Faculty of Computer Science

Technical University of Madrid

Master Thesis

University Master in artificial intelligence

Learning Bayesian networks from data by the incremental compilation of new network polynomials

Author: Marco A. Benjumeda Barquita
Supervisors: Concha Bielza Lozoya and Pedro Larrañaga Múgica

November 2014

Resumen

Los modelos probabilísticos gráficos son un importante campo de investigación en inteligencia artificial hoy en día. El alcance de este trabajo comprende el estudio de modelos probabilísticos gráficos dedicados a la representación de distribuciones probabilísticas discretas. Dos de las mayores líneas de investigación relacionadas con esta área se centran en la aplicación de inferencia sobre modelos probabilísticos gráficos y en el aprendizaje de modelos probabilísticos gráficos a partir de datos. Tradicionalmente, el proceso de inferencia y el proceso de aprendizaje han sido tratados por separado, pero dado que la estructura de los modelos aprendidos marca la complejidad de inferencia, este tipo de estrategias produce en muchas ocasiones modelos muy ineficientes. Con el fin de obtener modelos más ligeros, en esta tesis de fin de máster se propone un nuevo modelo para la representación de polinomios de red, a la que llamamos árboles polinómicos. Los árboles polinómicos son una representación complementaria de las redes Bayesianas que permite una evaluación eficiente de la complejidad de inferencia y proporciona un marco dedicado a la inferencia exacta. También proponemos un conjunto de métodos dedicados a la compilación incremental de árboles polinómicos y un algoritmo para el aprendizaje de árboles polinómicos a partir datos, que utiliza un método voraz de búsqueda y puntuación que incluye la complejidad de inferencia como una penalización en la función de puntuación.

Abstract

Probabilistic graphical models are a huge research field in artificial intelligence nowadays. The scope of this work is the study of directed graphical models for the representation of discrete distributions. Two of the main research topics related to this area focus on performing inference over graphical models and on learning graphical models from data. Traditionally, the inference process and the learning process have been treated separately, but given that the learned models structure marks the inference complexity, this kind of strategies will sometimes produce very inefficient models. With the purpose of learning thinner models, in this master thesis we propose a new model for the representation of network polynomials, which we call polynomial trees. Polynomial trees are a complementary representation for Bayesian networks that allows an efficient evaluation of the inference complexity and provides a framework for exact inference. We also propose a set of methods for the incremental compilation of polynomial trees and an algorithm for learning polynomial trees from data using a greedy score+search method that includes the inference complexity as a penalization in the scoring function.

Acknowledgements

I want to thank my supervisors Pedro Larrañaga and Concha Bielza for their guidance during this master thesis and their confidence in me. I also want to thank my parents for their huge support, that have allowed me to focus on my work without worrying about many other things.

Contents

List of Figures

List of Tables

Abbreviations

AC	**A**rithmetic **C**ircuit
AI	**A**rtificial **I**ntelligence
AIC	**A**kaike **I**nformation **C**riterion
BD	**B**ayesian **D**irichlet
BIC	**B**ayesian **I**nformation **C**riterion
BN	**B**ayesian **N**etwork
CHC	**C**onstrained **H**ill-**C**limbing
CPD	**C**onditional **P**robability **D**istribution
CPT	**C**onditional **P**robability **T**able
DAG	**D**irect **A**cyclic **G**raph
HC	**H**ill-**C**limbing
HCPT	**H**ill-**C**limbing for **P**olyomial **T**rees
JT	**J**unction **T**ree
LL	**L**og-**L**ikelihood
LMM	**L**ight **M**utual **M**in
LW	**L**ikelihood-**W**eighting
MCMC	**M**arkov **C**hain **M**onte **C**arlo
MDL	**M**inimum **D**escription **L**ength
MMHC	**M**ax **M**in **H**ill-**C**limbing
MMPC	**M**ax **M**in **P**arents and **C**hildren
MP	**M**essage **P**assing
NMLP	**N**ormalized **M**ean **L**og **P**robability

NP	**N**etwork **P**olynomial
PC	**P**arents and **C**hildren
PLS	**P**robabilistic **L**ogic **S**ampling
PT	**P**olyomial **T**ree

Chapter 1

Introduction

1.1 Graphical models and Bayesian networks

During the last decades, there has been an information revolution due to the rise of internet. There are multiple information sources from several fields. Social data can be obtained from the web, from social networks or from official data sources. but there are also huge amounts of data from specific fields. For example, there are multiple datasets related to finance and business, including the stock market data and electronic trading. In science, there are many experimental results published online, there are huge biomedical databases available, and different sources of information for almost every scientific field.

Modelling this data is one of the main topics of *artificial intelligence (AI)* nowadays. Models are necessary to reason about the phenomenas collected in the data or to get predictions of the outcome of certain events. The models are also useful to hide the complexity of the real world, and therefore simplify the representation of the problem that we are facing. They can also be applied in situations that are different from the ones where they were learned, so the models should learn the patterns in the data and recognise similar situations.

It is important to mention that most of the domains that we need to deal with in AI involve uncertainty. Probabilistic models are specially interesting, because they handle uncertainty applying the *probability theory*, that is long-established. These models use probabilities to indicate different degrees of certainty.

Trying to store the full joint probability distribution related to a problem would be intractable, so it is necessary to use a more compact representation of the probabilistic models that is useful in practice. The idea is to take advantage of the independences between the variables of the model to avoid unnecessarily big probability tables. *Probabilistic graphical models* represent the dependences between the variables graphically, usually with a graph structure where the nodes represent the variables of the model and the arcs represent the probabilistic relationships between the variables.

Bayesian networks (BNs) are one of the most popular and successful graphical models. The dependences of the model are represented by directed arcs, so the structure of BNs is a *direct acyclic graph (DAG).* The name of Bayesian networks comes from the *Bayes* rule, which they use to compute probabilities. Another interesting property of BNs is that they show the relationships between the variables intuitively, so they can be easily consulted or modified by experts. This makes the BN easier to understand for the users.

The next are some of the properties that make BNs so widely used:

- They are probabilistic models. They represent the uncertainty with probabilities, which makes the models coherent with the probability theory.
- They are graphical models. They are easily understandable and the relationships between the variables are intuitive.
- They are self-explanatory. They do not only give the solutions to problems, but they also provide the justification.
- They favour the representation of causal relationships.
- They can be created by experts or learned from data. Learning BNs from data can be also useful to learn some undiscovered relationships from complex domains.
- There are methods for BNs that can deal with high dimensional domains and large amount of variables.

Although Bayesian networks have many desirable properties, there are also some difficulties related to them that must be considered:

- The complexity of brute force approaches for performing inference in BNs is exponential in the number of variables and it has been proved that both exact inference (Cooper, 1990) and approximate inference (Dagum and Luby, 1993) in BNs are NP-Hard.

- Learning BNs from data is an *NP-complete* problem (Chickering, 1996) and state-of-the-art methods still capture many unnecessary relationships between the variables.

Graphical models, and Bayesian networks in particular, are a huge research field in AI, and there is a great effort and tones of work dedicated to the improvement of the performance and accuracy of the inference and learning procedures.

1.2 Motivations

Traditionally, the inference and learning methods for Bayesian networks have been treated separately. Research works focused on inference methods usually try to improve their efficiency while keeping its faithfulness to the network, while the works focused on learning methods usually try to find the best fitted networks with a limited number of parameters in a tractable time.

The problem with this approach is that the learning process affects dramatically to the efficiency of performing inference in the learned model. The shape of the model is crucial for obtaining networks where the computational complexity of exact inference is tractable. It would be interesting to have algorithms that consider the inference complexity of the learned models in a way that it is possible to learn not only fitted networks, but also models where exact inference is tractable.

The objective of this work is to create a new type of graphical model to complement Bayesian networks during the learning process, so it can be used as an inference complexity indicator, and also provide a framework for exact inference. The idea is that if we can avoid the addition of those arcs that imply a huge increment of the inference complexity and do not improve enough the fitting of the network, we should obtain thinner models without worsening too much their faithfulness, allowing exact inference in many cases.

1.3 Notation

In this section we make a brief review of the notation used in this work.

The notation for mentioning graphical models is a calligraphic capital letter, that can be combined with a sub-index or a super-index. The Bayesian networks are usually represented with a capital $\mathcal{B}$ and the polynomial trees are usually represented with a capital $\mathcal{P}$. An example is the PT $\mathcal{P}'$.

The sets of variables are represented with one or more calligraphic capital letters, that can be combined with a sub-index. An example would be the set of nodes $\mathcal{X}_P$.

Each variable is represented with one or more capital letters in italic, that can be combined with a sub-index with lower-case letters or numbers. An example would be the node X_i.

The precedence dependences in the models are represented as follows:

- $Pa(M, X_i)$: Parents of the node X_i in model M.
- $Ch(M, X_i)$: Children of the node X_i in model M.
- $Pred(M, X_i)$: Predecessors of the node X_i in model M.
- $Desc(M, X_i)$: Descendants of the node X_i in model M.

When knowing which model is M is trivial, sometimes the precedence notation is simplified and M is not mentioned. For example, $Pa(X_i)$ could be mentioned instead of $Pa(M, X_i)$.

1.4 Structure of the report

- **Chapter 1** contains an introduction to the graphical models and the motivations of this work.
- **Chapter 2** contains the definition of *Bayesian networks (BNs)*, and reviews some of the most relevant methods for inference and learning in BNs.
- **Chapter 3** is an introduction to *network polynomials (NPs)*. It includes the definition of *arithmetic circuits (ACs)* and reviews some methods for learning ACs from data.

- **Chapter 4** contains the definition of the *polynomial trees (PTs)*, the new model proposed for representing network polynomials. It includes the methods proposed for the incremental compilation and optimization of PTs, and a method for learning PTs from data.
- **Chapter 5** consists of a set of experimental tests over the proposed PT learning algorithm.
- **Chapter 6** includes the conclusions of this master thesis and discusses about possible future research work related to PTs.

Chapter 2

Bayesian Network Models

2.1 Bayesian networks

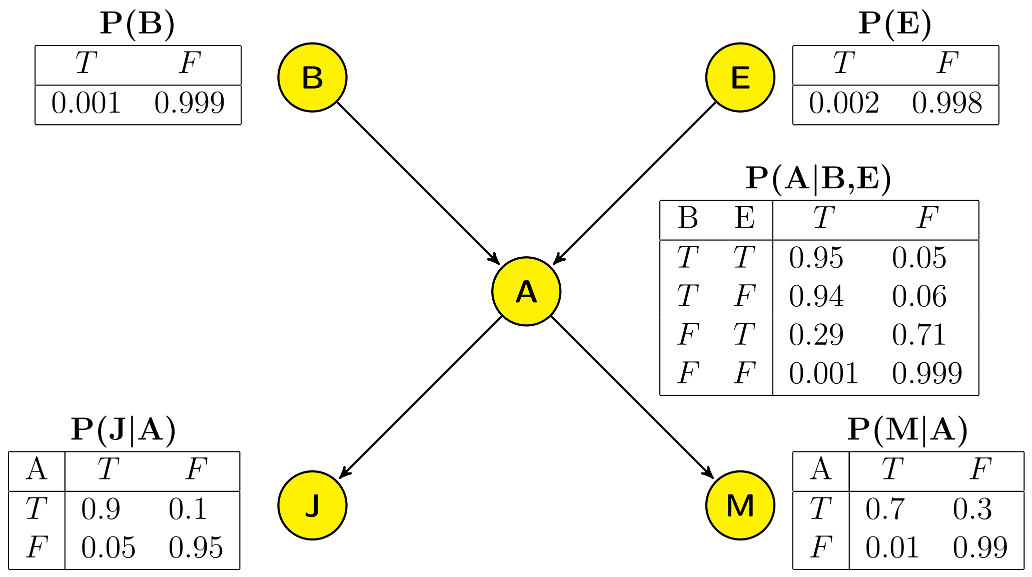
Bayesian networks are very powerful tools for modelling probability distributions. They belong to the family of probabilistic graphical models, and specifically, each BN represents a probability distribution over a set of variables $\mathcal{X} = \{X_1, X_2, \dots, X_n\}$.

A BN encodes the conditional dependences between the variables in $\mathcal{X}$, and it is composed of:

1. *Directed acyclic graph*: Each node of the graph represents a random variable in $\mathcal{X}$, and each directed arc $X_i \rightarrow X_j$ represents the conditional dependence between X_i and X_j.

2. *Parameters*: Each variable $X_i \in \mathcal{X}$ has a conditional probability distribution $P(X_i, Pa(X_i))$. This probability distributions are called the parameters of the network. The parameters of the network can be represented in multiple ways. For discrete networks, *conditional probability tables (CPTs)* are the most common choice. Gaussians (Geiger and Heckerman, 1994; Shachter and Kenley, 1989; Lauritzen and Spiegelhalter, 1988; Chevrolat et al., 1994) have been used for continuous networks.

The BN shown in Figure 2.1 is called Earthquake (Korb and Nicholson, 2003), and here we use it as a simple example of the BNs structure and functioning. This network models the behaviour of an alarm that can be activated by burglars, but also has a small probability of being activated by an earthquake, and the reactions of



P(B)

T	F
0.001	0.999

P(E)

T	F
0.002	0.998

P(A|B,E)

B	E	T	F
T	T	0.95	0.05
T	F	0.94	0.06
F	T	0.29	0.71
F	F	0.001	0.999

P(J|A)

A	T	F
T	0.9	0.1
F	0.05	0.95

P(M|A)

A	T	F
T	0.7	0.3
F	0.01	0.99

FIGURE 2.1: Bayesian network *Earthquake*

John and Mary (if they call or not) to the activation of the alarm. All the variables in the BN are Boolean, and they represent the next events:

- B: There is a burglary.
- E: There is an earthquake.
- A: The alarm activates.
- J: John calls.
- M: Mary calls.

As the network domain is discrete, the parameters are represented in CPTs (Figure 2.1). For example, it is easy to see that the probability of activation of the alarm is higher if there is a burglary and there is not an earthquake ($P(A = T|B = T, E = F) = 0.94$) than if there is an earthquake and there is not a burglary ($P(A = T|B = F, E = T) = 0.29$), or that there is more probability that John calls ($P(J = T|A = T) = 0.9$) than that Mary calls ($P(M = T|A = T) = 0.7$) if the alarm is activated. The concept of *conditional independence* is essential to obtain information from BNs. It is defined below:

Definition 1. ***(Conditional independence):*** *Given a probability distribution $\mathcal{P}$, two random variables X_a and X_b are conditionally independent given another random variable X_c if and only if:*

$$P(X_a, X_b|X_c) = P(X_a|X_c) \cdot P(X_b|X_c)$$

We use $(X_a \perp X_b | X_c)_P$ to denote that in $\mathcal{P}$, X_a and X_b are conditionally independent given X_c.

It is intuitive to see some of the dependences between the variables of the network. For example, it is simple to see that A depends on B, or that J depends on A, but there are some properties of BNs that allow assuming conditional independences among the variables of the network, which is extremely useful in practice. An interesting property of the BNs is the *Markov blanket* (Pearl, 1988).

Definition 2. ***(Markov blanket):*** *Let $\mathcal{B}$ be a BN over $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$. The Markov blanket $MB(X_i)$ of any node $X_i \in \mathcal{X}$ in $\mathcal{B}$ is the set of nodes composed by the parents of X_i, its children, and the parents of its children. Furthermore, any node is conditionally independent of the rest when conditioned on $MB(X_i)$.*

Although previous works have described sets of axioms that help finding the independences encoded in a BN, as the ones described in Pearl (1995), it is much simpler to use the independence statements from the topological properties of directed graphs, and in particular the concept of *d-separation* (Pearl, 1988; Geiger et al., 1990; Pearl, 1995).

Definition 3. ***(d-separation):*** *Let $\mathcal{X}_A$, $\mathcal{X}_B$ and $\mathcal{X}_C$ be three disjoint sets of nodes in a DAG $\mathcal{G}$. Let $\mathcal{T}$ be the set of possibles trials from any node $X_a \in \mathcal{X}_A$ to any node $X_b \in \mathcal{X}_B$, where a trial in the network is a succession of arcs in $\mathcal{G}$, no matter their directions. Then $\mathcal{X}_C$ blocks a trial $T_I \in \mathcal{T}$ if one of the following holds:*

1. *T_I contains a chain $T_{i-1} \rightarrow T_i \rightarrow T_{i+1}$ such that $T_i \in \mathcal{X}_C$.*

2. *T_I contains a fork $T_{i-1} \leftarrow T_i \rightarrow T_{i+1}$ such that $T_i \in \mathcal{X}_C$.*

3. *T_I contains a collider $T_{i-1} \rightarrow T_i \leftarrow T_{i+1}$ such that T_i and any of its descendants do not belong to $\mathcal{X}_C$.*

If all the trials in $\mathcal{T}$ are blocked by $\mathcal{X}_C$, then $\mathcal{X}_C$ d-separates $\mathcal{X}_A$ and $\mathcal{X}_B$, which is expressed by $(\mathcal{X}_A \perp \mathcal{X}_B | \mathcal{X}_C)_\mathcal{G}$.

It is convenient to know the relation between the topological properties of directed graphs and the independences of its underlying probability distribution. This relation is defined by the concept of *I-map* (Pearl, 1988).

Definition 4. ***(I-map):*** *Let $\mathcal{X}_A$, $\mathcal{X}_B$ and $\mathcal{X}_C$ be three disjoint sets of nodes in a DAG $\mathcal{G}$. Then $\mathcal{G}$ is an I-map of a probability distribution $\mathcal{P}$ if:*

$$(\mathcal{X}_A \perp \mathcal{X}_B | \mathcal{X}_C)_{\mathcal{G}} \implies (\mathcal{X}_A \perp \mathcal{X}_B | \mathcal{X}_C)_{\mathcal{P}}$$

This means that if $\mathcal{G}$ is an I-Map of $\mathcal{P}$, if two set of nodes $\mathcal{X}_A$ and $\mathcal{X}_B$ are d-separated by another set of nodes $\mathcal{X}_C$ in $\mathcal{G}$, then $\mathcal{X}_A$ and $\mathcal{X}_B$ are conditionally independent in $\mathcal{P}$. This concept allows a formal definition of BNs.

Definition 5. ***(Bayesian network):*** *Let $\mathcal{P}$ be a probability distribution over a set of variables $\mathcal{X}$, then a Bayesian network $\mathcal{B}$ is composed of a DAG $\mathcal{G}$ and a set of conditional probability distributions such that:*

- *Every node X_i in $\mathcal{G}$ represents a variable in $\mathcal{X}$, and has a conditional probability distribution $P(X_i|Pa(X_i))$ associated to it.*
- *$\mathcal{G}$ is a minimal I-map of $\mathcal{P}$. That is no arcs can be removed from $\mathcal{G}$ without negating the I-map property.*

2.2 Inference in Bayesian networks

The BN model is a complete representation of probability distributions that includes all the variables and their relationships in the model. This allows the calculation of the probability of conditional queries involving any variable of the network.

One of the main objectives of probabilistic models is to answer varied probability queries successfully. For this task it is necessary to perform some kind of reasoning. BNs can deal with diverse problems, and they support deductive, inductive and abductive reasoning. Deductive inference consists of obtaining conclusions from some given events. Inductive reasoning starts from some known events and looks for the causes of these events. Abductive inference tries to find the most likely hypothesis for the given observations.

The most common reasoning problems in BNs are the next:

- Prediction and diagnosis: The inference process for deductive and inductive reasoning in BNs is usually called *probability propagation* or *belief updating*. It consists of obtaining the posterior probability $P(\boldsymbol{Q}|\boldsymbol{e})$ of a set of query variables $\boldsymbol{Q}$ conditioned to a set of evidences $\boldsymbol{e}$.
- Maximum a posteriori (MAP): It is an abduction problem, usually called partial abduction. It consists of searching the most probable configuration of a set of variables in a BN given an evidence.

- Most probable explanation (MPE): It is an abduction problem, usually called total abduction. It consists of searching the most probable configuration of all variables not instantiated in a BN given an evidence. It is a particular case of the MAP problem.

The most desirable methods for probability propagation are those that allow obtaining the exact value of the probability $P(\boldsymbol{Q} = \boldsymbol{q}|\boldsymbol{E} = \boldsymbol{e})$ given the structure and parameters of a BN. This type of inference is called *exact inference*. There are many cases where exact inference is intractable or where an small error in the answers can be handled. Here, *approximate inference* is usually applied to reduce the computational cost of inference. It consists of obtaining approximate answers that include an error regarding to the BN. The most widely used approximate methods are the stochastic methods, that sample from the network to get an estimation of the result.

In this section we describe some of the most popular methods for both exact and approximate inference.

2.2.1 Exact inference

Performing inference directly from BNs using a brute force approach is extremely expensive computationally, so it is rarely used in practice. There are different methods that try to exploit the factorization encoded in the network.

One of the most popular methods is the *message passing (MP)* algorithm (Pearl, 1986). The MP algorithm is very efficient for the propagation of probabilities in polytrees. The main problem of this method is that it only works for polytrees, and these models are usually not enough for the representation of the knowledge in many real-world domains.

The most widely used approach for performing inference in BNs that are not polytrees is clustering. The main idea of these methods is to compile the network using a clustering technique to group the nodes in a way that the resultant structure is a polytree, and then perform the MP algorithm to the new model. The secondary structures obtained after the compilation of the BNs is another factorization of the joint probability distribution encoded by the BNs, and they are usually called *junction trees (JTs)*.

The JT algorithm was introduced by Lauritzen and Spiegelhalter (1988), but there are many different approaches based on this method. The structure of these algorithms is similar, and in most cases they follow the next steps:

1. Obtain the moral graph from the BN.

   First, the nodes with common parents in the BN are joined with a moral link, so the dependences that would be lost with the transformation of the DAG into an undirected graph are kept.

