j$ represents the fact that there exists $\rho \in \mathcal{R}(j)$ such that $i = j.\rho$. For a given node i with π_i parents in S, we instantiate a class randomly chosen among all the classes with precisely π_i reference slots. A given class C is generated as follows: we first create a DAG G_C with max_{attr} nodes, we then add to C k reference slots and max_{attr} attributes. Dependencies between attributes are defined using G_C. For each reference slot ρ, we create a slot chain $\rho.A$, where $A \in \mathcal{A}(I)$ is chosen randomly among all the attributes in $\mathcal{A}(I)$. The slot chain is then added as a parent of an attribute of C chosen randomly. DAGs are generated using the algorithm provided in Ide et al. (2004).

In our first set of experiments, we generated systems with an increasing number of instances. Each class contains 15 attributes ($max_{attr} = 15$), each attribute's domain size is equal to 4 ($domain = 4$) and each class has at most 4 incoming arcs ($max_{ref} = 4$). Finally, the minimal amount of classes required was set to $c = 5$, which implies that there are precisely $max_{ref} + 1 = 5$ classes in each system. These experiments highlight the behavior of PD when many repetitions can be found in the system. Fig. 6.6 shows the response times of PD, SPI and VE when no evidence is observed and with a number of instances varying from 100 to 1000. Clearly, in this case, PD significantly outperforms both VE and SPI.

An important factor is the ratio of PD's inference time over that of SPI. The gain of PD against SPI and that of SPI against VE are due to the presence of structural repetition in the generated networks. It can be seen that SPI's complexity is less impacted by the size of the system than VE's complexity. But for small systems with small classes, SPI does not guarantee a considerable speed gain. By exploiting pattern mining, PD significantly increases the gain obtained by repetition. Thus, where SPI does not perform well compared to VE, PD infers larger patterns that can drastically increase performance. In our first experiments, there is enough structure to see the possible gain provided by our new approach. Yet, we must also consider cases where there are few or even no

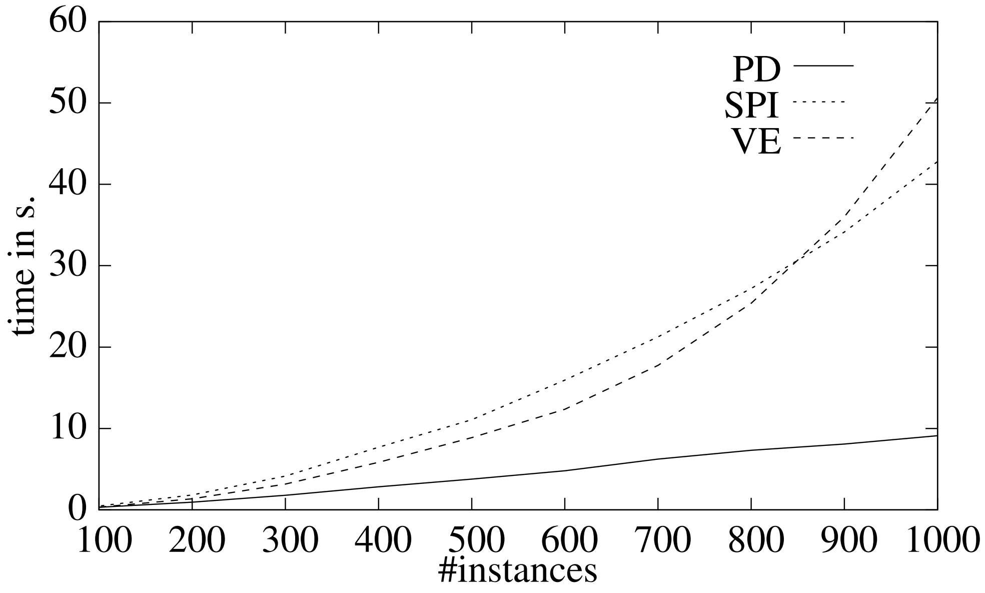


Figure 6.6: Structural repetition is an important factor for PD's performance: systems with strong repetitions favor drastically our approach.

structural repetitions. The amount of pattern repetitions can be influenced by the number of classes, so if we increase that number we should observe a less favorable ratio between PD's and SPI's inference time against VE. This is the purpose of our second set of experiments.

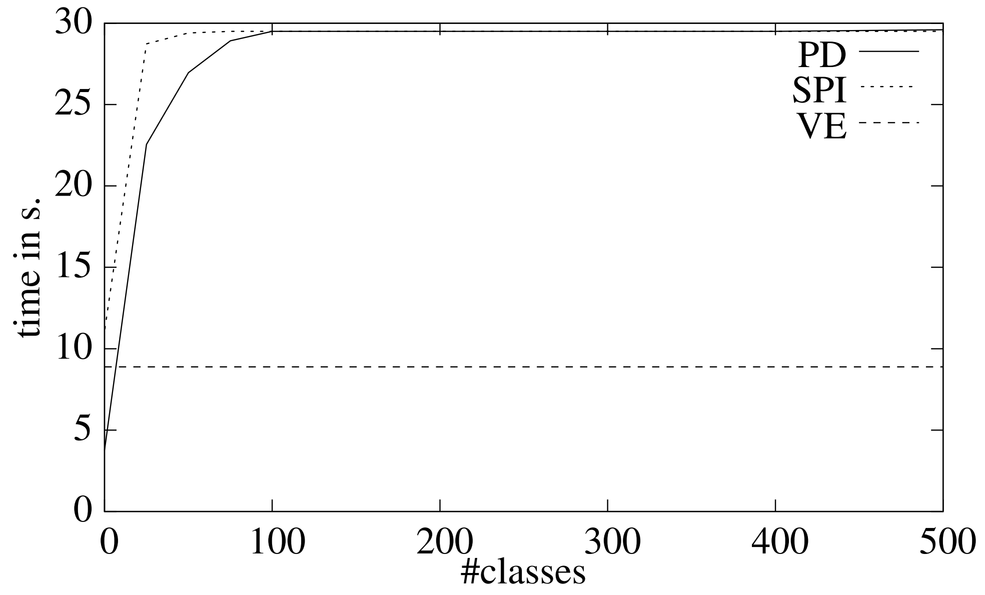


Figure 6.7: Structural repetition is an important factor for PD's performance. Unsurprisingly, performance decrease dramatically for systems with no structural repetition.