2. Triangulate the moral graph.

   This is a crucial phase, and there are multiple ways of proceeding in this step. In the triangulation a *chord* is introduced in all the cycles with a length bigger than three. These edges are called *fill-ins*.

3. Identify the cliques in the triangulated graph.

   The nodes of the secondary structure, that are formed for a group of nodes of the BN, are called *cliques*. In this step the *maximal complete subgraphs* should be identified in the triangulated graph. The identification of the cliques is dependent on the triangulation process.

4. Create the junction tree.

   The actions necessary for the creation of a valid JT from a set of cliques are the identification of the separators of the JT and the connection of the cliques.

5. Compute the new parameters.

   Lastly, the new CPDs should be computed according to the JT structure using the parameters of the BN.

A different approach for the compilation of BNs was proposed in Darwiche (2003). It is based on the representation of the network polynomials that are implicit in the BNs as *arithmetic circuits*, a useful framework for the graphical representation of polynomials. This work is reviewed in more detail in Chapter 3.

2.2.2 Approximate inference

The purpose of approximate inference is to reduce the computational complexity of the inference process in BNs, that is usually intractable for relatively large networks. The main disadvantage of these methods is that they add an error to the results.

The answers returned by approximate methods are an estimation of the real values. The most popular approximate methods are stochastic. In the stochastic methods the returned values are basically estimated by first, generating samples from the BN, and second, computing the probability of the query from the generated samples. The stochastic methods are based in the Law of large numbers, that states that the estimation should converge to the probability as the number of generated samples grows.

If we imagine that a conditional query $P(\boldsymbol{Q} = \boldsymbol{q}|\boldsymbol{E} = \boldsymbol{e})$ has been asked to the network, then the most obvious way of getting an approximate response with a stochastic approach is to generate samples from the BN, and then obtain the answer by $N_{qe} \div N_e$, where N_{qe} is the number of samples where $\boldsymbol{Q} = \boldsymbol{q}$ and $\boldsymbol{E} = \boldsymbol{e}$, and N_e is the number of samples where $\boldsymbol{E} = \boldsymbol{e}$. This is the general functioning of stochastic inference methods. *Probabilistic logic sampling (PLS)* (Henrion, 1988) is an approximate inference method that proceeds in this way, but using an ancestral ordering of the variables to sample from the BN. The PLS algorithm is described in Algorithm 2.1.

The main problem of the PLS algorithm is that all the samples that do not match with evidence $\boldsymbol{e}$ are rejected, which in practice could suppose that, in order to get a satisfactory convergence, a huge amount of samples should be generated. This makes the PLS algorithm intractable or unnecessarily slow in many cases. The *likelihood weighting (LW)* algorithm (Fung and Chang, 1989; Shachter and Peot, 1989) overcomes this difficulty by generating weighted samples that always match with evidence $\boldsymbol{e}$. Each sample and its weight is obtained using the LW particle generation procedure, and then the value of $P(\boldsymbol{q}|\boldsymbol{e})$ is calculated using the weight of the samples. Algorithm 2.2 describes the likelihood weighting algorithm for m samples.

Markov chain Monte Carlo (MCMC) methods are one of the most popular approaches for sampling from probability distributions. An stochastic process has the Markov property if in each iteration the future states depend only on the current state, so they are conditionally independent from the past states given the present

Algorithm 2.1 $PLS(\mathcal{B}, \boldsymbol{q}, \boldsymbol{e}, m)$

Input: BN $\mathcal{B}$ over $\mathcal{X}$, query variables $\boldsymbol{Q} = \boldsymbol{q}$, evidence $\boldsymbol{E} = \boldsymbol{e}$, number of samples m

Output: Approximate probability $P(\boldsymbol{q}|\boldsymbol{e})$

```
 1: let $X_1, \ldots, X_n$ be a topological ordering of $\mathcal{X}$
 2: let $smpl$ be an empty list
 3: for $i = 1, \ldots, m$ do
 4:     for $X_j$ in $\{X_1, \ldots, X_n\}$ do
 5:         let $\pi_{X_j}$ be the configuration of $Pa(X_j)$ in iteration $i$
 6:         Generate $x_{ij} \sim X_j|\pi_{X_j}$
 7:     end for
 8:     append $(x_{i1}, x_{i2}, \ldots, x_{in})$ to $smpl$
 9: end for
10: let $N_e$ be the number of samples in $smpl$ where $\boldsymbol{E} = \boldsymbol{e}$
11: let $N_{qe}$ be the number of samples in $smpl$ where $\boldsymbol{E} = \boldsymbol{e}$ and $\boldsymbol{Q} = \boldsymbol{q}$
12: $P(\boldsymbol{q}|\boldsymbol{e}) = N_{qe} \div N_e$
13: return $P(\boldsymbol{q}|\boldsymbol{e})$
```

Algorithm 2.2 $LW(\mathcal{B}, \boldsymbol{q}, \boldsymbol{e}, m)$

Input: BN $\mathcal{B}$ over $\mathcal{X}$, query variables $\boldsymbol{Q} = \boldsymbol{q}$, evidence $\boldsymbol{E} = \boldsymbol{e}$, number of samples m

Output: Approximate Probability $P(\boldsymbol{q}|\boldsymbol{e})$

```
 1: let $X_1, \ldots, X_n$ be a topological ordering of $\mathcal{X}$
 2: let $smpl$ be an empty list
 3: let $wl$ be an empty list
 4: for $i := 1, \ldots, m$ do
 5:     $w := 1$
 6:     for $X_j$ in $\{X_1, \ldots, X_n\}$ do
 7:         let $\pi_{X_j}$ be the configuration of $Pa(X_j)$ in iteration $i$
 8:         if $X_j \in E$ then
 9:             let $x_{ij}$ be the value of $X_j$ in $\boldsymbol{e}$
10:             $w := w \cdot P(x_{ij}|\pi_{X_j})$
11:         else
12:             Generate $x_{ij} \sim X_j|\pi_{X_j}$
13:         end if
14:     end for
15:     append $(x_{i1}, x_{i2}, \ldots, x_{in})$ to $smpl$
16:     append $w$ to $wl$
17: end for
18: $P(\boldsymbol{q}|\boldsymbol{e}) := \frac{\sum_{i=1}^{m} wl[i] \cdot I(smpl[i]\langle \boldsymbol{Q} \rangle = \boldsymbol{q})}{\sum_{i=1}^{m} wl[i]}$    ▷ $I = 1$ if $smpl[i]\langle \boldsymbol{Q} \rangle = \boldsymbol{q}$ and $I = 0$ otherwise
19: return $P(\boldsymbol{q}|\boldsymbol{e})$
```

state. The mechanism of MCMC methods consist of a *Markov chain* that is built in a way that it spends more time in the most important regions of the distribution, so they can sample successfully from complex probability distributions.

Gibbs sampling is a MCMC method that is specially useful for sampling the posterior distribution in BNs (Hrycej, 1990). It is a special case of the Metropolis Hastings algorithm (Metropolis et al., 1953; Hastings, 1970) that is simple to apply when the conditional distribution of each variable is easy to sample, which is the case of BNs. The method is divided in a warm-up period, used to converge to the target distribution, and a sampling period, where the useful samples are generated. Algorithm 2.3 applies the *Gibbs* sampling method to answer any conditional query $P(\boldsymbol{q}|\boldsymbol{e})$ in a BN $\mathcal{B}$ using m_w as the number of warm-up samples and m as the number of useful samples.

Algorithm 2.3 *Gibbs Sampling*$(\mathcal{B}, \boldsymbol{q}, \boldsymbol{e}, m, m_w)$

```
Input: BN $\mathcal{B}$ over $\mathcal{X} = \{X_1, \ldots, X_n\}$, query variables $\boldsymbol{Q} = \boldsymbol{q}$, evidence $\boldsymbol{E} = \boldsymbol{e}$,
       Number of samples $m$, Warm-up samples $m_w$
Output: Approximate Probability $P(\boldsymbol{q}|\boldsymbol{e})$
 1: let smpl be an empty list
 2: let $s_0$ be a random sample of $B$
 3: for $i := 1, \ldots, m + m_w$ do
 4:     for $X_j$ in $\{X_1, \ldots, X_n\}$ do
 5:         if $X_j$ is in $\boldsymbol{E}$ then
 6:             let $x_{ij}$ be the value of $X_j$ in $\boldsymbol{e}$
 7:         else
 8:             let $\pi_{X_j}$ be the configuration of $Pa(X_j)$ in iteration $i-1$ if $i \neq 1$,
                or in $s_0$ otherwise
 9:             Generate $x_{ij} \sim X_j|\pi_{X_j}$
10:         end if
11:     end for
12:     if $i > m_w$ then
13:         append $(x_{i1}, x_{i2}, \ldots, x_{in})$ to smpl
14:     end if
15: end for
16: let $N$ be the number of samples in smpl
17: let $N_q$ be the number of samples in smpl where $\boldsymbol{Q} = \boldsymbol{q}$
18: return $\frac{N_q}{N}$
```

The main problem of approximate methods is that it is complex to estimate the convergence of the samples to the target probability distributions, so many times the number of samples is predefined. If the number of samples is too large the algorithm can be extremely expensive, and if it is too small the method will not converge to the target probability distribution.

2.3 Scoring metrics

The scoring function has always a mayor impact in the process of learning the structure of BNs with score + search methods. Many different metrics have been proposed over the years, but most of them can be classified as Bayesian metrics or information-theory metrics. Basically, the methods belonging to the first type try find the network that maximizes the posterior probability distribution conditioned to the available data, and those belonging to the information theory metrics base the search on the data compression that can be achieved over the different candidate networks. Carvalho (2009) compares the performance of some methods belonging to both groups of metrics. The results do not show big differences, but in general the results are slightly better for Bayesian scoring functions in large datasets and for information-theory functions in smaller datasets.

While searching for the structure of a BN, the scoring function must evaluate multiple candidate networks, and when facing large datasets or networks the evaluation process is computationally expensive. Therefore, an essential property of the scoring functions is their decomposability. A scoring function is decomposable if the score can be expressed as the sum of a set of scores that depend only on a variable and its parents. If the metric is decomposable, a local change in the network supposes that the metric must compute the evaluation function only for the nodes involved in the change. This supposes a huge improvement in the efficiency of the evaluation process.

Next, we will consider some of the most widely used metrics belonging to both Bayesian and information-theory metrics. All the scoring functions displayed below are decomposable. These metrics are essential for the global purpose of this work, and in Chapter 4 we introduce a metric adapted to our method that is based on some of the scoring functions reviewed in this section. The notation used to define the metrics in this chapter is introduced next:

D: Dataset.

$\mathcal{X}$: Set of variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$.

$\mathcal{B}$: Bayesian network over $\mathcal{X}$.

x_{ik}: Value k of variable X_i.

w_{ij}: Configuration j of the parents of X_i.

θ_{ijk}: Parameter for the k-th state of X_i conditioned to w_{ij}, i.e., $P(X_i = x_{ik}|Pa(X_i) = w_{ij})$.

r_i: Number of states of variable X_i.

q_i: Number of possible configurations of the parents of X_i.

N_{ijk}: Number of samples of D where $Pa(X_i)$ are in their j-th configuration and X_i is in its k-th state.

N_{ij}: $\sum_{k=1}^{r_i} N_{ijk}$.

N: Number of samples of D.

N'_{ijk}: Exponents of the Dirichlet prior of θ_{ijk}.

N'_{ij}: $\sum_{k=1}^{r_i} N'_{ijk}$.

N': Equivalent sample size.

2.3.1 Bayesian metrics

The Bayesian scoring functions evaluate each DAG structure by computing the posterior probability distribution $P(\mathcal{B}|D)$ given a prior distribution over the possible networks conditioned on the data. The next ones are some of the main Bayesian score functions.

The *Bayesian Dirichlet (BD)* scoring function (Heckerman et al., 1995) faces the optimization problem by making five assumptions related to the user's prior knowledge and the database.

1. The first assumption is called *multinomial sample*. It means that the data can be partitioned into multinomial samples given the structure of $\mathcal{B}$. In other words, it considers that the data is exchangeable, so if an instance of the data is substituted by another instance, the new sample has the same probability as the old one.

2. The second assumption is about *parameter independence.* It means that the parameters related to each variable X_i are independent (*global parameter independence*), and also the parameters related to each configuration of the parents of X_i are independent (*local parameter independence*).

3. Third, it assumes *parameter modularity.* It means that the density of the parameters depends only on a variable and its parents.

4. Forth, it assumes that the parameters have a *Dirichlet* distribution. The Dirichlet distributions are desirable as priors because they are closed under multinomial sampling. This kind of distributions conjugate the prior distribution of the categorical and the multinomial distributions, so if a prior distribution is Dirichlet, the posterior, given a multinomial or categorical sample is also Dirichlet.

5. The last assumption is that there is *complete data.* Although this assumption requires a complete database, in Heckerman et al. (1995) it is suggested that incomplete data would not suppose a big obstacle for using the BD function, given that there are many methods that can handle missing data in practice that could be applied in combination with the BD metric. Some examples used to handle missing data are the use of the EM algorithm (Dempster et al., 1977), or Gibbs sampling (Yi and Li, 2011).

Thus, the BD function is:

$$P(\mathcal{B}, D) = P(\mathcal{B}) \times \prod_{i=1}^{n} \prod_{j=1}^{q_i} \left(\frac{\Gamma(N'_{ij})}{\Gamma(N_{ij} + N'_{ij})} \times \prod_{k=1}^{r_i} \frac{\Gamma(N_{ijk} + N'_{ijk})}{\Gamma(N'_{ijk})} \right) \tag{2.1}$$

Where N'_{ijk} are the hyperparameters for the Dirichlet priors of the parameters given the network structure, and Γ is the gamma function.

The BD metric requires the specification of all N_{ijk}, which is intractable in many cases, making the BD metric useless in practice many times. Nevertheless, there are several scoring functions that are a particular case of the BD metric that have proven to be effective, overcoming this difficulty. The BDe is an specific case of the BD scoring function. With the purpose of making the BD function more useful in practice, the BDe metric includes two extra assumptions.

6. The first one is the *likelihood equivalence*; i.e., two DAGs are equivalent if they encode the same joint probability distribution.

7. The second one is the *structure possibility*. It assumes that for any DAG $\mathcal{G}$, its probability is greater than 0 ($P(\mathcal{G}) > 0$).

These assumptions, in combination with the other five assumptions made in the BD metrics, make the BDe metric more tractable in practice than its predecessor, but it keeps requiring some knowledge that is not simple to find. The function that represents the BDe function is the same than the used for BD, but with the assignment $N'_{ijk} = N' \cdot P(X_i = x_{ik}, Pa(X_i) = w_{ij}|\mathcal{G})$, where N' is a parameter representing the equivalent sample size for the domain. The equivalent sample size is the sum of all the hyperparameters of the Dirichlet prior distribution conditioned to the network structure.

Another popular Bayesian metric is the K2 (Cooper and Herskovits, 1991, 1992), that is simpler than the previous two. The K2 is also a specific case of the BD function. In particular, it is the result of assigning $N'_{ijk} = 1$ in the BD metric. Equation (2.2) represents the *K2* scoring function.

$$P(\mathcal{B}, D) = P(\mathcal{B}) \times \prod_{i=1}^{n} \prod_{j=1}^{q_i} \left(\frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \times \prod_{k=1}^{r_i} (N_{ijk})! \right) \tag{2.2}$$

2.3.2 Information-theory metrics

This kind of metrics are based in the use of a measure of the data compression of the dataset D obtained with a DAG $\mathcal{G}$, that represents the structure of the learned BN. The main idea of this kind of methods is to find the optimal BN $\mathcal{B}$ that encodes the data D.

One way of obtaining the optimal compression of D given $\mathcal{B}$ is maximizing the *log-likelihood (LL)* of $\mathcal{B}$ conditioned to D. The LL scoring function is defined as follows:

$$LL(\mathcal{B}|D) = \sum_{i=1}^{n} \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} N_{ijk} \log \left(\frac{N_{ijk}}{N_{ij}} \right) \tag{2.3}$$

As it is shown in Carvalho (2009), adding an arc to $\mathcal{B}$ never decreases the LL value, so the use of this function alone usually causes overfitting in the learned model. There are a fair number of scoring metrics based on the LL function that include a penalty term for the complexity of the network. Equation (2.4) represents this kind of metrics.

$$LP(\mathcal{B}|D) = \sum_{i=1}^{n}\sum_{j=1}^{q_i}\sum_{k=1}^{r_i} N_{ijk}\log\left(\frac{N_{ijk}}{N_{ij}}\right) - Penalty(\mathcal{B}, D) \tag{2.4}$$

Some of the most popular scoring metrics that use this strategy are the *minimum description length (MDL)* (Bouckaert, 1993; Lam and Bacchus, 1994), the *Bayesian information criterion (BIC)* and the *Akaike information criterion (AIC)* (Akaike, 1974). These functions use $|\mathcal{B}|$, that is, the number of parameters of $\mathcal{B}$, as a measure of the complexity of the network. $|\mathcal{B}|$ is defined by:

$$|\mathcal{B}| = \sum_{i=1}^{n}(r_i - 1)q_i \tag{2.5}$$

On the one hand, when scoring BNs with the BIC function, that is based on the Schwarz Information Criterion (Schwarz, 1978), coincides with the MDL score (De Campos, 2006). The penalty term for both MDL and BIC functions is given by:

$$Penalty(\mathcal{B}, D) = \frac{1}{2}\log(N)|\mathcal{B}| \tag{2.6}$$

On the other hand, the AIC score is slightly different, and its penalty term is represented by:

$$Penalty(\mathcal{B}, D) = |\mathcal{B}| \tag{2.7}$$

An interesting feature of the *information-theory* metrics is that the network likelihood and the complexity penalty are obtained independently, and the value of both of them are computed explicitly. As we will see in the further chapters, this property makes it easier to combine the original scoring function with a new complexity measure of the network.

2.4 Learning the structure of Bayesian networks

In the past years there has been a huge interest in the creation of new methods for learning the structure of BNs from data. Although many different algorithms have been proposed, there are three main approaches that include most of them. The first one is to consider the learning process as a constrain satisfaction problem, trying to get the conditional independences between the variables by using a statistical hypothesis test, and then selecting the model that fits better the dependences and independences obtained in the tests. Two methods that belong to this family are the *parents and children (PC)* (Spirtes et al., 2000) and the *light mutual min (LMM)* (Mahdi and Mezey, 2013) algorithms. These techniques do not use an explicit score metric to test the likelihood between the network and the data, and instead they use statistical tests to get the skeleton of the network and then they orientate the edges by recovering the v-structures ($X_u \to X_v \leftarrow X_w$) of the network.

The second approach treats the learning process as an optimization problem. These methods, that are called score+search methods, search the BN structure that maximizes a scoring function given the available data. The most popular techniques used for learning the structure of BNs in the space of DAGs are greedy. These methods use heuristic information of the subsequent states in each step of their procedure, and they can be computationally expensive for big sets of variables. An example is the K2 algorithm (Cooper and Herskovits, 1991, 1992), that goes through each node in a predefined order, adding the best parent until no more improvements can be made or a threshold is reached. Other popular greedy methods are those using *hill-climbing (HC)* to solve the optimization problem. HC methods explore the search space in a finite number of steps starting from an initial solution. In each step the algorithm considers local changes, selecting the best solution, and it stops when none of the new solutions improves the current one.

The HC method, and unconstrained learning methods for learning BNs in general, are super-exponential in the number of variables of the learned model. This supposes that for big or high-dimensional datasets the learning process is intractable. There are some approaches that include constraints in the search process. This is the case of the *constrained hill-climbing (CHC)* algorithm (Gámez and Puerta, 2005), that is described in Algorithm 2.4. It restricts progressively the number of neighbours to be explored and evaluated, and uses a set of forbidden parents associated to each node that are initialized to empty at the beginning and updated during the search process depending on the metric difference for each local change tested. The main problem of the CHC method is that the use of the forbidden parents list can cause an early convergence to a local optimum, and it does not assure the return of a minimal I-map. The algorithm receives as an input an initial BN provided by the user, which structure can be a graph without arcs if we are going to learn the network from scratch, and it assumes that the scoring function must be maximized.

Algorithm 2.4 $CHC(\mathcal{B}, D)$

```
Input: BN $\mathcal{B}$ over $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, Data $D$
Output: BN $\mathcal{B}'$
1: let $S_{old}$ be a list over $i = 1, 2, \ldots, n$ such that $S_{old}[i] = score(X_i, Pa(X_i), D)$
2: let $\mathcal{B}'$ be a copy of $\mathcal{B}$
3: let $FP$ be a list of $n$ empty lists                ▷ Forbidden parents list
4: $OKToProceed := True$
5: while $OKToProceed = True$ do
6:                         ▷ The method $bestPred_{CHC}$ is defined in Algorithm 2.5
7:     let $\mathcal{B}_{new}$ and $S_{new}$ be the returned values of $bestPred_{CHC}(\mathcal{B}, D, S_{old}, FP)$
8:     if $\sum_{i=1}^{n} S_{new}[i] > \sum_{i=1}^{n} S_{old}[i]$ then
9:         $\mathcal{B}' \leftarrow \mathcal{B}_{new}$
10:        $S_{old} \leftarrow S_{new}$
11:    else
12:        $OKToProceed := False$
13:    end if
14: end while
15: return $\mathcal{B}'$
```

The *iCHC* and *2iCHC* algorithms (Gámez et al., 2011) use an iterative procedure where CHC is applied multiple times, allowing a more probable convergence to a global optimum and assuring the return of a minimal I-map. Algorithm 2.6 describes the 2iCHC method.

An alternative to the greedy score+search methods are the stochastic search methods. They usually reduce the computational cost of the learning process, but the results that they provide are not always consistent. Simulated annealing is widely used because it adds some intelligence to the stochastic search, trying to reach satisfactory global solutions that are close the global optimum. Wang et al. (2004) used the parallel two-level simulated annealing (Xue, 1993) method for learning BNs. This algorithm uses two levels for each candidate, a lower level and an upper level. The changes on the solutions at each step are made on the upper level. The lower level purpose is to improve the local optimization, so the acceptance of the new solutions depends on the value of the lower level objective function.

Algorithm 2.5 $bestPred_{CHC}(\mathcal{B}, D, S_{old}, FP)$

Input: BN $\mathcal{B}$, Data D, score S_{old}, Forbidden parents FP

Output: Best BN $\mathcal{B}_{new}$, Best score S_{best}

```
1: let changes be the list of local changes that could be made to $\mathcal{B}$
2: let $l$ be the length of the list of scores $S_{old}$
3: let $\mathcal{B}_{best}$ be a copy of $\mathcal{B}$
4: let $S_{best}$ be a copy of $S_{old}$
5: for change in changes do
6:     let $\mathcal{B}_{new}$ be a copy of $\mathcal{B}$
7:     let $S_{new}$ be a copy of $S_{old}$
8:     if change is the addition $X_a \to X_b$ then
9:         add $X_a$ to $Pa(\mathcal{B}_{new}, X_b)$
10:        $S_{new}(X_b) \leftarrow score(X_b, Pa(X_b), D)$
11:        if $S_{new}(X_b) < S_{old}(X_b)$ then
12:            add $X_b$ to $FP(X_a)$
13:            add $X_a$ to $FP(X_b)$
14:        end if
15:    else if change is the deletion of $X_a \to X_b$ then
16:        delete $X_a$ from $Pa(\mathcal{B}_{new}, X_b)$
17:        $S_{new}(X_b) \leftarrow score(X_b, Pa(X_b), D)$
18:        if $S_{new}(X_b) > S_{old}(X_b)$ then
19:            add $X_b$ to $FP(X_a)$
20:            add $X_a$ to $FP(X_b)$
21:        end if
22:    else if change is the reversal of $X_a \to X_b$ then
23:        delete $X_a$ from $Pa(\mathcal{B}_{new}, X_b)$
24:        add $X_b$ to $Pa(\mathcal{B}_{new}, X_b)$
25:        $S_{new}(X_b) \leftarrow score(X_b, Pa(X_b), D)$
26:        $S_{new}(X_a) \leftarrow score(X_a, Pa(X_a), D)$
27:        if $S_{new}(X_b) < S_{old}(X_b)$ or $S_{new}(X_a) > S_{old}(X_a)$ then
28:            add $X_b$ to $FP(X_a)$
29:            add $X_a$ to $FP(X_b)$
30:        end if
31:    end if
32:    if $\sum_{i=0}^{n} S_{new}[i] > \sum_{i=0}^{n} S_{best}[i]$ then
33:        $\mathcal{B}_{best} \leftarrow \mathcal{B}_{new}$
34:        $S_{best} \leftarrow S_{new}$
35:    end if
36: end for
37: return $\mathcal{B}_{best}$,$S_{best}$
```

Algorithm 2.6 $2iCHC(\mathcal{B}, D)$

```
Input: BN $\mathcal{B}$ over $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, Data $D$
Output: BN $\mathcal{B}'$
1: $\mathcal{B}_0 \leftarrow CHC(\mathcal{B}, D)$
2: $\mathcal{B}' \leftarrow CHC(\mathcal{B}_0, D)$
3: return $\mathcal{B}'$
```

The third approach combines the conditional independence tests and the score+search process to obtain the structure of the BN. These methods are called hybrid methods. The max-min hill-climbing (MMHC) (Tsamardinos et al., 2006) is a very popular hybrid method that uses HC for the local search. It first uses statistical hypothesis tests to find the dependencies between the variables and builds the skeleton of the network (edges without orientation) using the max-min parents and children algorithm (MMPC). Then, it orientates the arcs of the network using the HC algorithm.

All these methods, in combination with the scoring functions mentioned before, focus on improving the accuracy of the learned network, producing sometimes overfitting. As it was said before, some metrics like MDL and AIC include a penalization for the representation complexity of the network using the number of parameters of the model. The representation complexity and the inference complexity are sometimes very different for the same model (Beygelzimer and Rish, 2004). This can produce that a model with a reduced number of parameters can be exponentially slower than another with a similar representation complexity and a similar fit. So in practice, performing exact inference in the models learned using this type of methods is usually computationally expensive, and sometimes intractable when the size of the BN is too large. The most common solution is to use approximate inference in these situations, reducing the inference accuracy of the model.

2.5 Learning thin Bayesian networks

When learning BNs from data, many unnecessary dependences are usually stored in the model, increasing the complexity of the network and therefore slowing down the exact inference process. The main motivation of the learning process of BNs is to obtain fitted representations of the data that, when performing inference, obtain results that are accurate with the probability distribution implicit in the data. As we saw before, an accurate model can have a huge inference complexity that can make exact inference intractable, specially if it contains many unnecessary arcs. Exact inference is always desirable because, although approximate methods are widely used to deal with these type of situations, they include an error that can produce a relevant deterioration of the answers provided by the learned model.

A possible solution to learn models that allow exact inference is to use an estimation of the inference complexity in the learning process, with the objective of obtaining fitted models with a tractable inference complexity. Usually, the answers given by an approximate model that uses exact inference are better than the answers given by an slightly better fitted model that performs approximate inference. For example, Lowd and Domingos (2010) compared the use of exact inference in approximate models with the use of approximate inference in the original model. The difficulty for this approach is that obtaining a good measure of the inference complexity of a model is not straightforward.

For some of the most popular exact inference methods, such as JTs and some closely related variable elimination techniques, the inference complexity is exponential in the size of the largest clique of the tree. This property is called *treewidth*, and it is a good indicator of the inference complexity for probabilistic models. In the last years there have been proposed some methods that learn JTs from data using the treewidth of the models to reduce the inference complexity and to make exact inference tractable. The low-treewidth junction trees are usually called *thin junction trees* (Bach and Jordan, 2001). There are some approaches (Chechetka and Guestrin, 2008; Elidan and Gould, 2009) that learn JTs using a bounded treewidth. Vats and Nowak (2014) divided the learning process in multiple subproblems over the separators of the JTs. In order to learn thin junction trees, Shahaf and Guestrin (2009) used the *graph cuts* algorithm to select the best separator in each iteration. Flores (2005) studied the incremental compilation of JTs, and the changes that the addition, reversal or deletion of an arc in a BN could produce on a JT.

A different approach that is not related to JTs was presented by Lowd and Domingos (2008). It uses the incremental compilation of arithmetic circuits to obtain a tractable model. The ACs are closely related to the work presented in this master thesis, so a detailed review of this model is included in Chapter 3.

Chapter 3

Network Polynomials

3.1 Introduction to network polynomials

The probability distribution implicit in any BN can be also represented as a multi-linear function over two types of variables, indicators and parameters:

- *Indicators:* The evidence indicators $I(X_i = x_i)$ are Boolean functions that receive an instance x_i of a variable X_i. They return 1 if x_i is in the set of evidences or if the value of X_i is unknown, and 0 otherwise.
- *Parameters:* The network parameters $P(X_i = x_i|Pa(X_i) = \pi_i)$ for each variable instance x_i and each configuration of its parents π_i.

This multi-linear function is known as the *network polynomial* (Darwiche, 2003), and it is defined by the next expression:

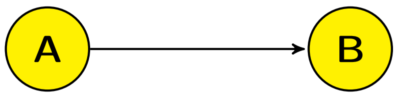
$$P(X_1 = x_1, \dots, X_n = x_n) = \sum_{i=1}^{n} \prod_{x_i,\pi_i \in \Omega_{X_i}} I(X_i = x_i) P(X_i = x_i|Pa(X_i) = \pi_i) \quad (3.1)$$

Where we use $x_i, \pi_i \in \Omega_{X_i}$ to represent each configuration of a variable and its parents x_i, π_i for the variable X_i.

This function represents the joint probability over a set of variables
$\mathcal{X} = \{X_1, \dots, X_n\}$, so the probability of any instance of the variables of the network can be computed with this formula by setting the indicators to the required values.

The instances of the variables are set by assigning the values 1 or 0 to the indicator variables.

Network polynomials allow answering any arbitrary marginal or conditional probabilistic query in linear time in the size of the polynomial, but the size of the polynomial is exponential in the number of variables of the network.



P(A)

T	F
0.6	0.4

P(B|A)

A	T	F
T	0.7	0.3
F	0.2	0.8

FIGURE 3.1: Bayesian network BN1.

For example, the network polynomial related to the BN shown in Figure 3.1 can be represented as follows:

$$\begin{aligned}P(A,B) = {} & I(A=True)I(B=True)P(A=True)P(B=True|A=True)\\ & + I(A=True)I(B=False)P(A=True)P(B=False|A=True)\\ & + I(A=False)I(B=True)P(A=False)P(B=True|A=False)\\ & + I(A=False)I(B=False)P(A=False)P(B=False|A=False)\end{aligned} \tag{3.2}$$

If we need to ask the network polynomial for the probability of the evidence $\boldsymbol{e} = (A = True, B = False)$, the evidence indicators $I(A = True)$ and $I(B = False)$ should be set to 1 and the indicators $I(A = False)$ and $I(B = True)$ should be set to 0. The resulting function is:

$P(A = True, B = False) = P(A = True) \cdot P(B = False|A = True) = 0.6 \times 0.3 = 0.18$

If we need to ask for a marginal probability such as $P(A = False)$, the only indicator set to 0 should be $I(A = True)$, and all the other indicators should be set to 1. For this query, the resulting function is:

$P(A = False) = P(A = False)P(B = True|A = False) + P(A = False)P(B = False|A = False) = 0.4 \times 0.2 + 0.4 \times 0.8 = 0.4$

Some other important features of the NPs are the properties of their partial derivatives. The function of a NP can be derived with respect to the indicators of the network or the parameters. Deriving the polynomial with respect to evidence indicators allows to compute all the partial derivatives with respect some evidence e, and therefore all the evidence instantiations that differ from the evidence indicator in only one variable. This can be useful to solve the maximum a posteriori problem by approximation using local search (Park and Darwiche, 2001; Park, 2002). For example, let us consider the partial derivative of $P(A, B)$ with respect to the indicator $I(A = False)$:

$$\begin{aligned}\frac{\partial P(A,B)}{\partial I(A = False)} &= I(B = True)P(A = False)P(B = True|A = False)\\ &+ I(B = False)P(A = False)P(B = False|A = False)\end{aligned}$$

We have seen before that the value of $P(A, B)$ at $\boldsymbol{e} = (A = True, B = False)$ is 0.18. Evaluating $\partial P(A, B)/\partial I(A = False)$ in $\boldsymbol{e}$ would return the value of $P(A = False, B = False)$, that is:

$$\begin{aligned}P(A = False, B = False) &= P(A = False)P(B = False|A = False)\\ &= 0.4 \times 0.8 = 0.32.\end{aligned}$$

The value of $\partial P(A, B)/\partial I(A = False)$ at $\boldsymbol{e} = (A = True, B = False)$ is given by:

$$\frac{\partial P(A = True, B = False)}{\partial I(A = False)} = 0 \times 0.4 \times 0.2 + 1 \times 0.4 \times 0.8 = 0.32$$

The partial derivatives with respect to the parameters of the network compute the deviation of the function produced by small changes in the parameters. This can be applied to sensitivity analysis. For example, with the purpose of setting bounds to state in which situations the changes in the parameters are relevant, Chan and Darwiche (2001) analysed the partial derivatives of probabilistic queries with respect to the parameters to study the sensitivity of these queries to changes in the parameters.

3.2 Arithmetic circuits

Although network polynomials have many desirable characteristics for representing BNs, their size is a huge practical issue given that they grow exponentially with the number of variables, making inference nearly intractable for common-size networks. Trying to overcome this difficulty, Darwiche (2003) proposed an alternative representation of the network polynomials, using *arithmetic circuits*.

ACs are a popular model for computing polynomials. The formula represented by any network polynomial can be captured in an AC, allowing a more compact representation that uses the distributive properties of the polynomials to reduce the size of the model and therefore the inference complexity. Also, it is interesting to mention that any JT can be interpreted as an AC that factorizes a network polynomial (Park and Darwiche, 2004), so ACs subsume JTs.

ACs are DAGs in which the inner nodes are addition and multiplication nodes and the leaves are numeric variables or constants. The evaluation of the circuits is straightforward and linear in the number of nodes of the graph. The circuit can be evaluated by computing the operations represented by each interior node from the values of its children, starting from the leaves.

For example, Equation (3.2) can be simplified using the distributive law to reformulate the polynomial, as it is shown in Equation (3.3). The new factorization of the polynomial reduces its complexity, leading from the 15 operations needed to evaluate Equation (3.2) to the 11 operations required to evaluate Equation (3.3). This kind of formulas can be easily represented by arithmetic circuits. The arithmetic circuit that encodes Equation (3.3) is shown in Figure 3.2.

$$\begin{aligned}P(A,B) = {} & I(A=True)P(A=True)\times(I(B=True)P(B=True|A=True)\\ & + I(B=False)P(B=False|A=True))\\ & + I(A=False)P(A=False)\times(I(B=True)P(B=True|A=False)\\ & + I(B=False)P(B=False|A=False))\end{aligned} \tag{3.3}$$

The reduction of the network polynomial size obtained by their compilation to ACs can be huge for polynomials over medium or big sets of variables, making it possible to represent some network polynomials in linear size in the number of variables,

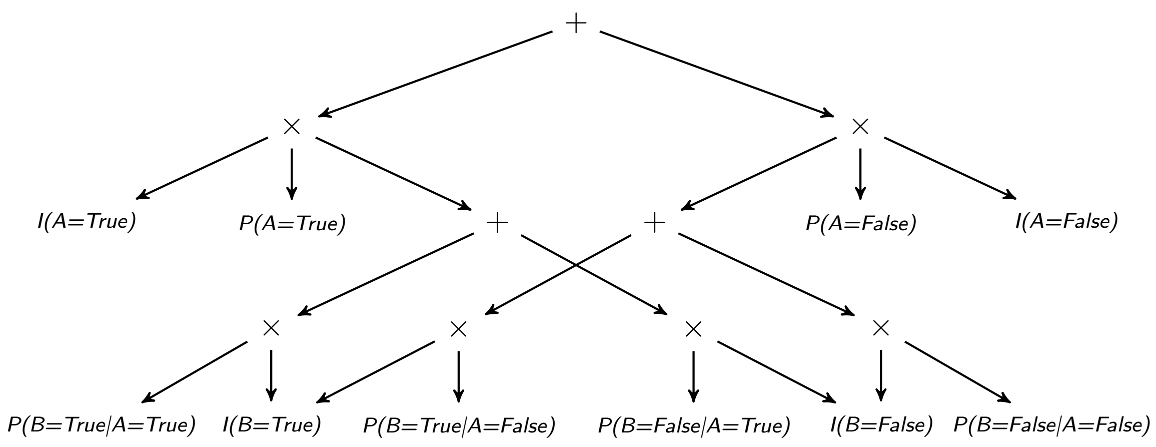


FIGURE 3.2: Arithmetic circuit AC1

allowing an inference complexity also linear in the number of variables. As it was stated by Darwiche (2003), ACs can answer any probability query that could be answered by a JT, that are the most popular framework for exact inference in BNs.

ACs also provide some useful properties related to the derivatives of the NPs that can help with the resolution of varied problems. As NPs, ACs can be evaluated or derived in linear time with the size of the circuit, making the complexity of inference linear in the size of the circuit and allowing obtaining the partial derivatives of the network polynomial with respect to the variables or the parameters of the network also in linear time. Darwiche (2003) proposed a method that, using a bottom-up evaluation of the circuit and a top-down propagation of the values, obtains all the partial derivatives of an AC given an evidence in linear time with the size of the circuit. In the previous section we showed some of the advantages of using the partial derivatives of NPs.

The MPE problem can be solved exactly in linear time using a simple reformulation of an AC. To do this, we would need to use maximizer nodes in the place of the addition nodes of the AC. The resultant model is called *maximizer circuit*. The maximizer circuit can return all the possible instances of the MPE for any evidence e required, and also the probability of the MPE for evidence $\boldsymbol{e}$ (Darwiche, 2009). The maximizer circuit that corresponds to the AC used in the previous example is shown in Figure 3.3.

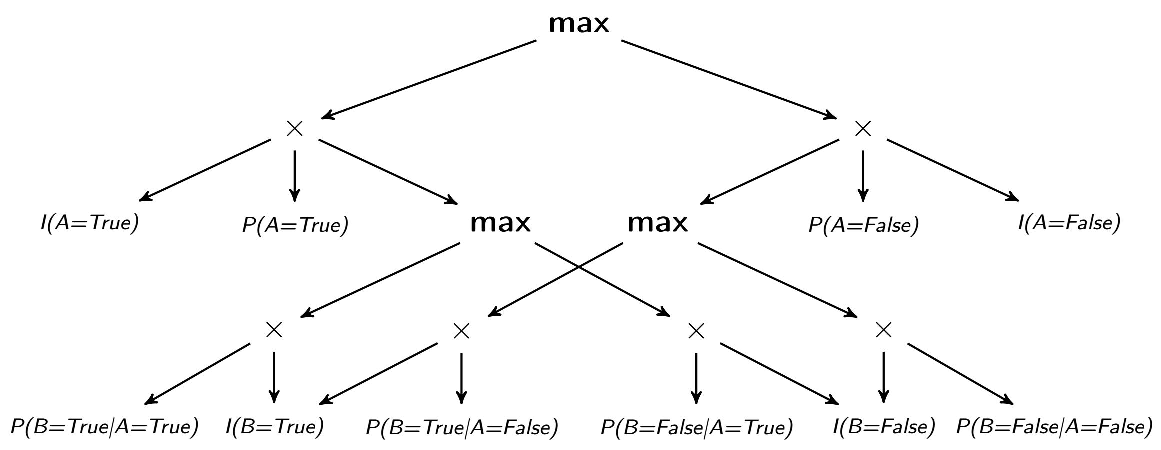


FIGURE 3.3: Maximizer circuit for AC1

3.3 Learning arithmetic circuits

The initial use of ACs in graphical models was for the compilation BNs, separating the offline and the online processes and providing an efficient framework for exact inference. The BNs can be compiled via JTs or by variable elimination to exploit the global structure of the network, but Darwiche (2003) also presented a method to compile the network exploiting its local structure. Learning ACs directly from data is a very challenging task. Darwiche uses the arithmetic circuits as a complementary representation obtained by the compilation of the original model (the BN). The method LearnAC was introduced in Lowd and Domingos (2008). It was the first approach that learns an AC directly from data. LearnAC starts from scratch, with an initial AC that represents the product of the marginal distributions $P(X_1, \ldots, X_n) = \prod_i \sum_{x_i \in \Omega_{X_i}} I(X_i = x_i) \cdot P(X_i = x_i)$, and then starts a greedy search process applying the best splits in each iteration to add the dependences between variables. The purpose of the LearnAC method is to include the inference complexity of the network in the score metric, and use it to learn tractable models. This score is described in Equation (3.4).

$$score(\mathcal{C}, D) = LL(\mathcal{C}|D) - k_e n_e(\mathcal{C}) - k_p n_p(\mathcal{C}) \tag{3.4}$$

Where:

$\mathcal{C}$: *Arithmetic circuit*

D: *Training data sample*

$LL(\mathcal{C}|D)$: *Log-likelihood of the training data*

$n_e(\mathcal{C})$,$n_p(\mathcal{C})$: *Number of arcs and parameters of the circuit respectively*

k_e,k_p: *Penalty terms per arc and parameters respectively*

The pseudocode of LearnAC is shown in Algorithm 3.1. The key of this algorithm is the splitting procedure, that updates the circuit incrementally. An split $S(\mathcal{C}, P, X_i)$ conditions the parameters in P in the variable X_i, which in a Bayesian network can be interpreted as arc additions. The experimental results presented by Lowd and Domingos (2008) show that the circuits obtained by learnAC have a tractable inference complexity, and exact inference in the obtained AC is more efficient than approximate inference using Gibbs sampling in BNs learned by other popular methods.

Algorithm 3.1 *LearnAC*$(\mathcal{X}, D)$

```
Input: set of variables $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, Data $D$
Output: Arithmetic circuit $\mathcal{C}$
 1: let $\mathcal{C}$ be an arithmetic circuit representing the product of marginals.
 2: $OKToProceed := True$
 3: while $OKToProceed = True$ do
 4:     let $\mathcal{C}_{best}$ be a copy of $\mathcal{C}$
 5:     $OKToProceed := False$
 6:     for each valid split $S(\mathcal{C}, P, X_i)$ do
 7:         $\mathcal{C}' \leftarrow SplitAC(\mathcal{C}, S(\mathcal{C}, P, X_i))$
 8:         if $score(\mathcal{C}', D) > score(\mathcal{C}_{best}, D)$ then
 9:             $\mathcal{C}_{best} \leftarrow \mathcal{C}'$
10:             $OKToProceed := True$
11:         end if
12:     end for
13:     if $OKToProceed = True$ then
14:         $\mathcal{C} \leftarrow \mathcal{C}_{best}$
15:     end if
16: end while
17: return $\mathcal{C}$
```

There are some other interesting works related to ACs learning. In Lowd and Rooshenas (2013), the LearnAC algorithm was adapted to learn Markov networks instead of BNs as ACs. The work presented by Lowd and Domingos (2010) studied the compilation of BNs with a very high treewidth, where the AC compilation proposed by Darwiche (2003) is intractable. Trying to overcome this difficulty they presented three new methods that apply approximate compilation, allowing exact inference in the approximate model. These algorithms sample from the original network to obtain data samples and then they apply the LearnAC algorithm to learn an AC.