In our second set of experiments we generated systems with an increasing number of classes ($c \in [0, 500]$) and 500 instances. The remaining parameters are equal to those of the first experiment. The goal here is twofold: we want to show that, when no structure is exploitable, there is no overhead in proceeding with the pattern mining and that pattern repetitions is critical for PD's performance. Fig. 6.7 shows that when the number of classes increases dramatically, the speed gain induced by PD and SPI are considerably less significant. If we compare those results with those obtained by VE, we see that PD and SPI are considerably counter-performing. To anyone familiar with structural inference, this is an unsurprising result and these results can be explained by the fact that the elimination order used by PD and SPI (inner attributes before outer attributes) is in most cases suboptimal. If PD and SPI show better results than VE in Fig. 6.6 it is only because the gain resulting from the reduction of redundant computations compensate the suboptimal elimination order. Fortunately, detecting repetition is trivial in an object-oriented framework as the amount of instantiations of each class is a good indicator of structural repetition. The presence of evidence is also a good indicator, as different evidence will break down the structure and thus reduce the amount of repetition in the network. We can easily switch to classic inference if needed by detecting situations which would lead to counter-performing results: few instantiations of each classes, heavy evidence, seemingly random evidence. Finally, we observe no over-cost due to pattern mining. This is also an unsurprising result as our pruning rules take into account frequencies and cut the mining process when such value is too low (here the minimal frequency allowed was set to 2).

Table 6.2: Third experiment: patterns mining efficiency for PD. Values are averages. Inst. stands for instances, attr. for attributes and pat. for pattern.

#inst.	#pat.	pat. repetition	max pat. repetition	#inst. per pat.	max inst. per pat.	% of attr. in a pat.
200	11.88	2.92	6.26	2.15	4.08	37.29%
400	24.68	3.40	10.46	2.25	4.71	47.20%
600	36.35	3.91	15.92	2.36	5.25	55.90%
800	46.51	4.50	20.25	2.45	5.62	64.09%
1000	54.19	5.25	30.07	2.62	6.12	75.54%

In our third experiment, we analyze the amount of patterns found by PD with the parameters from experiment 1 ($max_{attr} = 15, domain = 4, max_{ref} = 4, c = 5, max_{ref} + 1 = 5$). The results of this experiment are summarized in Tab. 6.2. A noticeable point is the low number of instances in each pattern. This is a consequence of our pruning rule which was designed to be strict. It favors smaller patterns because larger ones are in most cases less cost effective (they often induce a larger clique than an optimal elimination order would) and because they are less frequent. In general, discovered patterns consisted of few small patterns largely repeated and many different patterns less repeated. The latter were used to fill-in the gaps in the structure once the main patterns were applied. If we consider the last column of Tab. 6.2 we can see that the larger a system, the more the attributes covered. The fact that the coverage increases with the system size explains why the inference time of PD increases linearly with the system size: the large number of usable patterns compensates the complexity induced by the number of instances.

To conclude our experiments, we applied PD to a classic BN: the Pigs network. This network is remarkable in that it only contains two distinct CPTs, which are represented in our framework by two classes. The network in itself is too small to point out any significant gain in inference time, however it is still interesting to analyze the patterns found by PD. Our approach mined 14 different patterns. On average, they are repeated 11 times and the maximal amount of repetitions equals 45. Only patterns with 2 instances are found. Discovered patterns cover up to 69% of the 441 attributes present in the Pigs network. As for our previous results, our pruning rules favor smaller patterns since larger ones tend to be less cost effective and less frequent. While the size of the Pigs network does not enable to point out the efficiency of our approach in terms of inference time, the existence of such structures and the results we obtained over random networks can help conclude to the efficiency of our approach. We can also point out its usefulness w.r.t. modeling: by pointing out frequent patterns in a system we can infer new classes which can then be used by experts for modeling purposes.

Conclusion

This thesis objectives were to define a strong object-oriented framework for PGMs and exploit the structure encoded by the object-oriented paradigm to enhance probabilistic inference. We have seen that several existing framework proposed implementations of the object-oriented paradigm, some of them purposely (OOBNs, PRMs) others unwittingly (DBNs, MSBNs). Other frameworks chose to extend BNs using other representation paradigms (MEBNs, parfactors). In chapter 3 we presented each of these extensions and discussed why each of them neither implemented a fully object-oriented framework nor offered tools required to model complex systems. In chapter 4 we presented our first contribution, an extension of PRMs that reinforces their object-oriented aspect. We redefined and extended object-oriented notions such as class inheritance, subtype polymorphisms and abstraction. We also defined new concepts to PGMs, such as multiple inheritance using interfaces, attributes types inheritances and attributes cast descendants. We have also shown how such strong object-oriented PRMs can be represented using parfactors. In chapter 5 we extended Pfeffer's initial works and presented a new version of SVE. While simplifying it, it also highlighted several of its flaws. As SVE proved to be unpractical to our domains, we then proposed a generalization of structured inference. SPI is a generic inference scheme, like conditioning algorithms or junction tree based algorithms rather than a precise probabilistic inference algorithm. Indeed, we have shown how structure in object-oriented PGMs can be exploited to create a MN with fewer variables on which any classic inference algorithm can be applied. Our generalization of structured inference also offers the possibility of exploiting a classic probabilistic inference optimization technique: d-separation analysis. In chapter 6 we extended SPI by pushing the concept of structured inference a step further. Using the structure encoded by classes and instances, we adapted a frequent subgraph mining algorithm to detect and exploit instance patterns. Doing so enables the discovery of new classes that can be exploited by SPI to speed-up even more probabilistic inference.

Unfortunately, chapter 4 and appendix A do not detail the thought process leading to the creation of the SKOOL language and to the extensions added to PRMs. It is important to understand that these features have been the result of many hours of brainstorming among different experts in reliability and risk management. There also have been several systems defined using the SKOOL language. Such systems are not exposed in this thesis because they result of work of several members of the ANR SKOOB project, in which I have barely contributed. However, the tools developed during this thesis have been used by several experts to model systems using strong object-oriented features. It is clear that a public database of object-oriented PGMs must be created to offer the possibility of benchmarking different implementations.

Regarding implementation, the contributions presented in this thesis have all been implemented in the aGrUM framework (`http:://agrum.lip6.fr`). Implementing object-oriented PRMs have been an extremely difficult and time consuming task. This is certainly the biggest flaw of our contributions. Indeed, where BNs offer a simple framework for probabilistic reasoning, the framework presented in this thesis is complex and cannot be implemented trivially. Where many probabilistic inference algorithms can be easily implemented (VE, SS), algorithm 6 and 8 require good programing skills and consequent rewriting to become performing. However, such flaw is also characteristic of many new inference algorithms and BN extensions.