3.3.1 Example of split

The splitting procedure is the main key of learnAC method. Let us focus on the AC shown in Figure 3.4. It represents the polynomial $P(A,B,C) = (\sum_{a\in\Omega_A} I(a)P(a)) \cdot (\sum_{c\in\Omega_C} I(c)P(c) \cdot (\sum_{b\in\Omega_B} I(b)P(b|c)))$. The addition of the arc $B \rightarrow A$ to the BN corresponds to the split $S(\mathcal{C}, \{P(A = True), P(A = False)\}, B)$. So the parameters of A are conditioned to variable B, and now there must be a summation over all the instances of B above the parameters of A in the AC. The splitting procedure of LearnAC obtains an efficient factorization of the network polynomial. The resultant AC is shown in Figure 3.5, and the polynomial represented by the updated AC is $P(A,B,C) = \sum_{b\in\Omega_B} I(b)(\sum_{a\in\Omega_A} I(a)P(a|b)) \cdot (\sum_{c\in\Omega_C} I(c)P(c)P(b|c))$.

It must be noted that Figure 3.5 only shows the part of the AC where the indicator of B is $I(B = True)$. The other part of the circuit is identical to the part shown, but using $B = False$ instead of $B = True$ in all the parameters or indicators related to variable B. In Figure 3.4 and Figure 3.5 we use T to refer to *True* and F to refer to *False*.

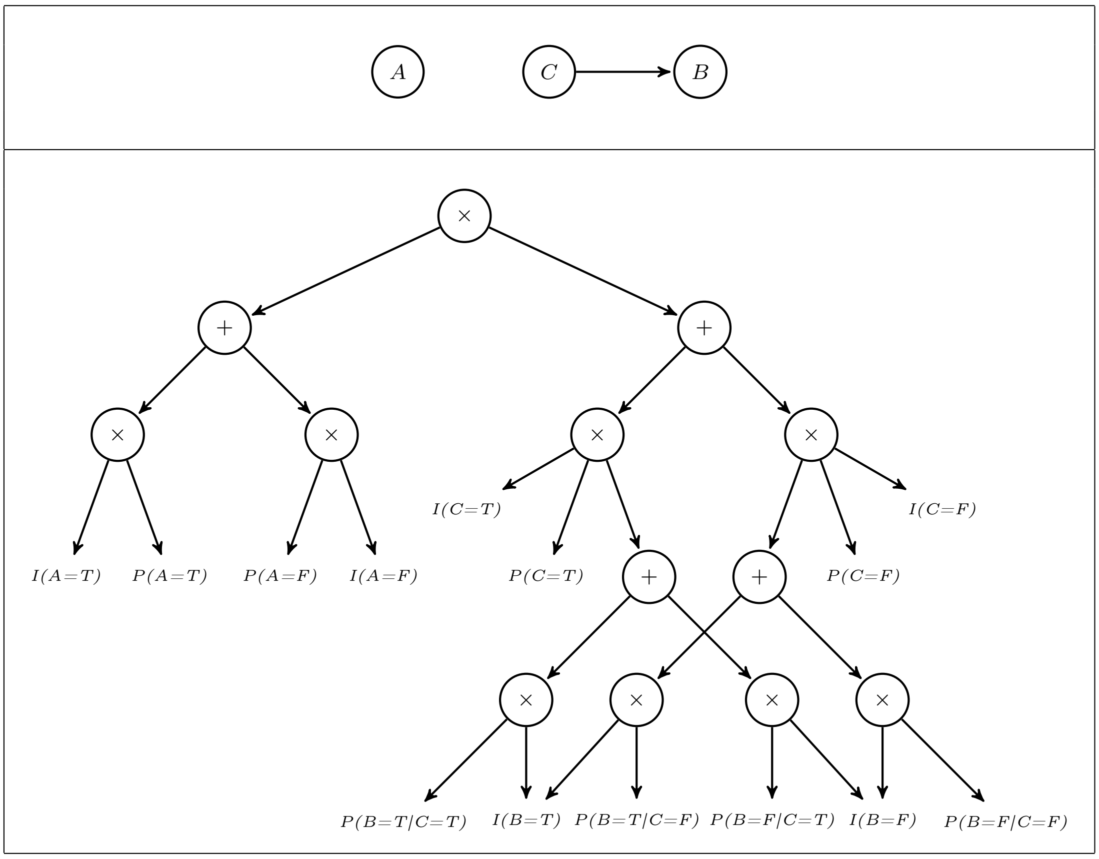


FIGURE 3.4: LearnAC example. BN (top) and AC (bottom) before split.

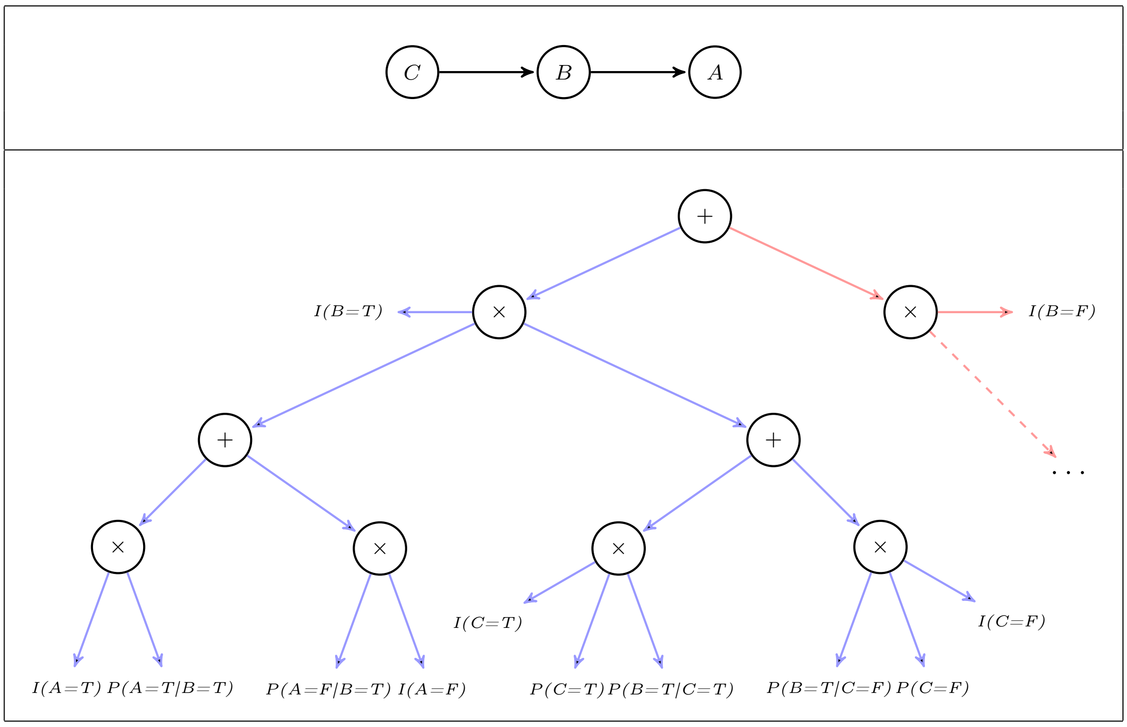


FIGURE 3.5: LearnAC example. BN (top) and AC (bottom) after split.

Chapter 4

Polynomial Trees

One of the disadvantages for the representation of BNs as ACs is that they are not a framework dedicated to probability distributions (Jaeger et al., 2006), because they can represent any kind of polynomials, including those that do not encode a network polynomial. For the task of learning a probabilistic model from data, it is essential to operate over a delimited search space, and the search space of arithmetic circuits is not bounded to the space of probability distributions. This fact makes the task of learning ACs directly from data very difficult. As we saw in Chapter 3, state-of-the-art methods for learning ACs from data like LearnAC are greedy methods that proceed by conditioning variables of the network to other variables. This is done by splitting the parameter nodes corresponding to the conditioned variable into the conditioning variable. The operations that LearnAC considers in each step are limited, including only the arc additions that produce a valid split of the current circuit. It would be very challenging to create flexible algorithms for learning ACs capable of doing and evaluating any possible movement during the search, including all the arc additions, deletions or reversals that maintain the integrity of the network. State-of-the-art methods for learning BNs usually consider all these possibilities.

Another disadvantage of the ACs is their lack of expressiveness. They are thought as a complementary model for BNs, but given that they can have a huge number of nodes and edges, it is extremely unintuitive to identify the conditional dependences between the variables of the network by only looking at the representation of the circuit, or to combine the graphical information of an AC with the graphical information provided by a BN.

The purpose of this master thesis is to find a new representation of the network polynomials that keeps the main properties of the ACs using a simple representation, easy to understand and highly compatible with the graphical representation of BNs. To achieve these goals, we propose a new graphical model complementary to BNs for representing discrete probability distributions, which we call *polynomial trees (PTs)*. As ACs, each PT encodes a compact network polynomial and is associated to a BN. A PT consists of the next elements:

1. A set of nodes $\mathcal{X}_P$, including a root node $*$ and a node $X_i \in \mathcal{X}$ for each variable of the probability distribution.

2. An indicator variable $I(X_i)$ associated to each node $X_i \in \mathcal{X}$. Each indicator can take the value of any state of the variable X_i and the value $None$ if it is not set.

3. A set of directed arcs that represent the operation dependences and ordering over all the variables of the network, forming a tree structure that connects all the nodes of the network, where the root node $*$ is the ancestor.

4. An associated Bayesian network $\mathcal{B}$ over $\mathcal{X}$. $\mathcal{B}$ is composed of a DAG representing the dependences in the network and of a set of parameters for each node $X_i \in \mathcal{X}$.

Basically, the complete graphical representation of a PT should consist of a DAG representing the dependences in the network and a tree representing the inference operation order. It is important to mention that there are usually multiple valid PTs for each BN, because there are multiple possible orders to perform exact inference on each network, so a difficulty for learning PTs would be to find those trees that require a lower number of operations per inference query.

As an example, let us focus on the structure of the BN BN2, that is shown in Figure 4.1. The network has 9 variables, and we will assume that all the variables are Boolean, and therefore they can only take the values *True* or *False*.

Figure 4.2 shows a PT that represents a valid operating order for BN2. It is possible to distinguish a DAG (in black), that shows the dependences between the variables of the network, and a tree (in red) that shows the operations chain used to perform exact inference in the PT. The root node $*$ represents the first operation to be executed in the inference procedure.

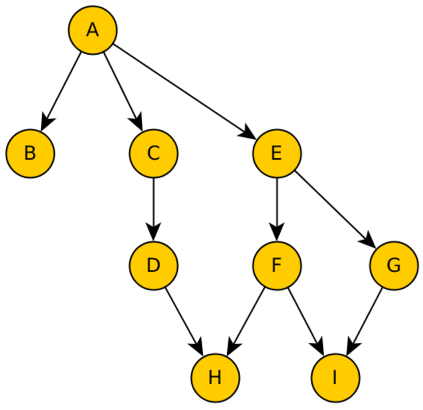


FIGURE 4.1: Structure of the Bayesian network BN2.

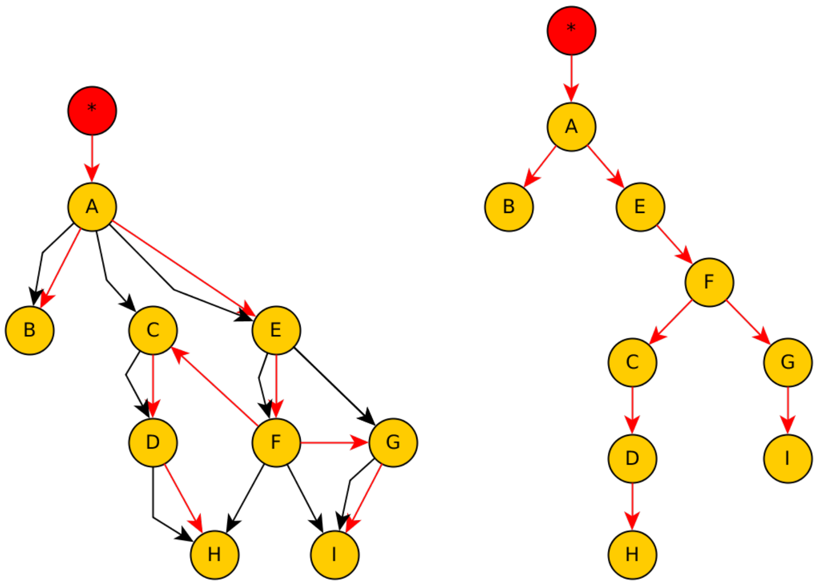


FIGURE 4.2: Combination of BN and PT for BN2 (left), and only the PT (right).

PTs are a complementary model for BNs, and each PT is associated to a BN. The soundness of a PT is the property that assures that any probabilistic query that could be answered by a BN $\mathcal{B}$ can be answered with the same response by its associated PT $\mathcal{P}$. As we will see in the next section, the inference process proposed for PTs consists of a top-down process for the propagation of the indicators and a bottom-up process for the computation of the probabilities. In order to compute the parameters of a node X_i, it is necessary that the indicators related to the parents of X_i in $\mathcal{B}$ are set, and this will only happen if all the nodes in $Pa(\mathcal{B}, X_i)$ are also predecessors of X_i in $\mathcal{P}$. The next paragraphs are the formal definitions of the soundness of any node in a PT and the soundness of the PTs respectively.

Definition 6. *Let $\mathcal{P}$ be a PT over $\mathcal{X}_P = \{*\} \cup \mathcal{X}$ with an associated Bayesian network $\mathcal{B}$ over $\mathcal{X}$. A node $X_i \in \mathcal{X}$ is sound if and only if $\forall X_j \in Pa(\mathcal{B}, X_i)$, $X_j \in Pred(\mathcal{P}, X_i)$.*

Definition 7. *Let $\mathcal{P}$ be a PT over $\mathcal{X}_P = \{*\} \cup \mathcal{X}$ with an associated Bayesian network $\mathcal{B}$ over $\mathcal{X}$. $\mathcal{P}$ is sound if and only if $\forall X_i \in \mathcal{X}$, X_i is sound.*

4.1 Inference in polynomial trees

Bayesian networks do not include an explicit exact inference procedure, and there are several different methods dedicated to this task, as it was shown in Chapter 2. Unlike BNs, ACs do represent a polynomial that can be used to perform exact inference explicitly in the circuit. The same happens with PTs, given that they encode a polynomial in a very compact representation, and this polynomial can be used to answer any joint probability query. It is straightforward to compile any PT into an AC, given that the evaluation of PTs proposed here uses a sums of products over the indicators of the tree and the parameters of its corresponding BN, and these operations can be easily captured by an AC. Nevertheless, there are multiple times where an AC cannot be compiled into a PT, given that the space of ACs is wider than the space of PTs.

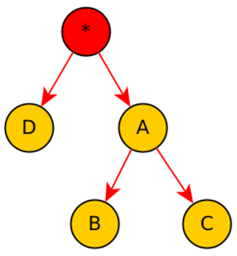
In this section we propose a method for performing exact inference in PTs. This method evaluates the polynomial encoded in a PT given a set of evidences, and it is closely related to the AC evaluation. The algorithm proceeds by first, executing a top-down process for the propagation of the indicators in the tree, and then it performs a bottom-up process where it computes the probability of the polynomial represented in the tree given the configuration of the indicators.

The concept of indicators for PTs is slightly different from the concept of indicators for ACs. While in ACs there is a binary indicator $I(X_i = x_i)$ for each instantiation x_i of variable X_i, here the indicators are simplified. There is only one indicator $I(X_i)$ associated to each variable of the tree, and $I(X_i)$ can take any value x_{ij} of X_i and the value *None*. If we set $I(X_i) = x_{ij}$ in a PT, it would be the same as setting $I(X_i = x_{ij}) = 1$ and $I(X_i = x_{ik}) = 0, \forall x_{ik}$ of X_i, $x_{ik} \neq x_{ij}$ in an AC. Setting $I(X_i) = None$ would correspond to setting $I(X_i = x_{ij}) = 1$ for any instantiation x_{ij} of X_i.

The algorithm proceeds as follows. First, it receives as an input the indicators of the query that is going to be evaluated by the polynomial. Let us use a set of indicators containing the indicator of each variable in the PT. For example, supposing that we need to evaluate the query $P(A = True, B = False)$ in the PT shown in Figure 4.3, the initial set of indicators would be:

$$\mathcal{I} = \{I(A) = True, I(B) = False, I(C) = None, I(D) = None\}$$

As it was shown before, there is an indicator for each variable, and each indicator can take the value of any instantiation of its corresponding variable or the value $None$, that indicates that the value of the indicator is not set. For each variable passed as an evidence to the tree the value of the corresponding indicator is set to the value of the evidence. Otherwise, the indicators related to any variable that is not in the evidence set takes the value $None$. In the example, variables A and B are passed as evidence in the query $P(A = True, B = False)$, so the values of their indicators are set to $I(A) = True$ and $I(B) = False$ respectively. Variables C and D do not appear in the evidence set of the query, so their values are set to $I(C) = None$ and $I(D) = None$ initially.



P(D)

T	F
0.6	0.4

P(A)

T	F
0.8	0.2

P(B|A)

A	T	F
T	0.3	0.7
F	0.5	0.5

P(C|A)

C	T	F
T	0.9	0.1
F	0.4	0.6

FIGURE 4.3: Example. A PT and the parameters of its corresponding BN.

Given an indicators set $\mathcal{I}$, then the probability of $\mathcal{I}$ in the PT can be computed using Equation (4.1) for the initial step and Equation (4.2) for the inductive steps. The first step is to evaluate the root node $*$, as it is described in Algorithm (4.1). The probability of $\mathcal{I}$ for $*$ can be obtained by evaluating each of its children given $\mathcal{I}$, and computing the final value as the product of all the values returned by these evaluations.

$$query(\mathcal{B}, \mathcal{P}, \mathcal{I}) = \prod_{X_i \in Ch(\mathcal{P}, *)} queryNode(\mathcal{B}, \mathcal{P}, X_i, \mathcal{I}) \tag{4.1}$$

In the example, the root node $*$ has two children, nodes A and D, so the result of evaluating $*$ is $query(\mathcal{B}, \mathcal{P}, \mathcal{I}) = queryNode(\mathcal{B}, \mathcal{P}, D, \mathcal{I}) \cdot queryNode(\mathcal{B}, \mathcal{P}, D, \mathcal{I})$

Algorithm 4.1 $query(\mathcal{B}, \mathcal{P}, \mathcal{I})$

```
Input: BN $\mathcal{B}$, PT $\mathcal{P}$, set of indicators $\mathcal{I}$
Output: Probability of $\mathcal{I}$ in $\mathcal{P}$
1: let $*$ be the root node of $\mathcal{P}$
2: $P := 1$
3: for $X_i \in Ch(\mathcal{P}, *)$ do
4:     $value := queryNode(\mathcal{B}, \mathcal{P}, X_i, \mathcal{I})$
5:     $P := P \cdot value$
6: end for
7: return $P$
```

The rest of the nodes can be evaluated using the process described by Algorithm 4.2. The operations needed to obtain the returned value are represented in Equation (4.2). Basically, if the value of the indicator $I(X_i)$ corresponding to variable X_i is not set ($X_i = None$), the result is equal to the sum of evaluating X_i, setting its indicator $I(X_i)$ to each of its possible instantiations. Otherwise, the returned value is the parameter that corresponds to the current configuration of variable X_i and its parents in the indicators set multiplied by the result of evaluating each children of X_i. That is,

$$\begin{aligned} queryNode(\mathcal{B}, \mathcal{P}, X_i, \mathcal{I}) = \sum_{x_i \in \Omega_{C_i}} & P(X_i = x_i | Pa(\mathcal{B}, X_i) = \pi_{X_i}) \\ \times & \prod_{X_j \in Ch(\mathcal{P}, X_i)} queryNode(\mathcal{B}, \mathcal{P}, X_j, I_{X_i}) \end{aligned} \tag{4.2}$$

Where:

- C_i: If $I(X_i) \neq None$, then $C_i = \{I(X_i)\}$, otherwise C_i is a set containing all the possible instances of X_i.
- π_{X_i}: Set with the value of each parent of X_i in $\mathcal{I}$.
- I_{X_i}: Set of indicators obtained after setting the value of $I(X_i)$ to x_i in $\mathcal{I}$.

Algorithm 4.2 *queryNode*$(\mathcal{B}, \mathcal{P}, X_i, \mathcal{I})$

Input: BN $\mathcal{B}$ over $\mathcal{X} = \{X_1, \dots, X_n\}$, PT $\mathcal{P}$, node X_i, set of indicators $\mathcal{I}$

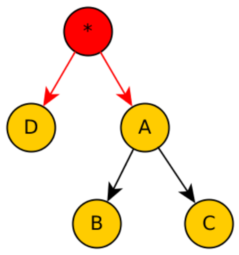
Output: Partial probability of $\mathcal{I}$ in P from the node X_i

```
1:  if $I(X_i) = None$ then
2:      let $C_i$ be a set containing all the possible instantiations of $X_i$
3:  else
4:      $C_i \leftarrow \{I(X_i)\}$
5:  end if
6:  $P_r := 0$
7:  for $c \in C_i$ do
8:      let $\pi_{X_i}$ be the configuration in $\mathcal{I}$ of $Pa(\mathcal{B}, X_i)$
9:      $P_{aux} := P(X_i = c | Pa(\mathcal{B}, X_i) = \pi_{X_i})$
10:     let $\mathcal{I}_c$ be a copy of $\mathcal{I}$
11:     $I_c(X_i) \leftarrow c$
12:     for $X_j \in Ch(\mathcal{P}, X_i)$ do
13:         $value := queryNode(\mathcal{B}, \mathcal{P}, X_j, \mathcal{I}_c)$
14:         $P_{aux} := P_{aux} \cdot value$
15:     end for
16:     $P_r := P_r + P_{aux}$
17: end for
18: return $P$
```

4.1.1 Example. Inference in polynomial trees

Let us imagine that we needed to evaluate node A in the PT used in this example, where we need to get the probability $P(A = True|B = False)$. The algorithm would proceed as follows. Knowing that the set of indicators is $\mathcal{I} = \{I(A) = True, I(B) = False, I(C) = None, I(D) = None\}$ and therefore $I(A) \neq None$, variable C_i would be set to $\{I(A)\}$, so $C_i = \{True\}$. The value returned by $queryNode(\mathcal{B}, \mathcal{P}, A, \mathcal{I})$ would be $P(A = True) \cdot (queryNode(\mathcal{B}, \mathcal{P}, B, \mathcal{I}_A) \cdot queryNode(\mathcal{B}, \mathcal{P}, C, \mathcal{I}_A))$. The complete process that should be executed in order to evaluate the PT of Figure 4.3 given the set of indicators $\mathcal{I} = \{I(A) = True, I(B) = False, I(C) = None, I(D) = None\}$ is shown below.

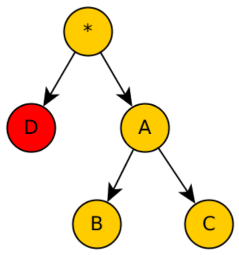
Step 1



$\mathcal{I} = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{None}, I(D) = \mathit{None}\}$

$\mathit{queryNode}(\mathcal{B}, \mathcal{P}, *, \mathcal{I}) = \mathit{queryNode}(\mathcal{B}, \mathcal{P}, D, \mathcal{I}) \cdot \mathit{queryNode}(\mathcal{B}, \mathcal{P}, A, \mathcal{I})$

Step 2



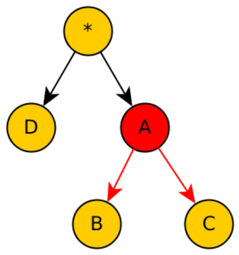
$\mathcal{I} = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{None}, I(D) = \mathit{None}\}$

$\mathcal{I}_{D_1} = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{None}, I(D) = \mathit{True}\}$

$\mathcal{I}_{D_2} = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{None}, I(D) = \mathit{False}\}$

$\mathit{queryNode}(\mathcal{B}, \mathcal{P}, D, \mathcal{I}) =$
$\mathit{queryNode}(\mathcal{B}, \mathcal{P}, D, \mathcal{I}_{D_1}) + \mathit{queryNode}(\mathcal{B}, \mathcal{P}, D, \mathcal{I}_{D_2}) =$
$P(D = \mathit{True}) + P(D = \mathit{False}) = 0.6 + 0.4 = 1$

Step 3

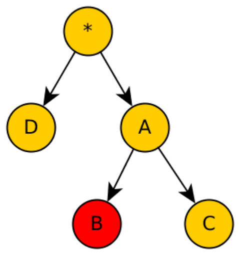


$\mathcal{I} = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{None}, I(D) = \mathit{None}\}$

$\mathcal{I}_A = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{None}, I(D) = \mathit{None}\}$

$\mathit{queryNode}(\mathcal{B}, \mathcal{P}, A, \mathcal{I}) =$
$P(A = \mathit{True}) \cdot (\mathit{queryNode}(\mathcal{B}, \mathcal{P}, B, \mathcal{I}_A) \cdot \mathit{queryNode}(\mathcal{B}, \mathcal{P}, C, \mathcal{I}_A)$

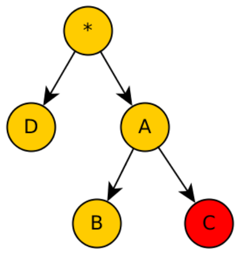
Step 4



$\mathcal{I}_A = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{None}, I(D) = \mathit{None}\}$

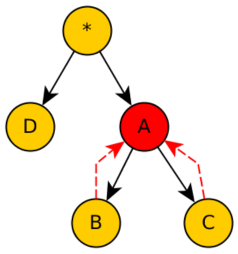
$\mathit{queryNode}(\mathcal{B}, \mathcal{P}, B, \mathcal{I}_A) = P(B = \mathit{False} | A = \mathit{True}) = 0.7$

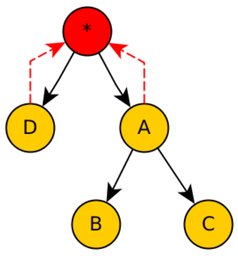
Step 5



$\mathcal{I}_A = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{None}, I(D) = \mathit{None}\}$
$\mathcal{I}_{C_1} = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{True}, I(D) = \mathit{None}\}$
$\mathcal{I}_{C_2} = \{I(A) = \mathit{True}, I(B) = \mathit{False}, I(C) = \mathit{False}, I(D) = \mathit{None}\}$

$\mathit{queryNode}(\mathcal{B}, \mathcal{P}, D, \mathcal{I}_A) =$
$\mathit{queryNode}(\mathcal{B}, \mathcal{P}, C, \mathcal{I}_{C_1}) + \mathit{queryNode}(\mathcal{B}, \mathcal{P}, C, \mathcal{I}_{C_2}) =$
$P(C = \mathit{True} | A = \mathit{True}) + P(C = \mathit{False} | A = \mathit{True}) = 0.9 + 0.1 = 1$

Step 6	
	$\mathcal{I}_A = \{I(A) = True, I(B) = False, I(C) = None, I(D) = None\}$ $queryNode(\mathcal{B}, \mathcal{P}, A, \mathcal{I}) =$ $P(A = True) \cdot queryNode(\mathcal{B}, \mathcal{P}, B, \mathcal{I}_A) \cdot queryNode(\mathcal{B}, \mathcal{P}, C, \mathcal{I}_A) =$ $0.8 \times 0.7 \times 1 = 0.56$

Step 7	
	$\mathcal{I} = \{I(A) = True, I(B) = False, I(C) = None, I(D) = None\}$ $queryNode(\mathcal{B}, \mathcal{P}, *, \mathcal{I}) =$ $queryNode(\mathcal{B}, \mathcal{P}, D, \mathcal{I}) \cdot queryNode(\mathcal{B}, \mathcal{P}, A, \mathcal{I}) =$ $1 \times 0.56 = 0.56$

4.2 Evaluating the complexity of polynomial trees

The PT representation is also an intuitive indicator of the inference complexity for the polynomial that it encodes. One of the main advantages of PTs is that it is simple to obtain exactly the maximum inference complexity of the model. In most state-of-the-art methods for learning thin probabilistic models the treewidth is used as an estimation of the inference complexity. Obtaining the treewidth of a graph is an NP-complete problem, so in most methods it is estimated (Beygelzimer and Rish, 2004). Another strategy that was reviewed in Chapter 2 is the incremental compilation of junction trees, that allows obtaining the exact value of the treewidth in linear time, but these methods are usually very computationally requiring and sometimes they use a very constrained search space that could affect the faithfulness of the model.

Using PTs, it is possible to obtain exactly the maximum number of operations that must be computed to solve any joint query asked to the network in linear time in the number of variables of the network. This makes the evaluation of PTs tractable for learning networks of any size. The method used in this report for the complexity evaluation of PTs, defined by Algorithm 4.3, works by evaluating the tree recursively. It starts in the root node, and in each step it gets the complexity of the sub-tree that has the current node as its root. This complexity is equal to the number of possible instances of the current node multiplied by the sum of evaluating the complexity of each of its children. The value of the complexity for each node is represented in Equation (4.3).

$$evalNode(\mathcal{P}, X_i) = |X_i| \cdot \left(1 + \sum_{X_j \in Ch(\mathcal{P}, X_i)} 1 + evalNode(\mathcal{P}, X_j)\right) \tag{4.3}$$

Where $|X_i|$: denotes the number of different possible instances of X_i.

Algorithm 4.3 $evalNode(\mathcal{P}, X_i)$

```
Input: Polynomial tree $\mathcal{P}$, current node $X_i$
Output: Evaluation of the complexity of $\mathcal{P}$
1: $eval = 0$
2: for $X_j \in Ch(\mathcal{P}, X_i)$ do
3:     $eval := eval + 1 + evalNode(\mathcal{P}, X_j)$
4: end for
5: let $len$ be the number of possible instantiations of $X_i$
6: return $len \cdot (1 + eval)$
```

4.2.1 Example. Evaluating the complexity of a polynomial tree

The next example shows the process performed by Algorithm 4.3 for evaluating the complexity of a PT. In particular, it computes the maximum number of operations that could be computed by P, where P is the PT shown in Figure 4.3. The operations computed in each step of the example, that are represented in Figure 4.4 are:

Step 1. $evalNode(\mathcal{P}, *) = |*| + (1 + (evalNode(\mathcal{P}, D) + 1) + (evalNode(\mathcal{P}, A) + 1))$
Step 2. $evalNode(\mathcal{P}, D) = |D| = 2$
Step 3. $evalNode(\mathcal{P}, A) = |A| \cdot (1 + (evalNode(\mathcal{P}, B) + 1) + (evalNode(\mathcal{P}, C) + 1))$
Step 4. $evalNode(\mathcal{P}, B) = |\mathcal{B}| = 2$
Step 5. $evalNode(\mathcal{P}, C) = |C| = 2$
Step 6. $evalNode(\mathcal{P}, A) = |A| \cdot (1 + (evalNode(\mathcal{P}, B) + 1) + (evalNode(\mathcal{P}, C) + 1)) = 2 \cdot (1 + (2 + 1) + (2 + 1)) = 14$
Step 7. $evalNode(\mathcal{P}, *) = |*| + (1 + (evalNode(\mathcal{P}, D) + 1) + (evalNode(\mathcal{P}, A) + 1)) = 1 \cdot (1 + (1 + 2) + (1 + 14)) = 19$

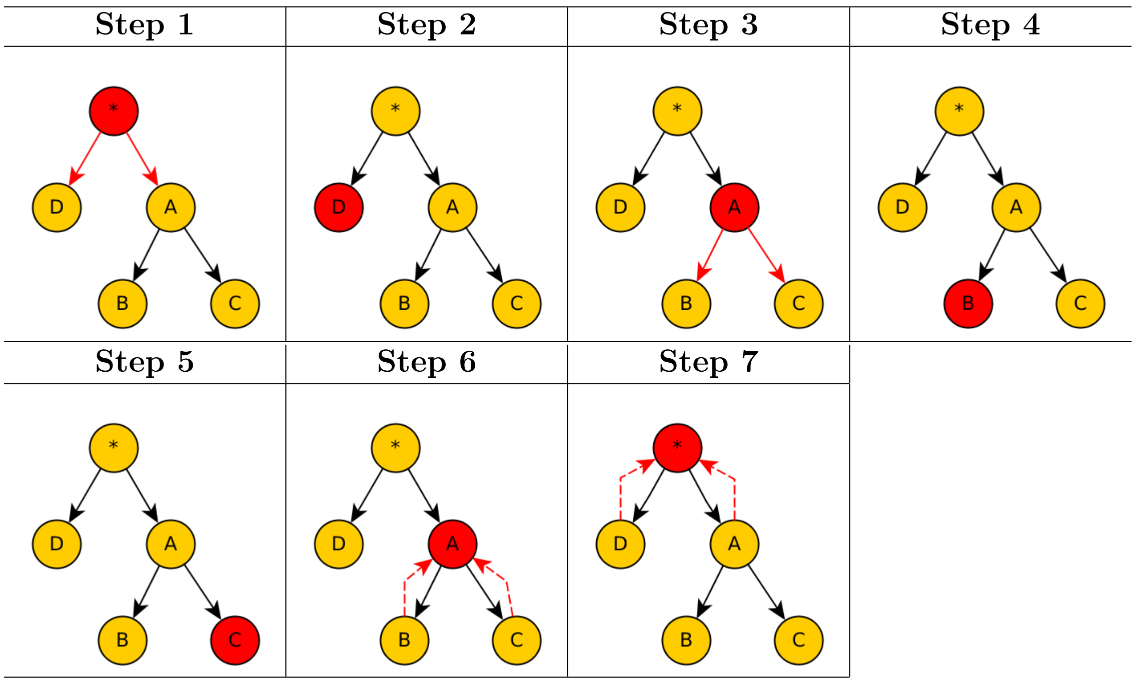


FIGURE 4.4: Example of the inference complexity calculation for P

4.3 Incremental compilation of polynomial trees

Some of the main advantages of using PTs for learning BNs is that the complexity of the model can be evaluated in linear time and that it provides a complementary model where it is straightforward to perform exact inference. This produces that the models learned by the incremental compilation of PTs will usually have a good performance for exact inference. There are multiple strategies to learn PTs, a possibility is to first learn the BN and then compile the complete network into a PT, as it was proposed in Darwiche (2003) for learning ACs, but it does not fulfil our goal, that is learning thin models from data. Another possibility is to compile a PT for each network candidate during the search, this would solve the previous problem, but it would demand too much computation, making the search extremely inefficient or even intractable for trees over large sets of variables. This paper proposes a method for learning PTs by the incremental compilation of the changes applied to BNs, allowing a low complexity evaluation framework and a feasible performance. Next, we introduce the concept of optimality in PTs.

Definition 8. *A polynomial tree $\mathcal{P}$ over $\{*\} \cup \mathcal{X}$ is optimal if it is sound and it encodes the polynomial with the minimum number of operations needed to compute the most expensive query that it can answer. The most expensive query in $\mathcal{P}$ is the one where the indicator of each variable in $\mathcal{X}$ is set to None.*

Although in the next sections we propose a method for learning PTs that uses an scoring function in particular, the main idea of this master thesis is to provide a framework that can be applied to any score+search method in combination with different metrics used for learning BNs. The PT obtained after the learning process must be sound, and it is desirable that it is close to the optimal PT. For this purpose we need to consider three types of operations:

- Addition of new arcs.
- Elimination of arcs.
- Reversion of arcs.

This section includes the methods required to compile incrementally these operations in PTs, allowing the incremental learning of the trees in parallel with the process for learning the structure of the BN. All the methods proposed assure that any learned PT is sound, but the obtained models may be far from an optimal PT. To overcome this difficulty we also include an optimization method with the purpose of obtaining desirable PTs with a tractable computational complexity.

4.3.1 Arc addition

The addition of an arc in a BN is straightforward, but in a PT the compilation process is not so simple, and it depends on the current configuration of the PT. There are three main scenarios that we need to consider, depending on the configuration of the tree in each step. Lets consider any addition operation $(X_{out} \rightarrow X_{in})$ that should be compiled in a PT $\mathcal{P}$ and included in its corresponding BN $\mathcal{B}$. We will refer to X_{out} as the output node and X_{in} as the input node. The scenarios that could be involved in any arc addition are the following:

- **Scenario 1.** $X_{out} \in Pred(\mathcal{P}, X_{in})$: This is the simplest scenario. The addition of an arc from X_{out} to X_{in} does not suppose any changes in $\mathcal{P}$, because with the change the tree keeps being sound and its optimality is not affected.
- **Scenario 2.** $X_{in} \in Pred(\mathcal{P}, X_{out})$: This scenario is the most complex one. The operation supposes a restructuring of the nodes between X_{in} and X_{out} in order to get a sound $\mathcal{P}$, and an optimization process should be applied to obtain a satisfactory tree.

- **Scenario 3.** $X_{out} \notin Pred(\mathcal{P}, X_{in})$ **and** $X_{in} \notin Pred(\mathcal{P}, X_{out})$: In this scenario, the node X_{out} and its predecessors must be also predecessors of X_{in} in $\mathcal{P}$. Then, an optimization process should be applied.

Imagining that we are learning the BN and the PT shown in Figure 4.5, Let us focus on some arc additions that cover the three different scenarios. In each example we show the resulting BN after applying the addition of the arc, and the resulting PT after compiling the change in the tree. The obtained PTs are sound but they may be far from optimal, given that we do not include the optimization process in these examples. Figure 4.6 corresponds to the addition of the arc $A \rightarrow E$. In this case A is currently a predecessor of E in the PT (scenario 1), so no changes in the tree are required. Figure 4.7 corresponds to the addition of $E \rightarrow A$. This change implies a reconfiguration of the network given that A is currently a predecessor of E in the PT (scenario 2), and now E must be a predecessor of A and the tree must keep its soundness. The last example corresponds to the addition of the arc $C \rightarrow F$. Given that the only predecessor in common between C and F is the root node $*$ (scenario 3), then C and all its predecessors must be placed as predecessors of F to maintain the soundness of the tree.

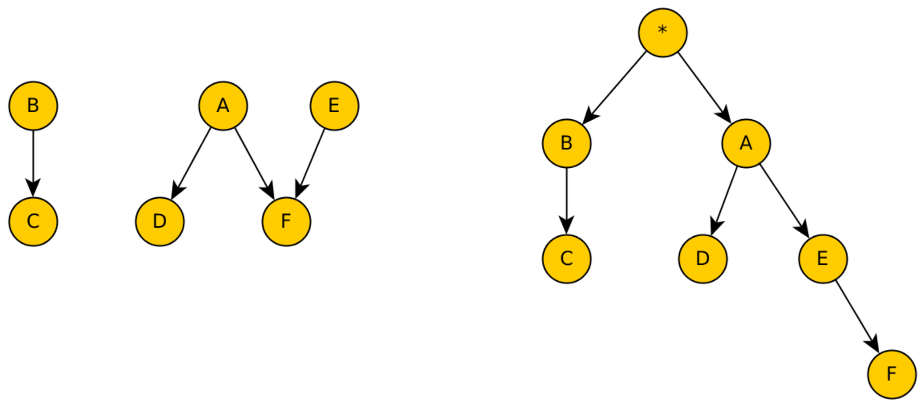


FIGURE 4.5: Examples of BN (left) and PT (right) respectively.

The method proposed in this paper for the incremental compilation of arc additions in PTs is described by Algorithm 4.4, and it proceeds as follows. The first step of the algorithm is to update the connections of the BN $\mathcal{B}$, adding and arc from X_{out} to X_{in} (line 5). After that, the changes will only affect to the PT $\mathcal{P}$. The next step is to identify the scenario that we are facing at the time, and for that it is necessary to get the first common predecessor X_{pred} of X_{out} and X_{in} in the tree. If X_{pred} is the node X_{out}, then X_{out} is currently a predecessor of X_{in}, and we are in scenario 1. If X_{pred} is the node X_{in}, then X_{in} is a predecessor of X_{out}, and we are

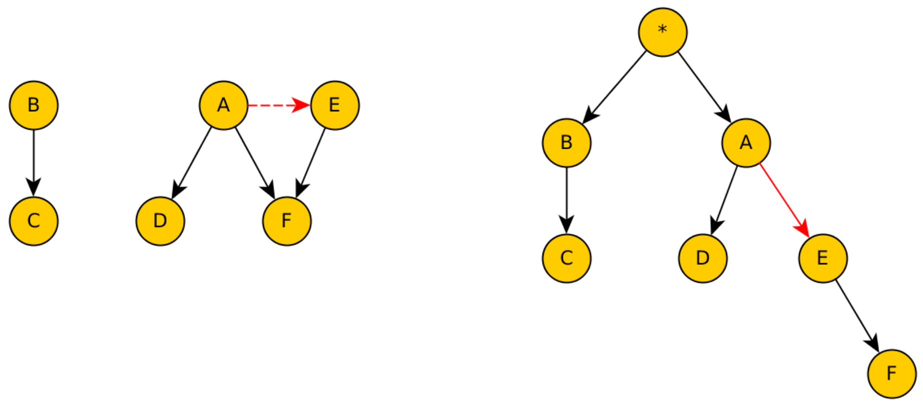


FIGURE 4.6: Example of arc addition for scenario 1. BN (left) and PT (right).

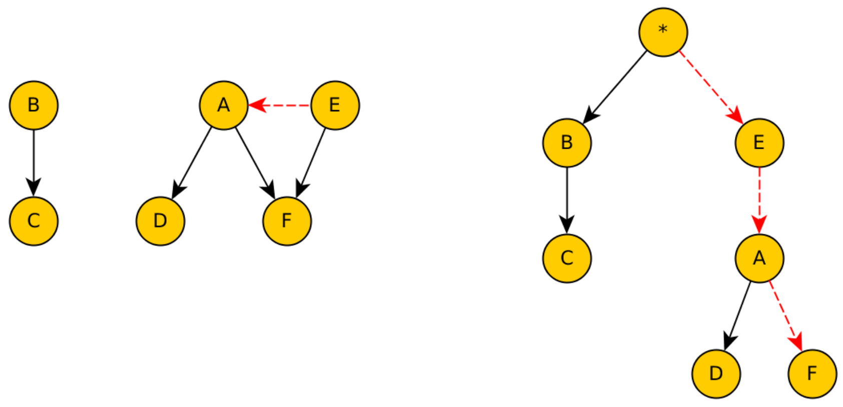


FIGURE 4.7: Example of arc addition for scenario 2. BN (left) and PT (right).

in scenario 2. Otherwise, neither X_{out} or X_{in} are predecessors of each other and we are facing scenario 3.