One of the most underused feature of this thesis is the bridge defined between PRMs and parfactors. Parfactors and lifted inference are one of the main source of recent contributions. They offer new possibilities, both in modeling systems and in inference. By proving that our object-oriented framework can be represented using parfactors, we created the possibility to add first-order features to our framework. In a way, MEBNs are maybe the best illustration to what can be FOPM using object-oriented modeling tools. However, there are also many possibilities regarding inference. Lifted inference is particularly inefficient in worlds with strong structural observations, i.e., systems in which relations between objects are known. However, such systems are particularly well exploited by our algorithms. Consequently there are many possibilities of combining lifted inference with structured inference to both speed-up probabilistic inference in FOPMs and to offer more expressive power to our framework.

Extending the pruning rule used by algorithm 8 is certainly one of the most interesting extensions of the work presented in this thesis. Furthermore, we can also imagine using such rules directly on instances, i.e., detect which classes offer good performance when attributes are eliminated at class level. We could also break down any structure induced by classes and directly search for classes at a ground level.

Finally, there is still considerable work regarding machine learning. Parameter learning is a must have feature for most practical industrial use of BNs, and we can infer the same for our framework. Structural learning is also an interesting domain and there is certainly some common ground with the content of chapter 6. Indeed searching for instances patterns in a system is a problem close to discovering classes in a BN.

Appendices

Appendix A

SKOOL language specification

The SKOOL language purpose is to model PRMs using a strong object-oriented syntax. SKOOL syntax is inspired by the Java programming language.

A.1 SKOOL project structure

As in Java, the SKOOL language is made of *compilation units* that are placed in *packages*. It is possible to encode in a single file an entire project but it is not recommended. Ideally, a package matches a specific directory and in each file we can find one compilation unit. The following is a sample SKOOL project.

```
fr\
  | skoob\
  | printers\
  | variables.skoot   // types definition
  | powersupply.skool // class PowerSupply definition
  | equipment.skool   // class Equipment definition
  | room.skool        // class Room definition
  | computer.skool    // class Computer definition
  | printer.skool     // class Printer definition
  | example.skoos     // system Example definition
  | query.skoor       // request query definition
```

File extensions can be used as indicator of the file's compilation unit nature. The following extensions are allowed: `.skool` for classes, `skoos` for systems, `skoot` for types and `skoor` for queries. As in Java, filenames convention is not mandatory and users can chose to regroup several compilation units in a single file. This is mostly useful when modeling small systems.

A.1.1 Compilation units

There exists four different compilation units. A compilation unit *declares* a specific element in the modeling process and can either be: an attribute type, a class, a system or a query. Each compilation unit can start with a header used to declare the package in which the unit belongs and its dependencies:

```
<SKOOL> ::= [<header>] <compilation_unit> [(<compilation_unit>)]
<header> ::= <package> | <import>
<compilation_unit> ::= <type_unit> |
                       <class_unit> |
                       <system_unit> |
                       <query_unit>
```

A.1.2 Header syntax

The syntax to declare the package to which a compilation units belongs is:

```
<package> ::= package <path> ";"
<path> ::= <word> [ ( "." <word> ) ]
<word> ::= <letter> (<letter> | <digit>)
<letter> ::= 'A'..'Z' + 'a'..'z'+ '_' .
<digit>  ::= '0'..'9'.
```

An example:

```
package fr.skoob.printers;
```

Basically, it is the path to the folder in which the unit's file is located. Directory separators are represented using dots and the environment variable defining the paths in which we can find packages is implementation specific. Ideally, it must have a similar behavior than the `CLASSPATH` variable used by Java.

Depending on the compilation unit's nature, we may need to use compilation units defined in other files. In such case, we say that a given compilation unit has dependencies. Declaring dependencies is done using the `import` keyword. The syntax is:

```
<import> ::= import <path> ";"
```

An example:

```
import fr.skoob.printers.computer;
```

The `import` instruction is made of a package's name and a compilation unit's name (not the file in which the unit is defined).

A.2 Attribute type declaration

In SKOOL we can only define discrete random variables that are either user-defined or one of the three built-in types: `boolean`, `int` and `real`. User-defined types are declared using the keyword `type`:

```
<type_unit> ::= <built_in> | <user_defined>
<user_defined> ::= <basic_type> | <subtype>
<basic_type> ::= type <word> <word> "," ( "," <word> )+ ";"
```

The rule `<basic_type>` defines a random variable's domain, the first `word` is the type's name and the following are the domain label's names. There must be at least two labels. The rule `<subtype>` is explained in the next section. Some examples:

```
boolean exists;
int (0,9) power;
real (0, 90, 180) angle;
type t_state OK, NOK;
```

A.2.1 Subtyping

A subtype can be declared using the `extend` keyword. A subtype declaration syntax is:

```
<subtype> ::= type <word> extend <word> <type_spec>
<type_spec> ::= <word> ":" <word> (<word> ":" <word>)+
```

The first `<word>` is the type's name, the second the names of its supertype and the rule `<type_spec>` defines the labels specializations: the first `<word>` belongs to the subtype and the second `<word>` belongs to the supertype. A example of subtype declaration:

```
type t_degraded extends t_state
OK: OK,
DYSFONCTION: NOK,
DEGRADED: NOK;
```

In this example, `DYSFONCTION` and `DEGRADED` are specialization of of the label `NOK` of type `t_state`. When declaring a subtype, it is mandatory that the supertype be visible, i.e., :

- either the supertype was declared in the same file before the subtype;
- or the supertype declaration unit has been imported.

A.2.2 Built-in types

The built-in types are: `boolean`, `int` and `real`. The `boolean` type is used to represent binary random variables taking the values `false` or `true`. Note that the order used is always `false` first and then `true`. The `int` must be used to defined random variables over ranges: `int(0,9) power` defines a random variable power over the domain integers from 0 to 9. The `real` type must be used to define random variables over discretized continuous domains. For example, `real(0, 90, 180) angle` defines a random variable over the two values `[0-90[`, `[90,180[`. When defining a random variable with the `real` there must be at least three parameters. The syntax of built-in types is:

```
<built_in> ::=
  boolean <word> ";" |
  int "(" <digit>* "," <digit>* ")" <word> ";" |
  real "(" <digit>* ( "," <digit>* )+ ")" <word> ";"
```

A.3 Class and interface declaration

Classes are declared as follows:

```
<class_unit> ::= <class> | <interface>
<class> ::= class <word> [ extend <word> ] "{" <class_elt>* "}"
<class_elt> ::= <reference_slot> | <attribute> | <parameter>
<interface> ::= interface <word> [ extend <word> ]
                "{" <interface_elt> "}"
<interface_elt> ::= <reference_slot> | <abstract_attr>
```

The first `word` is the class' name and the second (if any) the class' superclass. An example:

```
class A {
  // reference slots and attributes declaration
}
```