In scenario 1 , no actions are required to maintain the soundness and optimality of P.

In scenario 2 (line 7), it is necessary to swap the positions of the nodes X_{out} and X_{in} in the tree. The algorithm gets the list of nodes that are between X_{out} and X_{in} (line 8), that we will call $\boldsymbol{C_N}$ for now on. Given that the nodes in $\boldsymbol{C_N}$ may be predecessors of X_{in} in $\mathcal{B}$, they will need to be reorganized in most cases. For this reason, the algorithm uses a list to store the nodes belonging to $\boldsymbol{C_N}$ that are predecessors of X_{in} in $\mathcal{B}$ (line 10). We will refer to this list as $\boldsymbol{dL}$. The algorithm will also store the last nodes belonging to $\boldsymbol{C_N}$ that have been visited during the procedure. The last node visited that belongs to $Desc(\mathcal{B}, X_{in})$ will be assigned to C_{in}, and the last node visited that is not in $Desc(\mathcal{B}, X_{in})$ will be assigned to C_{out}.

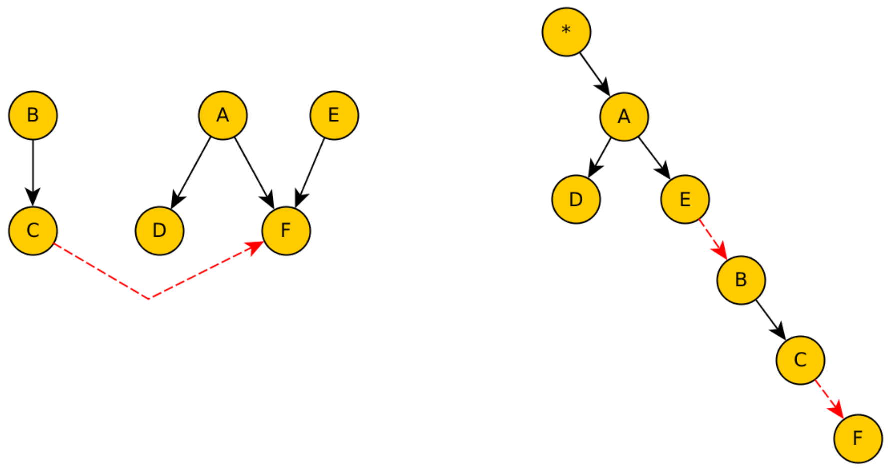


FIGURE 4.8: Example of arc addition for scenario 3. BN (left) and PT (right).

The algorithm will iterate over the nodes belonging to $\boldsymbol{C_N}$ (line 15), starting from the shallowest one and finishing in the deepest one. Before the first iteration, C_{in} is initialized to X_{in} (line 11) and the parent of X_{in} is assigned to C_{out} (line 12). In each iteration, the dependences in the BN of the visited node, which we will refer to as X_j, will be checked (line 18). If X_j has any of the nodes belonging to $\boldsymbol{dL}$ as one of its parents (line 23), which would mean that it is a descendant of X_{in} in $\mathcal{B}$, then the parent of X_j would be set to C_{in}, and X_j would be assigned to C_{in} and added to $\boldsymbol{dL}$. Otherwise (line 27), the parent of X_j would be set to C_{out}, and X_j would be assigned to C_{out}.

It is also necessary to consider the dependences of each branch hanging from the node (line 28). If X_j is not a predecessor of X_{in} in the BN and any node of the branch that hangs from X_j does, the soundness of $\mathcal{P}$ would be compromised, because the nodes of the branch would not have X_{in} as one of their predecessors in the network. To overcome this difficulty, the branch will pass to hang from the node stored in C_{in} instead. Applying these steps to the tree would always produce sound trees, but in some cases they can be far from optimal, so an optimization step is required.

In scenario 3 (line 40), node X_{out} is a predecessor of X_{in} in $\mathcal{B}$ but it is not in $\mathcal{P}$. To maintain the soundness of $\mathcal{P}$, node X_{out} and therefore its predecessors in $\mathcal{P}$ must be reorganized. It is enough to set X_{out} and its predecessors in the PT as direct predecessors of X_{in} by first, setting X_{out} as the parent of X_{in}, and setting the previous parent of X_{in} in $\mathcal{P}$ as the parent of the shallowest predecessor of X_{out} in $\mathcal{P}$ that is also a descendant of C_{out} in $\mathcal{P}$. As it happens with scenario 2, this procedure

Algorithm 4.4 *addArc*$(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$ - Part I

```
Input: BN $\mathcal{B}$, PT $\mathcal{P}$, output node $X_{out}$, input node $X_{in}$
Output: Updated BN $\mathcal{B}'$ and PT $\mathcal{P}'$
 1: let $\mathcal{P}'$ be a copy of $\mathcal{P}$
 2: let $\mathcal{B}'$ be a copy of $\mathcal{B}$
 3: let $X_{pred}$ be the first common predecessor of $X_{out}$ and $X_{in}$ in $\mathcal{P}$
 4: let $C_{pred}$ be the child of $X_{pred}$ that is a predecessor of $X_{in}$ in $\mathcal{P}$
 5: append $X_{out}$ to $Pa(\mathcal{B}', X_{in})$
 6:                                              ▷ Scenario 1 requires no actions
 7: if $X_{pred} = X_{in}$ then                                        ▷ Scenario 2
 8:     let $\boldsymbol{C_N}$ be the nodes that are descendants of $X_{in}$ and predecessors of $X_{out}$ in $P$,
        ordered from the shallowest to the deepest
 9:     $\boldsymbol{C_N} \leftarrow concatenate((X_{in}), \boldsymbol{C_N}, (X_{out}))$
10:     $\boldsymbol{dL} \leftarrow list(X_{in})$  ▷ list for $X_{in}$ and its descendants in $\mathcal{B}'$ that belong to $\boldsymbol{C_N}$
11:     $C_{in} \leftarrow X_{in}$                              ▷ Current tail of the list $\boldsymbol{dL}$
12:     $C_{out} \leftarrow Pa(\mathcal{P}', X_{in})$ ▷ Last visited node that is not a descendant of $X_{in}$ in $\mathcal{B}'$
13:     remove $C_{out}$ from $Pa(\mathcal{P}', X_{in})$
14:     let $l$ be the length of $\boldsymbol{C_N}$
15:     for $j := 1, \ldots, l-1$ do
16:         $X_j \leftarrow \boldsymbol{C_N}[j]$
17:         $flag \leftarrow False$
18:         for $X_k \in Pa(\mathcal{B}', X_j)$ do              ▷ Check if $X_j$ is a descendant of $X_{in}$
19:             if $X_k \in \boldsymbol{dL}$ then
20:                 $flag \leftarrow True$
21:             end if
22:         end for
23:         if $flag$ then                                    ▷ Update $C_{in}$ and $\boldsymbol{dL}$
24:             append $X_j$ to $\boldsymbol{dL}$
25:             $Pa(\mathcal{P}', X_j) \leftarrow C_{in}$
26:             $C_{in} \leftarrow X_j$
27:         else              ▷ Update $C_{out}$ and set the parent of each children of $X_j$
28:             for $X_k \in Ch(\mathcal{P}', X_j)$ do
29:                 if $X_k \neq \boldsymbol{C_N}[j+1]$ then
30:                     $Pa(\mathcal{P}', X_k) \leftarrow C_{in}$
31:                 end if
32:             end for
33:             $Pa(\mathcal{P}', X_j) \leftarrow C_{out}$
34:             $C_{out} \leftarrow X_j$
35:         end if
36:     end for
37:     set $C_{in}$ as the parent in $P'$ of all the nodes in $Ch(P, X_{out})$.
38:     $Pa(\mathcal{P}', X_{out}) \leftarrow C_{out}$    ▷ Set the parents of the input and the output nodes
```

Algorithm 4.4 *addArc*$(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$ - Part II

```
39:       $Pa(\mathcal{P}', X_{in}) \leftarrow X_{out}$
40:   else if $X_{pred} \neq X_{in}$ then                              ▷ Scenario 3
41:       $Pa(\mathcal{P}', C_{pred}) \leftarrow Pa(\mathcal{P}', X_{in})$
42:       $Pa(\mathcal{P}', X_{in}) \leftarrow X_{out}$
43:   end if
44:   return $\mathcal{B}', \mathcal{P}'$
```

ensures that the resultant tree is always sound but it may be sometimes far from optimal.

It is important to remark that the compilation of an edge addition will never affect to the soundness of $\mathcal{P}$. The proof is provided in Lemma 1 in Appendix A.

4.3.2 Arc deletion

The second operation that we need to consider is the arc deletion. On the one hand, it is straightforward to obtain a sound PT incrementally after an arc deletion in the BN, because it is enough to maintain the current configuration of the tree. On the other hand, a huge reduction in the complexity of the PT may be achieved optimizing the PT without the deleted arc.

The algorithm for arc deletion is extremely simple (see Algorithm 4.5). The only task required is to update the BN removing the old connection $(X_{out} \rightarrow X_{in})$. An optimization process should be applied to reduce the complexity of the tree. Figure 4.9 is an example of compiling the deletion of arc $A \rightarrow F$ in the BN and PT shown in Figure 4.5. There are no changes applied to the PT and it keeps being sound regarding to the new BN, but a model with a lower inference complexity could be obtained optimizing the tree.

As it is proved in the Lemma 2 in Apendix A, the method *deleteArc* never alters the soundness of a PT.

Algorithm 4.5 *deleteArc*$(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$

```
Input: BN $\mathcal{B}$, PT $\mathcal{P}$, output node $X_{out}$, input node $X_{in}$
Output: Updated BN $\mathcal{B}'$ and PT $\mathcal{P}'$
 1: let $\mathcal{B}'$ be a copy of $\mathcal{B}$
 2: remove $X_{out}$ from $Pa(\mathcal{B}', X_{in})$
 3: return $\mathcal{B}', P$
```

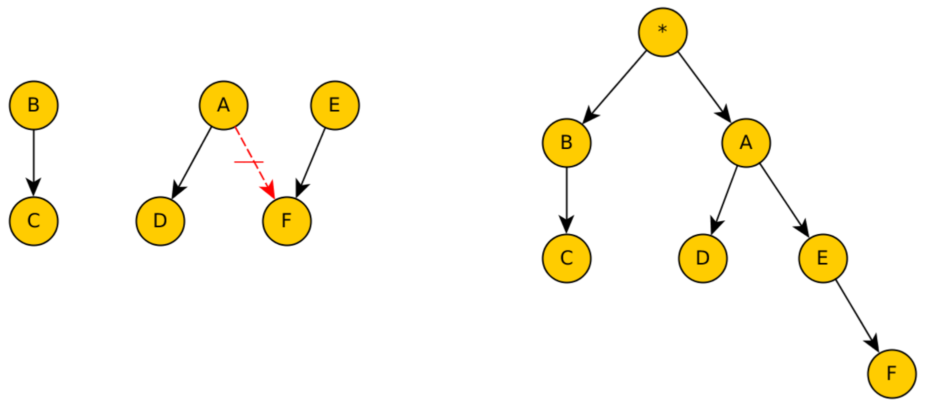


FIGURE 4.9: Example of an arc deletion. Initial BN (left) and PT (right).

4.3.3 Arc reversal

Arc reversal can be easily applied combining the deletion and addition of the reverse arc. Imagine that we need to reverse arc $X_{out} \rightarrow X_{in}$. The algorithm proposed in this report first removes arc $X_{out} \rightarrow X_{in}$. After this step we will be facing scenario 2 for the addition of arc $X_{in} \rightarrow X_{out}$, so it is enough to apply the algorithm described before for the addition of a new arc in these scenarios. The complete method for the addition of new PTs is described in Algorithm 4.6. Figure 4.10 shows the effects of compiling the reversal of arc $A \rightarrow F$ in the BN and PT shown in Figure 4.5.

Algorithm 4.6 $reverseArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$

```
Input: BN $\mathcal{B}$, PT $\mathcal{P}$, output node $X_{out}$, input node $X_{in}$
Output: Updated BN $\mathcal{B}'$ and PT $\mathcal{P}'$
1: $\mathcal{P}_0,\mathcal{B}_0 \leftarrow deleteArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$
2: $\mathcal{P}',\mathcal{B}' \leftarrow addArc(\mathcal{B}_0, \mathcal{P}_0, X_{in}, X_{out})$
3: return $\mathcal{B}', \mathcal{P}'$
```

By Lemma 3 in Appendix A, the arc reversal compilation method never affects the soundness of a PT.

4.3.4 Polynomial tree optimization

So far, we have proposed the methods that are necessary to compile a PT while learning BNs in parallel. The problem is that the PTs obtained after applying one of these methods can have an inference complexity much higher than the optimal PT. This means that an optimization process is required in order to obtain satisfactory PTs, in a way that during the learning process good solutions are not rejected because

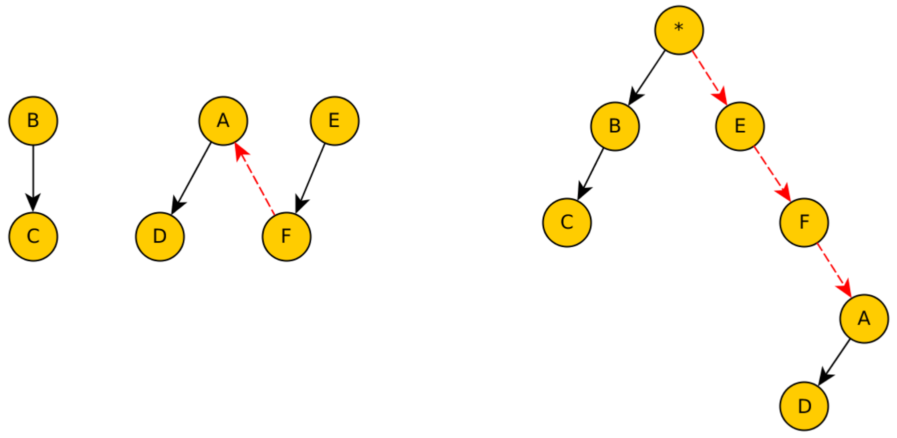


FIGURE 4.10: Example. Arc reversal. BN and PT respectively.

of a poor incremental compilation of the trees. Although the first objective of this process is to obtain optimal trees, it is necessary to consider that a PT should be computed for every candidate in each step of the learning process, and therefore the optimization algorithm will be also applied for each candidate of the tree. It is then essential to use an efficient and accurate method that allows obtaining satisfactory PTs requiring a feasible computational cost.

The method proposed in this master thesis for the optimization of PTs, which is defined by Algorithm 4.7, visits a node per iteration. It receives as input the nodes to optimize $\mathcal{X}_{OPT}$, the maximum number of iterations $nSteps$ and the parameter $alpha$, that is described below. The purpose of $nSteps$ and $alpha$ is to set a threshold in the computational cost of the optimization process, while allowing a satisfactory optimization. The algorithm consists of the next two phases:

- **Phase 1:** In each iteration of the first phase the deepest node belonging to $\mathcal{X}_{OPT}$ that can reduce the complexity of the PT is pushed up. If there is no node which pushing it up supposes a reduction of the complexity of the tree, then the algorithm stops.

- **Phase 2:** In each iteration of the second phase the shallowest node belonging to $\mathcal{X}_{OPT}$ that can reduce the complexity of the PT is pushed up. As it happens in phase 1, the algorithm stops if there are no available movements that reduce the complexity of the tree.

Parameter $alpha$ represents the portion of effort that the algorithm spends in Phase 1. The number of iterations in Phase 1 is set to $nSteps \times alpha$, while the number of iterations in Phase 2 is set to $nSteps \times (1 - alpha)$. If we face a problem where the learning time has a low relevance, it would be better if the parameter *nSteps* is set to a big number, and *alpha* should be close to 1. This would suppose a better fit for each candidate of the tree.

The key of the optimization process is to find the right local movements that improve the network in each iteration. The procedure used here to explore possible changes in one node is defined by Algorithm 4.8. The method *pushUpNode* tries to swap the position of the node to be optimized X_i with its parent $Pa(\mathcal{P}, X_i)$. This method involves movements for the nodes X_i and $Pa(\mathcal{P}, X_i)$ and the children of both of them. Each child of X_i or $Pa(\mathcal{P}, X_i)$ that is not X_i is the ancestor of a branch hanging from X_i or $Pa(\mathcal{P}, X_i)$ that could compromise the soundness of the tree, so the dependences of all the nodes in each branch must be considered.

Let $\mathcal{B}$ and $\mathcal{P}$ be the BN and PT received as an input by *pushUpNode* and $\mathcal{P}'$ the PT that it returns. We will sometimes refer to $Pa(\mathcal{P}, X_i)$ as P_i and to $Pa(\mathcal{P}, Pa(\mathcal{P}, X_i))$ as P_p. Let us use the section of the PT shown in Figure 4.11 as an example to show the operations performed by *pushUpNode*. In the first step of the procedure, all the arcs that join X_i and its parent with the branches hanging from them are deleted. The arc that joins $Pa(\mathcal{P}, X_i)$ with its parent is also deleted. This step is represented in Figure 4.12.

In the second step of the algorithm (Figure 4.13) the new parent of X_i is set, so $Pa(\mathcal{P}, Pa(\mathcal{P}, X_i))$ is the parent of X_i in $\mathcal{P}'$. In the third step (Figure 4.14), the arcs from X_i and $Pa(\mathcal{P}, X_i)$ to the unassigned branches are added. The branches that contain any node that is a descendant of $Pa(\mathcal{P}, X_i)$ in the BN must now hang from $Pa(\mathcal{P}, X_i)$, while the rest of the branches should hang from X_i to reduce the inference complexity of the tree.

The last step required is to assign the new parent of $Pa(\mathcal{P}, X_i)$. Here, there are two options, depending on if $Pa(\mathcal{P}, X_i)$ or any of its descendants in $\mathcal{P}$ is also a descendant of X_i in $\mathcal{B}$. If this holds, then it is necessary to set X_i as the new parent of $Pa(\mathcal{P}, X_i)$, as shown in Figure 4.15. Otherwise, $Pa(\mathcal{P}', X_i)$ keeps its old parent from $\mathcal{P}$ (Figure 4.16).

Algorithm 4.7 *optimize*($\mathcal{B}, \mathcal{P}, \mathcal{X}_{OPT}, nSteps, alpha$) - Part I

Input: BN $\mathcal{B}$, PT $\mathcal{P}$, nodes to optimize $\mathcal{X}_{OPT} = \{X_1, \ldots, X_n\}$, maximum number of iterations *nSteps*, portion of down steps *alpha*

Output: Optimized PT $\mathcal{P}'$

```
 1: let $\mathcal{P}'$ be a copy of $\mathcal{P}$
 2: let $flagsChange$ be an empty list
 3: let $depths$ be an empty list
 4: let $\mathcal{X}' := \{X'_1, \ldots, X'_n\}$ be a copy of $\mathcal{X}_{OPT}$
 5:                                    ▷ Initialization of change flags and depth flags.
 6: for $X_i \in \mathcal{X}'$ do
 7:     append $True$ to $flagsChange$
 8:     let $depth$ be the depth of $X_i$ in $\mathcal{P}'$
 9:     append $depth$ to $depths$
10: end for
11: $nStepsDown := \lfloor nSteps \times alpha \rfloor$          ▷ Number of down steps.
12: $iteration = 0$
13: while $iteration < nSteps$ do
14:     $bestNode \leftarrow None$
15:     if $iteration < nStepsDown$ then
16:         let $bestNode$ be the shallowest node $X'_j$ where $flagsChange[j] = True$
17:     else
18:         let $bestNode$ be the deepest node $X'_j$ where $flagsChange[j] = True$
19:     end if
20:     if $bestNode = None$ then
21:         return $\mathcal{P}'$
22:     end if
23:     $parent \leftarrow Pa(\mathcal{P}', bestNode)$
24:     $\mathcal{P}', flag \leftarrow pushUpNode(\mathcal{B}, \mathcal{P}', bestNode)$
25:     if $flag = True$ then                                ▷ Update flags.
26:         $iteration := iteration + 1$
27:         if $parent \in \mathcal{X}'$ then
28:             $flagChange[parent] := False$
29:         end if
30:         for $X_j \in Ch(\mathcal{P}', parent)$ do
31:             if $X_j \in \mathcal{X}'$ then
32:                 $flagChange[X_j] := True$
```

Algorithm 4.7 *optimize*$(\mathcal{B}, \mathcal{P}, \mathcal{X}_{OPT}, nSteps, alpha)$ - Part II

```
33:             else
34:                 $\mathcal{X}' \leftarrow \mathcal{X}' \cup \{X_j\}$
35:                 append $True$ to $flagChange$
36:                 let $depth$ be the depth of $X_j$ in $\mathcal{P}'$
37:                 append $depth$ to $depths$
38:             end if
39:         end for
40:     else
41:         $flagChange[bestNode] := False$
42:     end if
43: end while
```

Algorithm 4.8 *pushUpNode*$(\mathcal{B}, \mathcal{P}, X_i)$ - Part I

Input: BN $\mathcal{B}$, PT $\mathcal{P}$, node to be moved X_i
Output: PT $\mathcal{P}'$, $flag$ that indicates if a change was made

```
 1: let $\mathcal{P}'$ be a copy $P$
 2: $\mathcal{P}_i \leftarrow Pa(\mathcal{P}', X_i)$
 3:                 ▷ If the parent node is a parent of $X_i$ in the BN return $False$
 4: if $\mathcal{P}_i \in Pa(\mathcal{B}, X_i)$ then
 5:     return $\mathcal{P}, False$
 6: end if
 7: let $dscs$ be an empty list
 8: let $branches$ be an empty list
 9:     ▷ Check if the branches hanging from $X_i$ contain descendants of $\mathcal{P}_i$ in $\mathcal{B}$
10: for $X_j \in Ch(\mathcal{P}', X_i)$ do
11:     let $dsc$ be $True$ if $X_j$ or any of its descendants $\mathcal{P}'$ is a descendant of $\mathcal{P}_i$ in
        $\mathcal{B}$ and $False$ otherwise
12:     append $dsc$ to $dscs$
13:     append $X_j$ to $branches$
14: end for
15:     ▷ Check if the branches hanging from $\mathcal{P}_i$ contain descendants of $\mathcal{P}_i$ in $\mathcal{B}$
16: for $X_j \in Ch(\mathcal{P}', \mathcal{P}_i)$ do
17:     let $dsc$ be $True$ if $X_j$ or any of its descendants $\mathcal{P}'$ is a descendant of $\mathcal{P}_i$ in
        $\mathcal{B}$ and $False$ otherwise
18:     append $dsc$ to $dscs$
19:     append $X_j$ to $branches$
20: end for
21:                     ▷ Set the parents of each branch belonging to $branches$
22: $Pa(\mathcal{P}', X_i) \leftarrow Pa(\mathcal{P}', \mathcal{P}_i)$
23: let $len$ be the length of $branches$
```

Algorithm 4.8 $pushUpNode(\mathcal{B}, \mathcal{P}, X_i)$ - Part II

```
24: for $j = 1, \dots, len$ do
25:     if $dscs[j] = True$ then
26:         $Pa(\mathcal{P}', branches[j]) \leftarrow \mathcal{P}_i$
27:     else
28:         $Pa(\mathcal{P}', branches[j]) \leftarrow X_i$
29:     end if
30: end for
31: if $\mathcal{P}_i$ is a predecessor of $X_i$ in $\mathcal{B}$ then
32:     $Pa(\mathcal{P}', \mathcal{P}_i) \leftarrow X_i$
33: end if
34:                   ▷ Compare the complexity of the old network and the new one
35: $eOld := evalNode(\mathcal{P}, \mathcal{P}_i)$
36: $eNew := evalNode(\mathcal{P}, X_i)$
37: if $eNew < eOlds$ then
38:     $\mathcal{P} \leftarrow \mathcal{P}'$
39:     return $\mathcal{P}', True$
40: else
41:     return $\mathcal{P}, False$
42: end if
```

As it is demonstrated in Lemma 4 in Appendix A, *pushUpNode* method, and therefore *optimize* method presented in this section does not alter the soundness of any PT.

4.4 Learning polynomial trees from data

The previous section described a group of methods that allow the incremental compilation of PTs considering the addition, deletion and reversal of arcs in the BNs. Therefore, we have the tools required to learn PTs in parallel with BNs. The framework provided for the incremental compilation of PTs has been made with the objective of making it flexible and adaptable to most of the score+search methods known to date. It is straightforward to add the compilation step to any method that applies local changes during the search process, such as greedy search methods or stochastic methods.

The methods for incremental compilation were created with the purpose of obtaining satisfactory trees efficiently in each compilation step. This approach matches perfectly with greedy methods like K2 and HC, but some stochastic or evolutionary methods may require a more refined optimization process given that a large amount

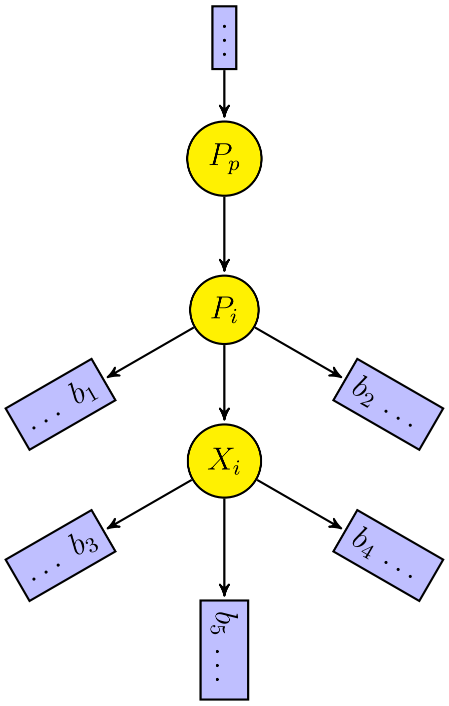


FIGURE 4.11: Section of a PT

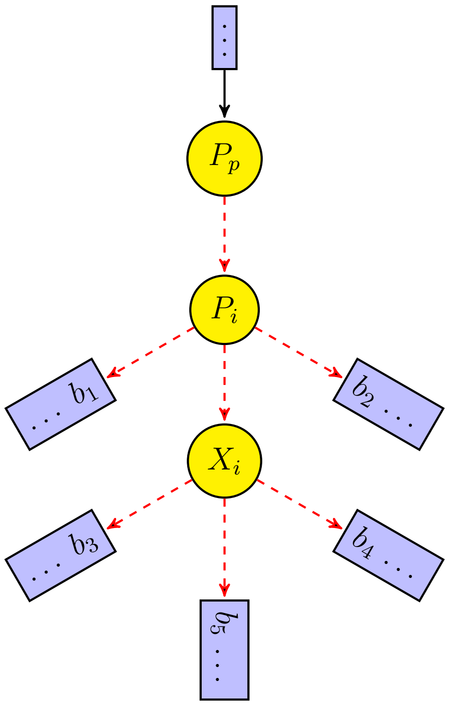


FIGURE 4.12: Step 1

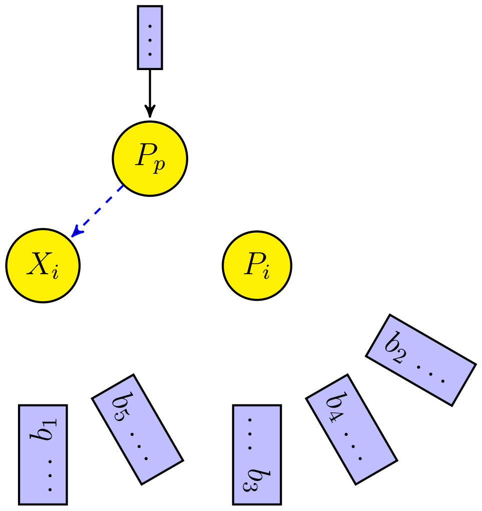


FIGURE 4.13: Step 2

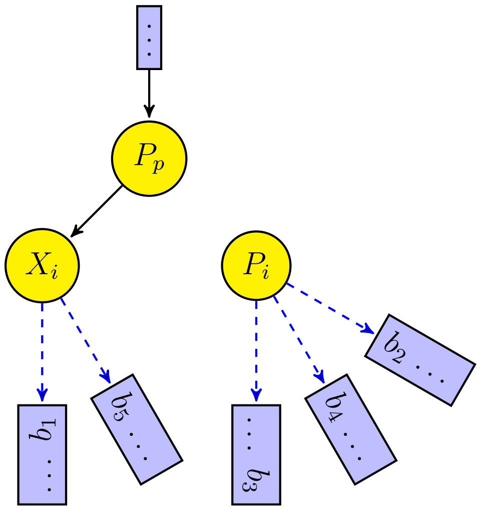


FIGURE 4.14: Step 3

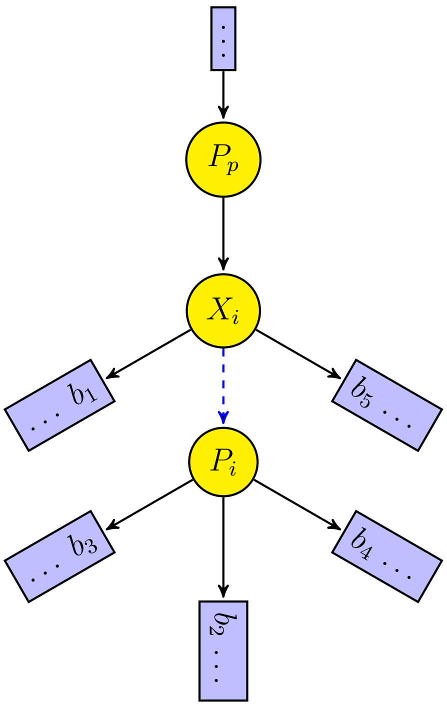


FIGURE 4.15: Step 4a

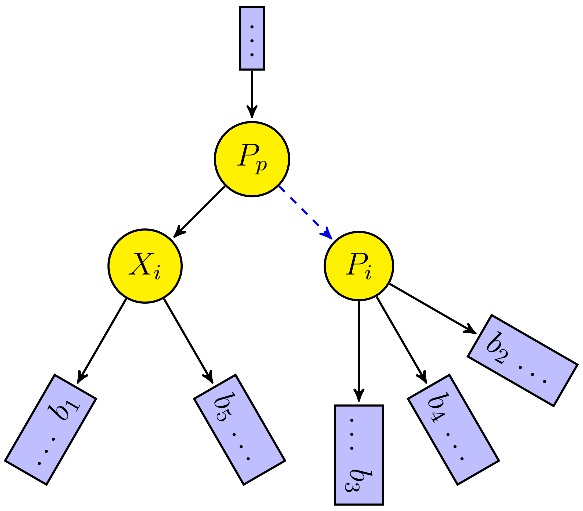


FIGURE 4.16: Step 4b

of changes over the same candidate could create a noticeable gap between the optimal PT for the candidate and the current one. The computational time required for a refined optimization of PTs should be tractable for methods that manage a relatively small number of candidates, but it would carry an expensive computational cost for greedy methods that explore all the possible local changes in each iteration of the algorithm. In this section, a method for learning PTs is proposed.

4.5 Scoring function

The objective of the metric proposed in this work for learning PTs is to measure the accuracy and the inference complexity of the model. The method *evalNode* introduced in the previous sections returns the maximum number of operations n_{op} required to answer any joint query in a PT. This is a very appropriate inference complexity indicator, and here we use it as a penalization term in the scoring function.

We use the log-likelihood metric to measure the accuracy of the model. Although other information-theory or Bayesian metrics could be used instead, the log-likelihood fits specially well with our inference complexity indicator. The reason is that most metrics include an implicit or explicit penalization for the representational complexity of the learned model, while log-likelihood does not include any penalization. This property of log-likelihood, that usually causes overfitting in the learned models, is desirable here because we penalize log-likelihood with the complexity of inference.

Although using n_{op} penalizes implicitly the number of parameters in a network, if an arc addition $X_{out} \rightarrow X_{in}$ is compiled in a PT $\mathcal{P}$, and X_{out} is currently a predecessor of X_{in} in P (scenario 1 of *addArc*), then the penalization of n_{op} to this change is too low, which could suppose the addition of unnecessary arcs. Therefore, a penalization to the number of parameters is also included in the scoring function. The score is defined in Equation (4.4).

$$scorePT(\mathcal{B}, \mathcal{P}, D) = LL(\mathcal{B}, D) - k_n \cdot \log(N) \cdot n_{op} - k_p \cdot \log(N) \cdot |\mathcal{B}| \tag{4.4}$$

Where the parameters k_n and k_p represent the weight of n_{opt} and of the number of parameters $|\mathcal{B}| = \sum_{i=1}^{n}(r_i - 1)q_i$ respectively for the model penalization.

4.6 Hill-climbing for polynomial trees

For this master thesis, we have adapted the HC algorithm, which is widely used for learning the structure of BNs in the space of DAGs. In particular, we have used a version of HC that is close to the 2iCHC method, using a forbidden parents list in two iterations. We call this method *hill-climbing for polynomial trees (HCPT)*. It proceeds as follows. First, the BN and PT are initialized assuming full independence among the variables. In the case of the initial PT $\mathcal{P}_0$, the full independence is represented by the tree where all the nodes in the network hang from the root node, encoding the products of marginals $P(\mathcal{P}_0) = \prod_{i=1}^{n} \sum_{j=1}^{r_i} P(X_i = x_{ij}) I(X_i = x_{ij})$. Then, in each step the same local changes are applied to the BN and the PT for each candidate, using the incremental compilation methods explained before. Each candidate is evaluated using the scoring function described in the previous section, that measures the accuracy of the candidate, penalizing those PTs with a large inference complexity.

The method presented in Algorithm 4.9 is similar to the 2iCHC algorithm, that learns the structure of BNs using a forbidden parents (FP) list to constrain the search space. HCPT performs two iterations of Algorithm 4.10 to achieve a better convergence to the probability distribution of the data. The algorithm searches among all possible additions, deletions and reversals of arcs in each iteration and applies the change that maximizes the score of the model. The procedure responsible of finding the best candidate in each iteration is $bestPred_{HCPT}$ (Algorithm 4.11).

The FP list used in HCPT is updated in a different way from the used in 2iCHC. This decision was made due to the inclusion of the inference complexity in the metric $scorePT$. In 2iCHC, if the addition of an arc $X_a \to X_b$ worsens the score or the deletion of an arc $X_a \to X_b$ improves the score, then X_a is included to $FP(X_b)$ and X_b is included to $FP(X_a)$. Using $scorePT$, the addition or deletion of an arc $X_a \to X_b$ can carry an extremely different computational cost than doing the same operation with the arc $X_b \to X_a$, so it would not be correct if we constrain both movements when only one of them has a worse score.

Now, it is essential to know that the algorithm HCPT learns a sound PT, so it can be used afterwards to perform inference without the risk of crashing. Given that HCPT starts from scratch and only uses algorithms *addArc*, *reverseArc*, *deleteArc* and *optimize* to make changes in the learned PT and BN, then by Theorem 1 the PTs learned by HCPT are always sound. Theorem 1 is demonstrated in Appendix A.

Theorem 1. *Let Alg be any algorithm for learning in parallel a PT $\mathcal{P}$ and a BN $\mathcal{B}$ from scratch that, to make any change in $\mathcal{P}$ or $\mathcal{B}$ during the learning process, only uses the methods addArc, reversArc, deleteArc or optimize. A PT $\mathcal{P}'$ is always sound regarding to a Bayesian network $\mathcal{B}'$ if $\mathcal{P}'$ and $\mathcal{B}'$ are the output of Alg.*

Algorithm 4.9 $HCPT(\mathcal{X}, D)$

Input: Set of nodes $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, Data D

Output: BN $\mathcal{B}'$, PT $\mathcal{P}'$

```
1: let $B$ be the empty network over $\mathcal{X}$
2: let $P$ be the product of marginals over $\mathcal{X}$
3: $B_0, P_0 \leftarrow HCPT_1(\mathcal{B}, \mathcal{P}, D)$
4: $\mathcal{B}', \mathcal{P}' \leftarrow HCPT_1(B_0, P_0, D)$
5: return $\mathcal{B}', \mathcal{P}'$
```

Algorithm 4.10 $HCPT_1(\mathcal{B}, \mathcal{P}, D)$

Input: BN $\mathcal{B}$ over $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, PT $\mathcal{P}$ over $\mathcal{X} \cup \{*\}$, Data D

Output: BN $\mathcal{B}'$, PT $\mathcal{P}'$

```
 1: $S_{old} \leftarrow scorePT(\mathcal{B}, \mathcal{P}, D)$
 2: let $\mathcal{B}'$ be a copy of $\mathcal{B}$
 3: let $FP$ be a list of $n$ empty lists
 4: $OKToProceed := True$
 5: while $OKToProceed = True$ do
 6:     $\mathcal{B}_{new}, \mathcal{P}_{new}, S_{new} \leftarrow bestPred_{HCPT}(\mathcal{B}, \mathcal{P}, D, S_{old}, FP)$
 7:     if $S_{new} > S_{old}$ then
 8:         $\mathcal{B}' \leftarrow \mathcal{B}_{new}$
 9:         $\mathcal{P}' \leftarrow \mathcal{P}_{new}$
10:         $S_{old} \leftarrow S_{new}$
11:     else
12:         $OKToProceed := False$
13:     end if
14: end while
15: return $\mathcal{B}', \mathcal{P}'$
```

Algorithm 4.11 $bestPred_{HCPT}(\mathcal{B}, \mathcal{P}, D, S_{old}, FP)$

Input: BN $\mathcal{B}$, PT $\mathcal{P}$, Data D, score S_{old}, forbidden parents FP

Output: Best BN $\mathcal{B}_{best}$, Best PT $\mathcal{P}_{best}$, Best score S_{best}

```
1: let changes be the list of local changes that could be made to $\mathcal{B}$
2: let $\mathcal{B}_{best}$ be a copy of $\mathcal{B}$
3: let $\mathcal{P}_{best}$ be a copy of $\mathcal{P}$
4: let $S_{best}$ be a copy of $S_{old}$
5: for change in changes do
6:     let $S_{new}$ be a copy of $S_{old}$
7:     if change is the addition $X_a \to X_b$ then
8:         $\mathcal{B}_{new}, \mathcal{P}_{new} \leftarrow addArc(\mathcal{B}, \mathcal{P}, X_a, X_b)$
9:         $S_{new} \leftarrow scorePT(\mathcal{B}_{new}, \mathcal{P}_{new}, D)$
10:        if $S_{new} < S_{old}$ then
11:            add $X_a$ to $FP(X_b)$
12:        end if
13:    else if change is the deletion of $X_a \to X_b$ then
14:        $\mathcal{B}_{new}, \mathcal{P}_{new} \leftarrow deleteArc(\mathcal{B}, \mathcal{P}, X_a, X_b)$
15:        $S_{new} \leftarrow scorePT(\mathcal{B}_{new}, \mathcal{P}_{new}, D)$
16:        if $S_{new} > S_{old}$ then
17:            add $X_a$ to $FP(X_b)$
18:        end if
19:    else if change is the reversal of $X_a \to X_b$ then
20:        $\mathcal{B}_{new}, \mathcal{P}_{new} \leftarrow reverseArc(\mathcal{B}, \mathcal{P}, X_a, X_b)$
21:        $S_{new} \leftarrow scorePT(\mathcal{B}_{new}, \mathcal{P}_{new}, D)$
22:        if $S_{new} < S_{old}$ then
23:            add $X_b$ to $FP(X_a)$
24:        end if
25:    end if
26:    if $\mathcal{P}_{new} > \mathcal{P}_{best}$ then
27:        $\mathcal{B}_{best} \leftarrow \mathcal{B}_{new}$
28:        $\mathcal{P}_{best} \leftarrow \mathcal{P}_{new}$
29:        $S_{best} \leftarrow S_{new}$
30:    end if
31: end for
32: return $\mathcal{B}_{best}$,$\mathcal{P}_{best}$,$S_{best}$
```

Chapter 5

Experimental Results

The purpose of this chapter is to show and discuss the results obtained for inference and learning using PTs. In particular, using the method HCPT presented in the previous chapter. The idea is to compare the impact of including the PT framework to the original method, in this case 2iHC, and compare the accuracy of inference in both models, checking if exact inference in the model learned by HCPT is now tractable.

For this, it is necessary to choose a group of datasets for learning and testing the networks. The datasets used in this report were generated from varied real-world BNs. Using well known BNs allows having always a satisfactory number of samples for the learning and testing processes and reduces the risks of obtaining biased results.

The BNs used for generating the training and testing datasets were all obtained from the bnlearn repository:

- ALARM is a medium size network. It is the representation of a monitoring system (Beinlich et al., 1989).
- HEPAR II is a large network for the diagnosis of liver disorders (Onisko, 2003).
- WIN95PTS is a large network for handling printer troubleshooting in Windows 95.

The basic properties of each BN are shown in Table 5.1, while the statistics of the datasets generated (using PLS) from them are shown in Table 5.2.

	ASIA	**HEPAR II**	**WIN95PTS**
Number of nodes	37	70	76
Number of arcs	46	123	112
Number of parameters	509	1453	574
Average Markov blanket size	3.51	4.51	5.92
Average degree	2.49	3.51	2.95
Maximum in-degree	4	6	7

Table 5.1: Basic properties of the BNs used for the experiments.

	ASIA	**HEPAR II**	**WIN95PTS**
Training samples	25000	25000	25000
Test samples	40000	40000	40000

Table 5.2: Sizes of the datasets generated for the experiments.

5.1 Test methodology

To evaluate the inference process with the new method, we need an indicator of the accuracy of the learned model. The KL-divergence (Kullback and Leibler, 1951) is specially useful for obtaining the divergence between two probability distributions, where one is considered as the True model P and the other is considered an approximation Q of this model. The formula of the KL-divergence for discrete distributions is shown in Equation (5.1).

$$D_{KL}(P||Q) = \sum_{x} P(x) \log \frac{P(x)}{Q(x)} \tag{5.1}$$

As computing the KL-Divergence is intractable for the learned models and the datasets used in this master thesis, we have used the *normalized mean log probability (NMLP)* as an approximation to measure the inference accuracy. The *mean log probability* (without normalization) was used in Lowd and Domingos (2008) with the same purpose. The NMLP is obtained using a set of 500 samples from the test data. From each sample a conditional probability query with randomly selected query and evidence variables is generated. The query and evidence variables take the values of their configuration in the sample, and it is asked to P and Q to measure the inference accuracy of Q regarding to P. Equation (5.2) defines the value of the $NMLP(P||Q)$ for m queries.

$$NMLP(P||Q) = \frac{1}{m}\sum_{i=1}^{m}\left|\log\frac{P(q(i)|e(i))}{Q(q(i)|e(i))}\right| \tag{5.2}$$

As a complementary inference accuracy indicator, we have also used the *mean square error (MSE)* between the results obtained performing inference in Q and P. The MSE for m queries is defined in Equation (5.3).

$$MSE(P||Q) = \frac{1}{m}\sum_{i=1}^{m}(P(q(i)|e(i)) - Q(q(i)|e(i)))^2 \tag{5.3}$$

Where $q(i)$ is the instantiation of Q in sample i, and $e(i)$ is the instantiation of E in sample i.