A.3.1 Reference slot declaration

In the SKOOL language, simple and complex reference slots are declared differently. Simple reference slots can only refer to a single instance and complex reference are considered as arrays. The syntax for declaring a reference slot is:

```
<reference_slot> ::= [internal] <word> [ "[" "]" ] <word> ";"
```

The keyword `internal` can be used to specify a reference slot as internal to a class. Consequently, the reference cannot be accessed by attributes outside of its encapsulating class. The first `word` is the reference slot range's name:, if it is complex `[ ]` are added as suffixes to the range's name and the last `word` is the reference slot's name. The following is an example of two reference slots declaration:

```
// Simple reference slot
A refA;
// Complex reference slot
B[] refB;
// Simple internal reference slot
internal class_name ref_name;
// Complex internal reference slot
internal class_name[] ref_name;
```

A.3.2 Attribute declaration

Attributes are declared as follows:

```
<attribute> ::= <word> <word> [ dependson <parents> ]
                ( <CPT> | <function> ) ";"
<parents> ::= <path> ( "," <path> )*
<CPT> ::= "{" ( <raw_CPT> | <rule_CPT> ) "}"
```

The first word is the attribute's type, the second its name. Dependencies are defined as a list of parents separated by commas. Each parent is defined by a <path>, i.e., a list of reference slots ending by an attribute (there can only be an attribute's name in such paths). We will detail CPTs declaration in the next section. Functions will be detailed in section A.6. The following is an example of attribute declaration:

```
// An attribute with no parents
a_type a_name {
  cpt_declaration
};
// An attribute with two parents
another_type another_name dependson parent_1, parent_2 {
  cpt_declaration
};
```

A.3.3 Raw CPT declaration

When declaring a raw CPT, all values of the CPT must be given. In such case, the values order is paramount. The declaration used in SKOOL is by columns, i.e., column in the CPT must sum to one. Let us consider the boolean attributes X, Y and Z such that X depends on Y and Z. The first value in X's CPT declaration will be the probability $P(X = false|Y = false, Z = false)$ and the next value is done by increasing the domain of the last attribute by one. In this case, the second value is the probability $P(X = false|Y = false, Z = true)$. When the last attribute reached its last value, we set to its first value and increase the previous attribute. For example, the third value of X's CPT would be the probability $P(X = false|Y = true, Z = false)$. The following illustrates how we can use comment to make raw CPT definitions easier to read.

```
boolean X dependson Y, Z {
  // Y=       |     false    |     true     |
  // Z=       | false | true | false | true |
  /* false */ [ 1.0,    0.3,   1.0,    0.01,
  /* true  */   0.0,    0.7,   0.0,    0.99 ]
};
```

Of course, the CPT declaration is dependent on the order in which the parents are declared. The syntax of a raw CPT declaration is straightforward:

```
<raw_CPT> ::= "[" <float>+ ( "," <float>+ )+ "]"
```

A.3.4 Rule based CPT declaration

Rule based declarations exploit wildcards to reduce the number of parameters for CPT with redundant values. A rule syntax is:

```
<rule_CPT> ::= ( <word> ("," <word>)* ":" <float> ";" )+
```

There is no limit in the number of rules and when two rules overlap the last rule takes precedence. The following is an example of rule based declaration using the previous example:

```
boolean X dependson Y, Z {
// Y,      Z:     X=false, X=true
   *,      false: 1.0,     0.0;
   true,   true:  0.01,    0.99;
   false,  true:  0.3,     0.7;
};
```

A.3.5 Parameters

Parameters are declared using the following syntax:

```
<parameter> ::= <word> <word> [ default <word> ] ";"
```

The first `word` is the parameter's type, the second its and the third (if any) is the parameter's default value. It must be a valid label of the parameter's type. Some examples:

```
// A parameter with no default value
boolean X;
// A parameter with a default value
boolean Y default true;
```

When the parameter does not have any default value, it will be necessary to provide one for each instantiation of its encapsulating class in the system declaration.

A.3.6 Interface's abstract attributes

An abstract attribute in an interface declaration syntax is:

```
<abstract_attr> ::= <word> <word> ";"
```

Where the first `<word>` is the abstract attribute's type and the second its name.

A.4 System declaration

A system is declared as follows:

```
<system> ::= system <word> "{" <system_elt>* "}"
<system_elt> ::= <instance> | <affectation>
```

The first `word` is the system's name. A system is composed of instance declarations and affectations. Affectations either assign an instance to an instance's reference slot or assign a parameter's value. The following is illustrates a system declaration:

```
system name {
  // body
}
```

A.4.1 Instance declaration

The syntax to declare an instance in a system is:

```
<instance> ::= <word> [ "[" digit* "]" ] <word> ";"
```

The first `word` is the instance's class and the second its name. For example, if we have a class A, we could declare the following instance:

```
A an_instance;
```

We may want to declare arrays of instances. To do we need to add `[n]` as a suffix to the instance's type, where `n` is the number of instances already added in the array. if $n = 0$ then we can simply write `[]`.

```
// An empty array of instances
A_class[] a_name;
// A array of 5 instances
A_class[5] another_name;
```

A.4.2 Affectation

```
<affectation> ::= <path> += <word> ";" |
                  <path> = <word> ";"
```

It is possible to add instances into an array, using the += operator:

```
// Declaring some instances
A_class x;
A_class y;
A_class z;
// An empty array of instances
A_class[] array;
// Adding instances to array
array += x;
array += y;
array += z;
```

Reference affectation is done using the = operator:

```
class A {
  boolean X {[0.5, 0.5]};
}

class B {
  A myRef;
}

system S {
  // declaring two instances
  A a;
  B b;
  // Affecting b's reference to a
  b.myRef = a;
}
```

In the case of multiple references, we can either use the = to affect an array or the += operator to add instance one by one:

```
class A {
  boolean X {[0.5, 0.5]};
}

class B {
  A myRef[];
}

system S1 {
  // declaring an array of five instances of A.
  A[5] a;
  // declaring an instance of B
  B b;
  // Affecting b's reference to a
  b.myRef = a;
}
// An alternative declaration
system S2 {
  // declaring three instances of A
  A a1;
  A a2;
  A a3;
  // declaring an instance of B
  B b;
  // Affecting b's reference to a
  b.myRef += a1;
  b.myRef += a2;
  b.myRef += a3;
}
```

We can mix = and +=operators, but operator = overwrites previous affectations. Parameters without default values must be defined each time their encapsulating class is instantiated. If is also possible to define values for parameters with default values. To do so, we use the = operator.

```
class A {
  boolean param;
}

system S {
  // declaring an instance of A
  A a;
  // Affecting b's reference to a
  a.param = true;
}
```

The value assigned to a parameter must be valid given the parameter's type.

A.5 Query unit declaration

A query unit is defined using the keyword `request`. Its syntax is the following:

```
<query_unit> ::= request <word> "{" <query_elt>* "}"
<query_elt> ::= <observation> | <query>
<observation> ::= ( <path> = <word> ) |
                  ( unobserved <path> )
                  ";"
<query> ::= "?" <path> ";"
```

The first `word` is the query's name. In a query unit we can alternate between observations and queries. An observation observe an attribute with a given value. Evidence are affected using the = operator. A query over attribute `X` asks to infer the probability $P(X|\mathbf{e})$ where $\mathbf{e}$ is evidence over attributes in the system. This is done using the `?` operator. The keyword `unobserve` can be used to remove evidence over an attribute.

```
system mySystem {
  // a system declaration
}

request myQuery {
  // adding evidence
  mySystem.anObject.aVariable = true;
  mySystem.anotherObject.aVariable = 3;
  mySystem.anotherObject.anotherVariable = false;
  // asking to infer some probability value given evidence
  ? mySystem.anObject.anotherVariable;
  // remove evidence over an attribute
  unobserve mySystem.anObject.aVariable;
}
```

A.6 Functions

Functions in SKOOL are considered as tools to define attributes CPTs. They replace the CPT declaration by a specific syntax depending to which family the function belongs to. There exit three kinds of functions in SKOOL. The first kink regroups built-in functions called aggregators. These functions are used to quantify information hold in multiple reference slots. The second sort regroups deterministic functions and the third probabilistic functions. The last two sorts of functions are not built-in functions and are implementation specific. We only provide a generic syntax to keep uniformity between different SKOOL implementations. All functions share the same syntax:

```
<function> ::= ( "=" | "~" ) <word> "(" [ <parameters> ] ")"
<parameters ::= <word> ( "," <word> )*
```

The use of = is reserved for deterministic functions and ~ for probabilistic functions. There are only four built-in functions in the SKOOL language that are deterministic functions called aggragators. There are four built-in aggregators in the SKOOL language: `min`, `max`, `exists` and `forall`.

The `min` and `max` functions require a single parameter: a list of slot chains pointing to attributes. The attributes must all be of the same type or share some common supertype. If the common type is not a int, then the type's values order is used to compute the min and max values.

```
class A {
  // Some declarations
  int(0,10) myMax = max([chain_1, chain_2, ...]);
  // Some declarations
  int(0,10) myMin = max(chain_1);
}
```

If there is only one element in the list of slot chains the `[]` are optional.

The `exits` and `forall` require two parameters: a list of slot chains and a value. As for `min` and `max`, all attributes referenced in the slot chains list must share a common type or supertype. The value must be a valid value of that common supertype. `exists` and `forall` attribute type must always be a boolean.

```
class A {
  // Some declarations
  boolean myExists = exists([chain_1, chain_2, ...], a_value);
}
```

A.7 Class inheritance and interfaces

Class inheritance and interface implementation are two kinds of class specialization available in the SKOOL language.

A.7.1 Class inheritance

The first form of specialization is called class inheritance. It consists in defining a class, called the subclass, by specializing another class, called the superclass. The syntax to declare a subclass is as follows:

```
class MySubClass extends MySuperClass {
  // reference slots and attributs declaration
}
```

A subclass inherits all the reference slots and attributes defined by its superclass (and those inherited by the superclass' superclass). Thus, if we do not want to change properties inherited, we do not need to declare them. However, all inherited elements are considered as encapsulated by the subclass, thus they can be used to define dependencies in the subclass attributes. The following example illustrates this principle:

```
class C {
  boolean z {[0.3, 0.7]};
}

// A superclass
class A {
  C myRef;
  boolean b {[0.2, 0.8]};
}

// A subclass of A
class B extends A {
  t\_state state dependson b, maRef.z
  { [ /* ... */ ] };
}
```

Another important feature of class inheritance is reference slot overloading and attribute overloading. It is possible to:

- specialize attributes or references slots;
- changes dependencies of an inherited attribute;
- changes the CPT of an inherited attribute.

Specializing attribute consist in changing that attribute type by a subtype. Specializing a reference slot consists in changing the reference slot range type by a subclass. Finally, either changing an attribute dependencies or its CPT is done by redeclaring the attribute.

A.7.2 Interfaces

An interface is an abstract class, i.e., it does not have any probabilistic dependencies. Interfaces are used to define multiple class inheritance: when a class implements an interface it must declare all the reference slots and attributes contained in the implemented interface. Eventually, the class can specialize them. Declaring an interface is similar to declaring a class, but uses the keyword `interface` and does only defined parameters without any default value. The following is an example of interface declaration:

```
interface I {
  boolean b;
}

interface J {
  boolean c;
}

class A implements I, J {
  bool b {[0.1, 0.9]};
  bool c dependson c {[ /* ... */ ]};
}
```

Interfaces can be used by references as their range. In such case, when the reference slots encapsulated class is instantiated, only instance of classes implementing the reference slot range can be affected to it. Finally, it is possible to defined subinterfaces using the keyword `extends`:

```
interface I {
  bool b;
}

interface J extends I {
  bool c;
}
```

Interface inheritance only allows to specialize reference slot or attributes, since there is no probabilistic information in interfaces.

Bibliography

David Allen and Adnan Darwiche. New advances in inference by recursive conditioning. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*, pages 2 – 10, 2003.

C. Anderson, P. Domingos, and D. Weld. Relational markov models and their application to adaptive web navigation. In *Proceedings of the eighth ACM SIGKDD International Conference on Knwoledge Discovery and Data Mining*, pages 143–152, Edmonton, Canada, 2002. ACM Press.

A. Arias and F. Diez. Operating with potentials of discrete variables. *International Journal of Approximate Reasoning*, 46:166–187, 2007.

S. Arnborg, D. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *Algebraic Discrete Methods*, 1987.

F. Bacchus, S. Dalmao, and T. Pitassi. Value elimination bayesian inference via backtracking search. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence*, 2003.

Olav Bangsø. *Object Oriented Bayesian Networks*. PhD thesis, Aalborg University, March 2004.

Olav Bangsø and Krisian G. Olesen. Applying object oriented bayesian networks to large medical decision support systems. In P. Doherty B. Tessem, P. Ala-Siuru and B. Mayoh, editors, *Proceedings of the 8th Scandinavian Conference on Artificial Intelligence*, pages 25–36. IOS Press, 2003.

Olav Bangsø and Pierre-Henri Wuillemin. Object oriented bayesian networks: A framework for topdown specification of large bayesian networks and repetitive structures. Technical report, Department of Computer Science, Aalborg University, 2000a.

Olav Bangsø and Pierre-Henri Wuillemin. Top-down construction and repetitive structures representation in bayesian networks. In *Proceedings of the 13th Florida Artificial Intelligence Research Society Conference*, 2000b.

Olav Bangsø, Nicolaj Sønderberg-Madsen, and Finn V. Jensen. A bayesian network framework for the construction of virtual agents with human-like behaviour. In *Proceedings of the Third European Workshop on Probabilistic Graphical Models*, pages 35–42, 2006.

Olava Bangsø, M. Julia Flores, and Finn V. Jensen. Plug & play object oriented bayesian networks. In *Proceedings of the Tenth Conference of the Spanish Association for Artificial Intelligence*, 2003.

Ole E. Barndorff-Nielsen. *Infomation and Exponential Families in Statistical Theory*. John Wiley & Sons Ltd, 1978.

Ann Becker, Reuven Bar-Yehuda, and Dan Geiger. Randomized algorithms for the loop cutset problem. *Journal of Artificial Interlligence Research*, 2000.

Jeff Bilmes and Chris Bartels. On triangulating dynamic graphical models. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence*, pages 47–56, Acapulco, Mexico, 2003.

Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1 – 23, 1993.

Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific independence in bayesian networks. In *Proceedings of the 12th Annual Conference on Uncertainty in AI*, pages 115 – 123, 1996.

Wray L. Buntime. Operations for learning with graphical models. *Journal of Artificial Interlligence Research*, 2:159–225, 1990.

Mark Chavira and Adnan Darwiche. Compiling bayesian networks with local structure. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1306 – 1312, 2005.

Mark Chavira and Adnan Darwiche. Compiling bayesian networks using variable elimination. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2443–2449, 2007.

Mark Chavira, Adnan Darwiche, and Manfred Jaeger. Compiling relational bayesian networks for exact inference. *International Journal of Approximate Reasoning*, 42: 4–20, 2006.

Morgan Chopin and Pierre-Henri Wuillemin. Optimizing the triangulation of dynamic bayesian network. In Petri Myllymäki, Teemu Roos, and Tommi Jaakkola, editors, *Proceedings of the Fifth European Workshop on Probabilistic Graphical Modelsedited*. HIIT Publications, 2010.

Stephen Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on Theory of Computing*, pages 151–158, 1971.

G. F. Cooper. Probabilistic inference using belief network is np-hard. Technical Report KSL-87-27, Medical Computer Science, Stanford University, Stanford, California, 1987.

Robert G. Cowell, A. Philip Dawid, Steffen L. Lauritzen, and David J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Statistics for Engineering and Information Science. Springer, 1999.

Fabio G. Cozman. Generalizing variable elimination in bayesian networks. In *Workshop on Probabilistic Reasoning in Artificial Intelligence*, 2000a.

Fabio G. Cozman. Credal networks. *Artificial Intelligence*, 120:199–233, July 2000b.

Fabio G. Cozman. Axiomatizing noisy-or. Technical report, In Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04, 2004.

I. Csiszàr. I-diverfence geometry of probability distributions and minimizations problems. *The annals of probability*, 3(1):146–158, 1975.

Paulo C. G. da Costa and Kathryn B. Laskey. Multi-entity bayesian networks without multi-tears. Technical report, George Mason University Department of Systems Engineering and Operations Research, April 2005a.

Paulo C. G. da Costa and Kathryn B. Laskey. Of klingons and starships: Bayesian logic for the 23rd century. In *Proceedings of the 21th Annual Conference on Uncertainty in AI*, 2005b.

Paulo C. G. da Costa and Kathryn B. Laskey. Pr-owl: A framework for probabilistic ontologies. In *Proceedings of the Fourth International Conference on Formal Ontology in Information Systems*, 2006.

Paulo C. G. da Costa, Francis Fung, Kathryn B. Laskey, Michael Pool, Masami Takikawa, and Edward J. Wright. Mebn logic: A key enabler for network centric warfare. In *10th Annual Command and Control Research and Technology Symposium*, 2005.

P. Dagum and M. Luby. Approximating probabilistic inference in bayesian belief networks is np-hard. *Artificial Intelligence*, 60(1):141–153, 1993.

A. Darwiche. Dynamic jointrees. In G.F. Cooper and S. Moral, editors, *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*, pages 97–194, San Francisco, CA, 1998. Morgan Kaufmann.

Adnan Darwiche. Recursive conditioning. *Artificial Intelligence Journal*, 125(1-2): 5–41, 2001a.

Adnan Darwiche. Constant-space reasoning in dynamic bayesian networks. *International Journal of Approximate Reasoning*, 26(3):161–178, April 2001b.

Adnan Darwiche. A differential approach to inference in bayesian networks. *Journal of the ACM*, 50(3):280 – 305, 2003.

Adnan Darwiche. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press, 2009.

Adnan Darwiche. Inference in bayesian networks: A historical perspective. In Hector Geffner Rina Dechter and Joe Halpern, editors, *Heuristics, Probability and Causality. A Tribute to Judea Pearl*. College Publications, 2010.

R. de Salvo Braz, E. Amir, and D. Roth. Lifted first-order probabilistic inference. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1319–1325, 2005.

Rodrigo de Salvo Braz. *Lifted First-Order Probabilistic Inferenc*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the 12th Annual Conference on Uncertainty in AI*, pages 211 – 219, 1996.

R. Dechter. Bucket elimination: A unifying framework for reasoning. In M. I. Jordan, editor, *Learning in graphical models*, pages 75–104. MIT Press, 1999.

F.J. Díez. Local conditioning in bayesian networks. *Artificial Intelligence*, 87:1 – 20, 1996.

Pedro Domingos and Matthew Richardson. Markov logic: A unifying framework for statistical relational learning. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

Pedro Domingos, Kok Stanley, Poon Hoifung, Richardson Matt, and Singla Parag. Unifying Logical and Statistical AI. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 2–7. MA: AAAI Press, 2006.

D. Draper. Clustering without (thinking about) triangulation. In *Proceedings of the 11th Conference on Uncertainty in Artificial Intelligence*, 1995.

M. J. Druzdzel and H. J. Suermondt. Relevance in probabilistic models : "backyards" in a "small world". Technical report, AAAI, 1994.

Armelle Fay and Jean-Yves Jaffray. A justification of local conditioning in bayesian networks. *International Journal of Approximate Reasoning*, 2000.

M. Fishelson and D. Geiger. Optimizing exact genetic linkage computations. *Journal of Computational Biology*, 11(263-275), 2004.

B. Frey. Extending factor graphs so as to unify directed and undirected graphical models. In *Proceedings of the 19th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 257–264, 2003.

Nir Friedman, Lise Getoor, Daphne Koller, and Avi Pfeffer. Learning probabilistic relational models. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.

M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979a.

M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979b.

M.R. Garey, D.S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of ACM symposium on theory of computing*, pages 47–63, 1974.

D. Geiger and C. Meek. Graphical models and exponential families. In *Proceedings of the 14th Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, 1998.

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

Lise Getoor, Nir Friedman, Daphne Koller, Avi Pfeffer, and Ben Taskar. Probabilistic relational models. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

Kevin Grant. Efficient indexing for recursive conditioning algorithms. In 537-542, editor, *Proceedings of the 23rd Florida Artificial Intelligence Research Society Conference*, 2010.

M. Halldórsson. Approximations of weighted independent set and hereditary subset problems. *Journal of Graph Algorithms and Applications*, 2000.

Joseph Y. Halpern. An analysis of first-order logics of probability. *Artificial Intelligence*, 46:311–350, 1990.

J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424, 1994.

David Heckerman, Chris Meek, and Daphne Koller. Probabilistic entity-relationship models, prms, and plate models. In L. Getoor and B. Taskar, editors, *An Introduction to Statistical Relational Learning*. MIT Press, 2007.

M. Horsch and D. Poole. A dynamic approach to probabilistic inference using bayesian networks. In *Proceedings of the sixth Conference on Uncertainty in AI*, pages 155–161, 1990.

R. A. Howard and J. E. Matheson. Influcen diagrams. In R. A. Howard and J. E. Matheson, editors, *Readings on the Principles and Applications of Decision Analysis II*, pages 719–762. Strategic Decision Group, 1984.

C. Huang and A. Darwiche. Inference in belief networks a procedural guide. *International Journal of Approximate Reasoning*, 15(3):225–263, 1996.

J. S. Ide, F. G. Cozman, and F. T. Ramos. Generating random Bayesian networks with constraints on induced width. In *Proc. of ECAI'04*, pages 323–327, 2004.

A. Inokuchi, T. Washio, and H. Motoda. A general framework for mining frequent subgraphs from labeled graphs. *Fundamenta Informaticae*, 2005.

Manfred Jaeger. Relational bayesian networks. In Morgan Kaufmann, editor, *Proceedings of UAI-97*, San Francisco, CA, 1997.

Manfred Jaeger. On the complexity of inference about probabilistic relational models. *Artificial Intelligence*, 117:297 – 308, 1999.

F.V. Jensen. Junction trees and decomposable hypergraphs. Technical report, Judex Datasystemer, Aalborg, Denmark, 1988.

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

M. I. Jordan, editor. *Learning in graphical models*. Cambridge MA: MIT Press, 1999.

M. Richard Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*. Springer, 1972.

K. Kersting and L. De Raedt. Bayesian logic programs. Technical report no. 151, Institute for Computer Science, University of Freiburg, Germany, April 2001.

J. H. Kim and J. Pearl. A computational model for combined causal and diagnostic reasoning in inference system. In *Proceedings of the 8th International Joint Conference on AI*, pages 190 – 193, 1983.

Jacek Kisynski and David Poole. Lifted aggregation in directed first-order probabilistic models. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 2009.

U. Kjærulff. Triangulation of graphs — algorithms giving small total state space. Technical Report R-90-09, Dept. of Maths and Computer Science, Aalborg University, 1990.

Uffe Kjærulff. dhugin: A computational system for dynamic time-sliced bayesian networks. *International Journal of Forecasting*, 11(1):11–89, 1994.

D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

D. Koller, D. McAllester, and A. Pfeffer. Effective bayesian inference for stochastic programs. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, 1997a.

Daphne Koller and Avi Pfeffer. Object-oriented bayesian networks. In *Proceedings of the 13th Annual Conference on Uncertainty in AI*, pages 302–313, 1997.

Daphne Koller and Avi Pfeffer. Probabilistic frame-based systems. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 580–587, 1998.

Daphne Koller, Alon Levy, and Avi Pfeffer. P-classic: A tractable probabilistic description logic. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997b.

Daphne Koller, Nir Friedman, Lise Getoor, and Ben Taskar. Graphical models in a nutshell. In Lise Getoor and Ben Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

F. Ksichischang, B. Frey, and H. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47, 2001.

M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of ICDM'01*, 2001.

J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning (ICML)*, 2001.

Kathryn B. Laskey. Mebn: A language for first-order bayesian knowledge bases. *Artificial Intelligence*, 172(2-3), 2008.

Kathryn B. Laskey, Suzanne M. Mahoney, and Ed Wright. Hypothesis management in situation-specific network construction. In *Proceedings of the 17th Annual Conference on Uncertainty in AI*, 2001.

Kathryn B. Laskey, Ghazi Alghamdi, Xun Wang, Daniel Barbara, Ed Wright Tom Shackleford, and Julie Fitzgerald. Detecting threatening behavior using bayesian networks. In *Proceedings of the Conference on Behavioral Representation in Modeling and Simulation*, 2004.

S.L. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their applications to expert systems. *Journal of the Royal Statistical Society*, 50(2):157 – 224, 1988.

Steffen L. Lauritzen. *Graphical Models*. Oxford University Press, 1996.

Vasilica Lepar and Prakash P. Shenoy. A comparison of lauritzen-spiegelhalter, hugin, and shenoy-shafer architectures for computing marginals of probability distributions. In G. F. Cooper and S. Moral, editors, *Proceedings of the 14th Annual Conference on Uncertainty in AI*. Morgan Kaufmann, 1999.

Uri N. Lerner. *Hyrbid bayesian networks for reasoning about complex systems*. PhD thesis, Stanford University, 2002.

R. Duncan Luce and Albert D. Perry. A method of matrix analysis of group structure. *Psychometrika*, 14(2), 1949.

A.L. Madsen and F.V. Jensen. LAZY propagation: A junction tree inference algorithm based on lazy inference. *Artificial Intelligence*, 113(1–2):203–245, 1999.

Suzanne M. Mahoney and Kathryn B. Laskey. Network engineering for complex belief networks. In *Proceedings of the 12th Annual Conference on Uncertainty in AI*, 1996.

Suzanne M. Mahoney and Kathryn B. Laskey. Network fragments: Representing knowledge for constructing probabilistic models. In *Proceedings of the 13th Annual Conference on Uncertainty in AI*, 1997.

Suzanne M. Mahoney and Kathryn B. Laskey. Constructing situation specific networks. In *Proceedings of the 14th Annual Conference on Uncertainty in AI*, 1998a.

Suzanne M. Mahoney and Kathryn B. Laskey. Extensible multi-entity directed graphical models: A framework for incremental model construction. In *presented at the 1998 Valencia conference on Bayesian statistics*, 1998b.

D. Malioutov, J. Johnson, and A. Willsky. Walk-sums and belief propagation in gaussian graphical models. *Journal of Machine Learning Research*, 7(20):31–64, 2006.

Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pages 1352–1359, 2005.

Brian Milch, Luke S. Zettlemoyer, Kristian Kersting, Michael Haimes, and Leslie Pack Kaelbling. Lifted probabilistic inference with counting formulas. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pages 1062–1068, 2008.

S. Muggleton. Stochastic logic programming. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 254–264. IOS Press, Amsterdam, 1996.

Kevin Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, University of California, 2002.

H. Pasula and S. Russell. Approximate inference for first-order probabilistic languages. In *Proceedings of the 17th international joint conference on Artificial intelligence*, pages 741–748, Seattle, WA, USA, 2001.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman, 1988.

J. Pearl. Influence diagrams - historical and personal perspectives. *Decision Analysis*, 2 (4):232–234, December 2005.

J. Pearl and A. Paz. Graphoids: graph-based logic for reasoning about relevance relations. In B. Duboulay, D. Hogg, and L. Steels, editors, *Advances in Artificial Intelligence-II*, pages 357–363, 1987.

Judea Pearl. Reverend bayes on inference engines. In *Proceedings Americain Association of Artificial Intelligence National conference AI*, pages 133 – 136, 1982.

Judea Pearl. A constraint-propagation approach to probabilistic reasoning. In *Uncertainty in Artificial Intelligence*, pages 357 – 369. L. Kanal and J. Lemmers (Eds.), 1986.

Judea Pearl. *Causility*. Cambridge University Press, 2009.

Avi Pfeffer. *Probabilistic Reasoning for Complex Systems*. PhD thesis, Stanford University, 1999.

Avi Pfeffer. IBAL: A probabilistic rational programming language. In *International Joint Conference on Artificial Intelligence*, 2001.

Avi Pfeffer, Daphne Koller, Brian Milch, and K.T. Takusagawa. SPOOK: A system for probabilistic object-oriented knowledge representation. In *Proceedings of the 14th Annual Conference on Uncertainty in AI*, 1999.

Dabid Poole. First-order probabilistic inference. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 985–991, 2003.

L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of IEEE*, volume 77, 1989.

L.R. Rabiner and B.H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, pages 4–15, 1986.

N. Robertson and P. D. Seymour. Graph minots ii. algorithms aspects of tree-width. *Journal of Algorithms*, 7:309 – 322, 1986.

D. Rose, G. Lueker, and R.E. Tarjan. Algorithmic aspects of vertex elimination on graphs. *SIAM J. on Computing*, 5:266–283, 1976.

D.J. Rose. Triangulated graphs and the elimination process. *J. Math. Analysis and Applications*, 1970.

R. Shachter. Evaluating influence diagrams. *Operations Research*, 34:871–882, November-December 1986.

R. Shachter. Bayes-ball: The rational pastime. In *Proceedings of the 14th Annual Conference on Uncertainty in AI (UAI)*, pages 480–487. Morgan Kaufmann, 1998.

R. Shachter and C. Kenley. Gaussian influence diagrams. *Management Science*, 35: 527–550, 1989.

Rita Sharma and David Poole. Probabilistic reasoning with hierarchically structured variables. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, 2005.

P.P. Shenoy and G. Shafer. Axioms for probability and belief-function propagation. In North-Holland, editor, *Uncertainty in Artificial Intelligence*, volume 4, pages 169–198. R. D. Shachter, T. S. Levitt, L. N. Kanal, and J. F. Lemmer, 1990.

Y. Shibata. On the tree representation of chordal graphs. *Journal of Graph Theory*, 12 (3):421–428, 1988.

P. Smyth, D. Heckerman, and M. Jordan. Probabilistic independence networks for hidden markov probability models. *Neural Computation*, 9(2):227 – 269, 1997.

T. Speed and H. Kiiveri. Gaussian markov distributions over finite graphs. *The annals of statistic*, 14(1):138–150, 1986.

M. Studeny. Condtional independence relations have no finite complete characterization. In *Proceedings of the 11th Prague Conference on Information Theory, Statistical Decision and Random Process*, pages 27–31. Springer, 1990.

H. J. Suermondt and G. F. Cooper. Probabilistic inference in multiply connected belief networks using loop cutsets. *International Journal of Approximate Reasoning*, 4(283–306), 1990.

C. Sutton and A. McCallum. Collective segmentation and labeling of distant entities in information extraction. In *ICML Workshop on Statistical Relational Learning and its Connections to other fields*, 2004.

C. Sutton and A. McCallum. An introduction to conditional random fields for relational learning. In L. Getoor and B. Taskar, editors, *Introduction to Statistical Relational Learning*. MIT Press, 2007.

Ben Taskar, P. Abbeel, and Daphne Koller. Discriminative probabilistic models for relational data. In *Proceedings of the 18th Conference on Ucerntainty in Artificial Intelligence*, 2002.

Keiji Kanazawa Thomas Dean. A model for reasoning about persistence and causation. *Computational Intelligence*, 5(2):142–150, February 1989.

M.P. Wellman, J.S. Breese, and R.P. Goldman. From knowledge bases to decision models. *The Knowledge Engineering Review*, 7(1):35–53, November 1992.

N. Wermuth. Linera recursive equations, covariance selection and path analysis. *Journal of the American Statistical Association*, 75:325–33, 1980.

Y. Xiang, B. Pant, A. Eisen, M.P. Beddoes, and D. Poole. Multiply sectioned bayesian networks for neuromuscular diagnosis. *Artificial Intelligence in Medicine*, 5:293–314, 1993a.

Yang Xiang. Generalized non-impending noisy-and trees. In *Proceedings of the 23th International Florida Artificial Intelligence Research Society Conference*, pages 555–560, 2010.

Yang Xiang, David Poole, and Michael P. Beddoes. Multiply sectioned bayesian networks and junction forests for large knowledge based systems. *Computational Intelligence*, 9(2):171–220, 1993b.

X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of ICDM'02*, 2002.

Xiang Yang. *Probabilistic Reasoning in Multi-Agent Systems: A Graphical Models Approach*. Cambridge University Press, 2002. ISBN 0521813085.

Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Understanding belief propagation and its generalizations. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, 2001.

Nevin Lianwen Zhang and David Poole. Exploiting causal independence in bayesian network inference. *Journal of Artificial Interlligence Research*, 5:301–328, 1996.

N.L. Zhang and D. Poole. A simple approach to bayesian network computation. In *proceedings of the 10th Canadian Conference on Artificial Intelligence*, pages 16–22, Banff, Alberta, Canada, 1994.