The queries asked to original distribution P are estimated using the test datasets, by computing the probability of the query variables among all the samples in the test datasets that match with the configuration of the evidence variables. The queries asked to Q are obtained by performing inference in the learned model. In this report we compare the results obtained with exact inference in the polynomials trees learned by the $HCPT$ algorithm with the results obtained with approximate inference in the BNs learned by $2iCHC$.

For approximate inference, we use the likelihood-weighting algorithm to estimate the probability of the queries. It is interesting to compare both the accuracy and the inference time for the different methods. For this purpose we use three different sampling sizes for the likelihood weighting: quick (200 samples), medium (1000 samples) and slow (2000 samples). This allows comparing the efficiency and accuracy of the new method against very fast inference procedures and other that are slower but achieve a better convergence to the target probability distribution.

5.2 Learning Results

First, we compare the results obtained after the learning process for each procedure. The networks obtained by the incremental compilation of PTs are learned by the HCPT method, and they are compared with the BNs learned by the 2iCHC algorithm in combination with the MDL scoring function. The 2iCHC is an state-of-the-art algorithm that allows learning large BNs in a tractable time with an accuracy that is close to the HC algorithm. For the HCPT method, the parameters k_n and k_p were set to 0.5 and 0.15 respectively.

As it is shown in Chapter 4, HCPT is an adaptation of the 2iCHC algorithm. Given that the main purpose of the work is to provide a framework for the incremental compilation of PTs that could be easily applied in most score+search methods, it is of special interest to compare HCPT with 2iCHC. For each network, we show the log-likelihood per sample of each model against the test dataset, the number of arcs of the network, the maximum number of parameters for a variable (*maxParents*), the average number of parameters per variable (*avgParents*), the total number of parameters in the network (*nParents*), and the time required by the learning method to learn the network. See Tables 5.3-5.5.

ALARM	*HC*	*HC_PT*
Log_Likelihood	−10.47	−10.75
arcs	50	49
maxParents	3	3
avgParents	1.351	1.324
nParams	270	248
Time	0 h 11 m	0 h 7 m

TABLE 5.3: Learning results for ALARM

WIN95PTS	*HC_PT*	*HC*
Log_Likelihood	−9.97	−9.04
arcs	120	118
maxParents	4	7
avgParents	1.579	1.553
nParams	313	489
Time	0 h 52 m	1 h 1 m

TABLE 5.4: Learning results for WIN95PTS

HEPAR II	*HC_PT*	*HC*
Log_Likelihood	−32.69	−32.59
arcs	123	88
maxParents	4	3
avgParents	1.757	1.257
nParams	446	284
Time	0 h 27 m	0 h 33 m

TABLE 5.5: Learning results for HEPAR II

Although the faithfulness of the models is studied with the inference results, we use the likelihood between the networks and the test data as a first indicator. It was expected that the likelihood of the models was better for those networks learned by 2iCHC than the ones learned by HCPT, given that HCPT penalizes the movements that cause a relevant increment in the inference complexity of the network. The results show that the differences in the likelihood are small. The number of parameters is smaller in ALARM and WIN95PTS for HCPT, mainly because the weight of the inference complexity in the score is set in a way that it favours learning thin models for a very efficient inference process. The number of parameters in HEPAR II is higher for HCPT. The number of parameters is a representational complexity measure, and a model with a lower representational complexity can have a higher inference complexity.

The times required for the learning process are similar for both 2iCHC and HCPT. Although in general they are slightly better for HCPT, it is mainly because it learns thinner models, and if the weight of the inference complexity was decremented the learning time of HCPT should be higher. Nevertheless, this shows that the time fraction used for the incremental compilation and the evaluation of PTs is small compared with the time spent by the scores.

5.3 Inference Results

In this section we compare the inference results obtained with exact inference over the models learned using the HCPT method against the inference results obtained with approximate inference over the BNs learned with the 2iCHC algorithm. We use the LW algorithm for approximate inference.

We generate sets of queries from the test dataset. The queries are then computed in each model and the results are compared with the probability of the queries in the test dataset using the NMLP and the MSE as the measure of inference accuracy between the learned models and the test data.

The results for each network are presented in tables. Each table contains the results for a network and a fixed number of query or evidence variables, and shows the following information:

- **Method:** Methods used to learn the PT and BN. In the case of the algorithms learned with 2iCHC there is also a specification of the number of samples used for approximate reasoning.
- **Num Q / Num E:** Number of query and evidence variables respectively.
- **NMPL:** Normalized mean log probability.
- **MSE:** Mean square error.
- **Mean time:** Mean time per query in seconds.

We also include a chart to show graphically the inference accuracy results shown in the tables. Tables 5.6-5.7 and Figures 5.1-5.4 show the inference results for ALARM network, Tables 5.7-5.8 and Figures 5.5-5.8 correspond to the inference results for HEPAR II network, and Tables 5.9-5.10 and Figures 5.9-5.12 show the inference results for WIN95PTS network.

Method	Num E (%)	NMLP	MSE	Mean Time (s)
HC_Quick	10	0.066	0.01	0.1214
HC_Quick	15	0.079	0.019	0.1179
HC_Quick	20	0.088	0.019	0.1157
HC_Quick	25	0.1	0.032	0.1136
HC_Medium	10	0.06	0.009	0.6065
HC_Medium	15	0.073	0.018	0.5997
HC_Medium	20	0.085	0.02	0.589
HC_Medium	25	0.092	0.031	0.5817
HC_Slow	10	0.06	0.009	1.2315
HC_Slow	15	0.071	0.017	1.2141
HC_Slow	20	0.086	0.019	1.1965
HC_Slow	25	0.089	0.031	1.1828
HC_PT	10	0.028	0.001	0.054
HC_PT	15	0.026	0.002	0.0493
HC_PT	20	0.04	0.005	0.0455
HC_PT	25	0.027	0.004	0.0415

Table 5.6: Inference in ALARM. 15 % query variables

Method	Num Q (%)	NMLP	MSE	Mean Time (s)
HC_Quick	10	0.097	0.022	0.117
HC_Quick	15	0.084	0.02	0.1173
HC_Quick	20	0.081	0.022	0.1191
HC_Quick	25	0.058	0.012	0.119
HC_Medium	10	0.092	0.021	0.5958
HC_Medium	15	0.084	0.019	0.5988
HC_Medium	20	0.08	0.023	0.6222
HC_Medium	25	0.057	0.01	0.5998
HC_Slow	10	0.087	0.021	1.2077
HC_Slow	15	0.082	0.018	1.2102
HC_Slow	20	0.08	0.022	1.2159
HC_Slow	25	0.055	0.01	1.2179
HC_PT	10	0.036	0.004	0.0527
HC_PT	15	0.031	0.003	0.0493
HC_PT	20	0.03	0.004	0.0463
HC_PT	25	0.024	0.002	0.0448

Table 5.7: Inference in ALARM. 15 % evidence variables

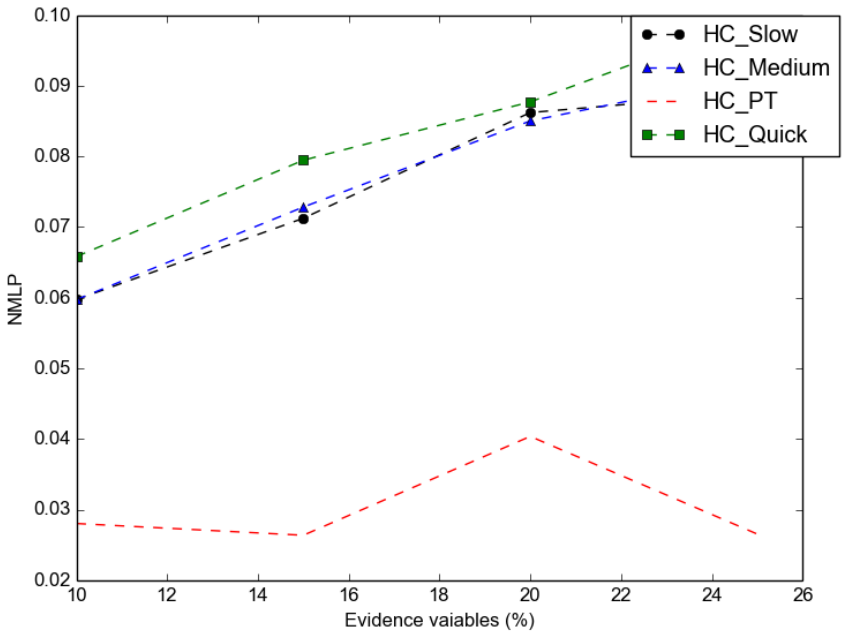


FIGURE 5.1: Inference in ALARM. NMLP for 15 % query variables

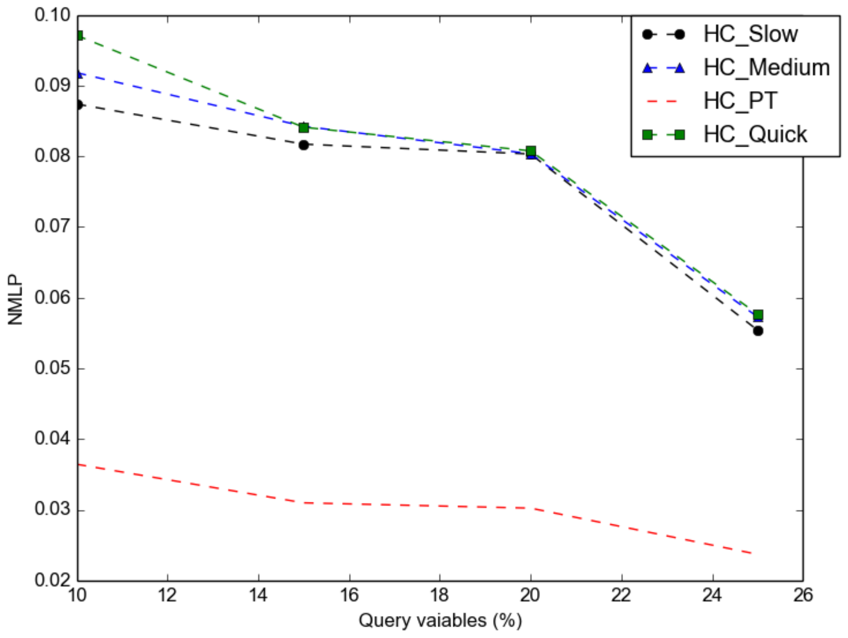


FIGURE 5.2: Inference in ALARM. NMLP for 15 % evidence variables

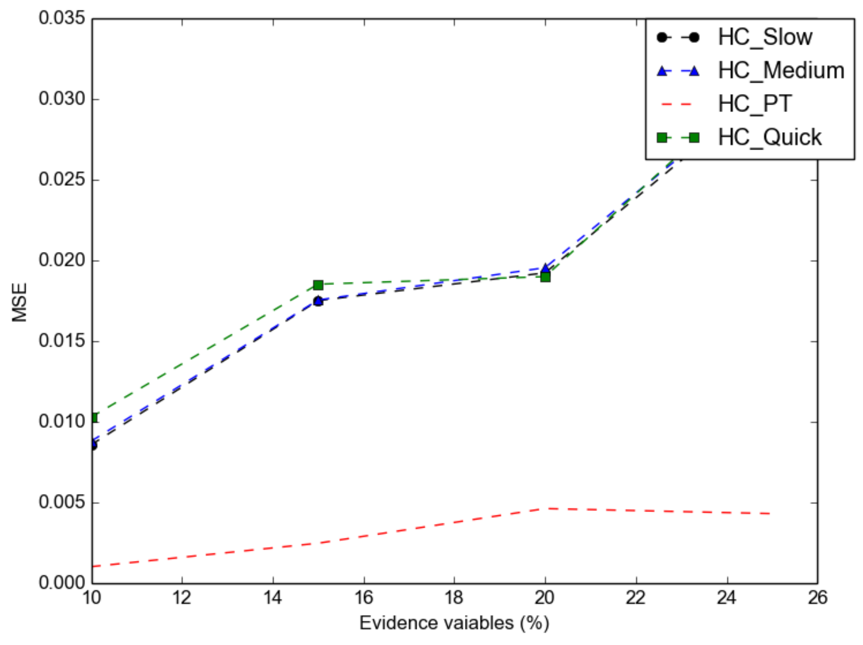


FIGURE 5.3: Inference in ALARM. MSE for 15 % query variables

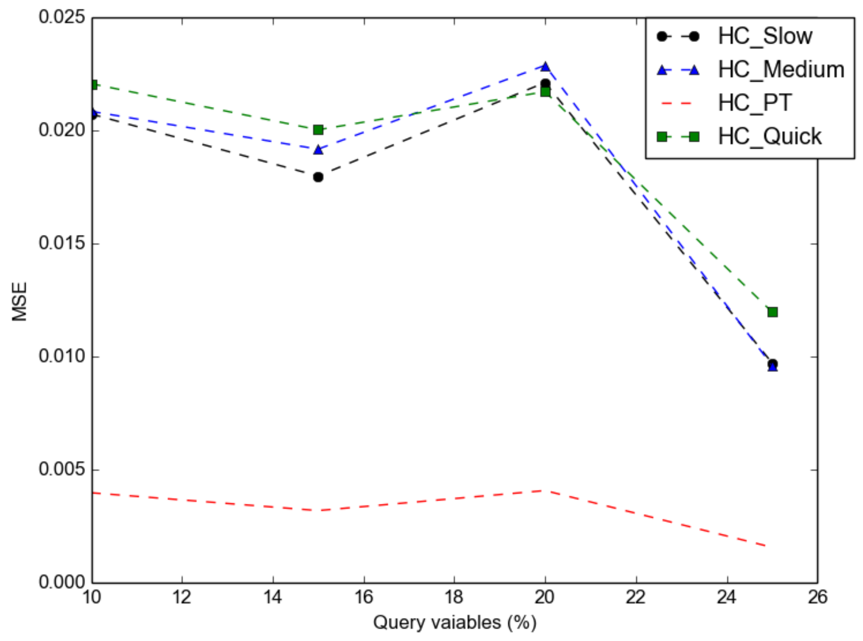


FIGURE 5.4: Inference in ALARM. MSE for 15 % evidence variables

Method	Num E (%)	NMLP	MSE	Mean Time (s)
HC_Quick	10	0.048	0.0	0.2274
HC_Quick	15	0.053	0.006	0.2266
HC_Quick	20	0.058	0.006	0.2277
HC_Quick	25	0.071	0.018	0.2235
HC_Medium	10	0.036	0.0	1.1488
HC_Medium	15	0.049	0.006	1.1431
HC_Medium	20	0.059	0.005	1.1435
HC_Medium	25	0.073	0.017	1.129
HC_Slow	10	0.036	0.0	2.3094
HC_Slow	15	0.044	0.006	2.2848
HC_Slow	20	0.059	0.005	2.2939
HC_Slow	25	0.071	0.017	2.2582
HC_PT	10	0.026	0.0	0.047
HC_PT	15	0.041	0.006	0.044
HC_PT	20	0.051	0.005	0.0439
HC_PT	25	0.061	0.015	0.0399

Table 5.8: Inference in HEPAR II. 15 % query variables

Method	Num Q (%)	NMLP	MSE	Mean Time (s)
HC_Quick	10	0.063	0.003	0.2252
HC_Quick	15	0.051	0.001	0.2258
HC_Quick	20	0.051	0.001	0.2251
HC_Quick	25	0.034	0.0	0.2303
HC_Medium	10	0.052	0.002	1.1413
HC_Medium	15	0.049	0.001	1.145
HC_Medium	20	0.048	0.001	1.1386
HC_Medium	25	0.039	0.0	1.1615
HC_Slow	10	0.051	0.002	2.282
HC_Slow	15	0.047	0.001	2.5577
HC_Slow	20	0.046	0.001	2.3005
HC_Slow	25	0.037	0.0	2.3501
HC_PT	10	0.041	0.002	0.0454
HC_PT	15	0.039	0.0	0.0448
HC_PT	20	0.042	0.001	0.0425
HC_PT	25	0.045	0.0	0.0447

Table 5.9: Inference in HEPAR II. 15 % evidence variables

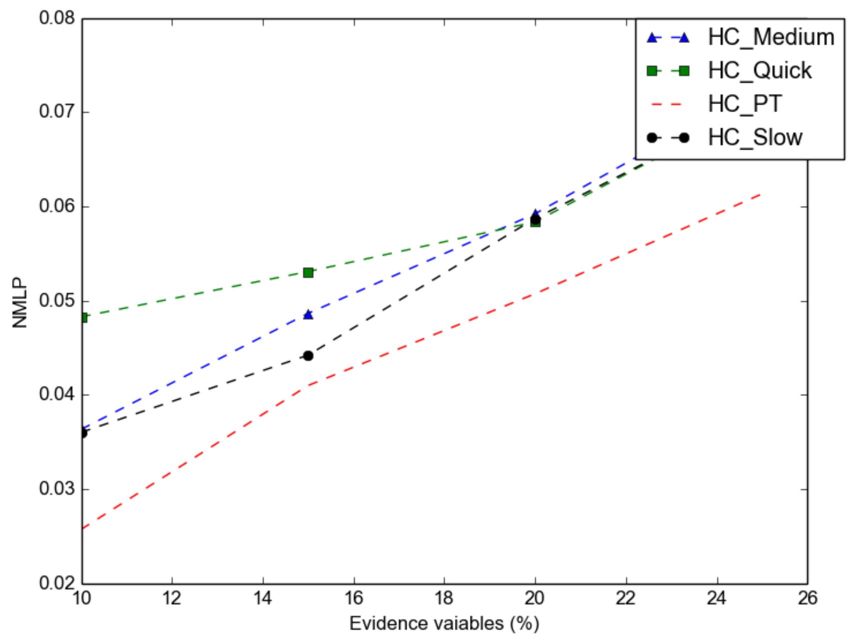


FIGURE 5.5: Inference in HEPAR II. NMLP for 15 % query variables

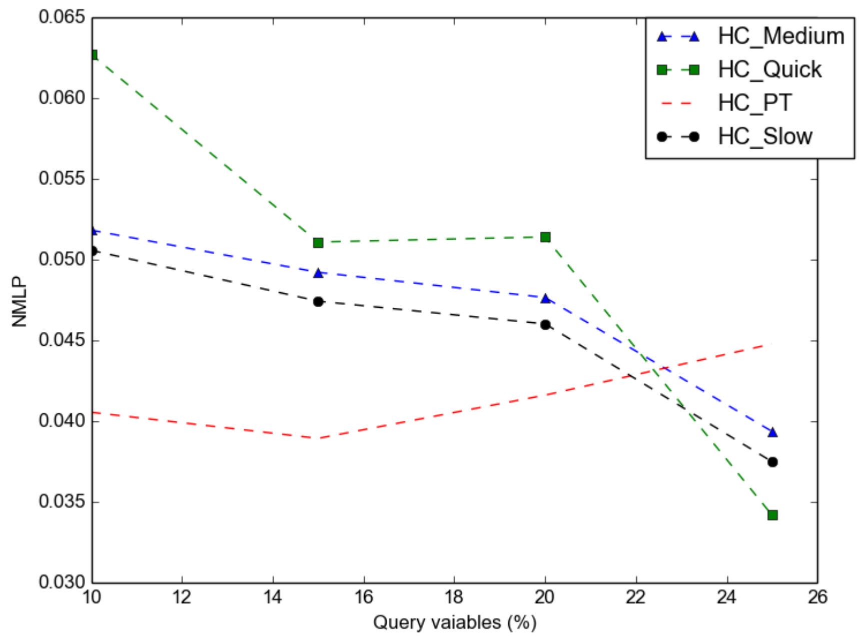


FIGURE 5.6: Inference in HEPAR II. NMLP for 15 % evidence variables

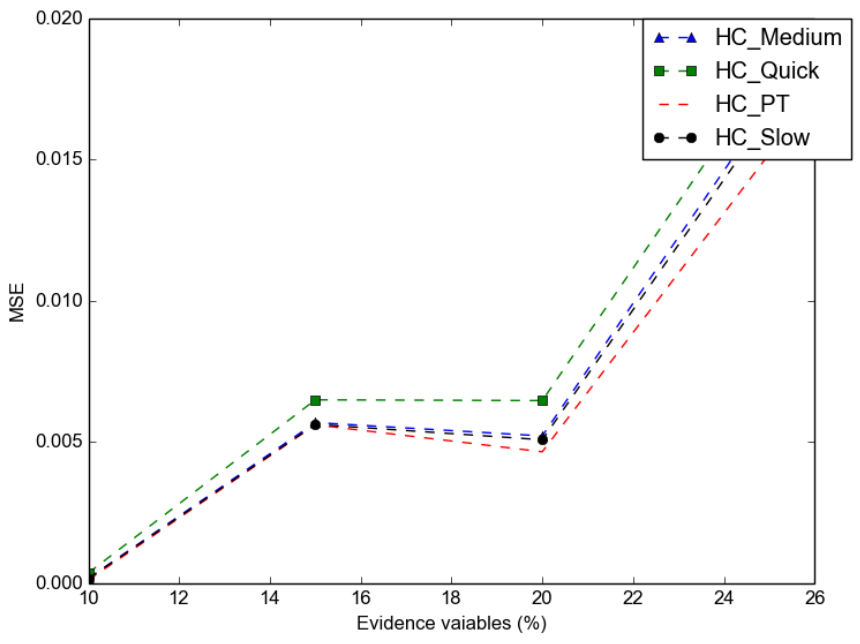


FIGURE 5.7: Inference in HEPAR II. MSE for 15 % query variables

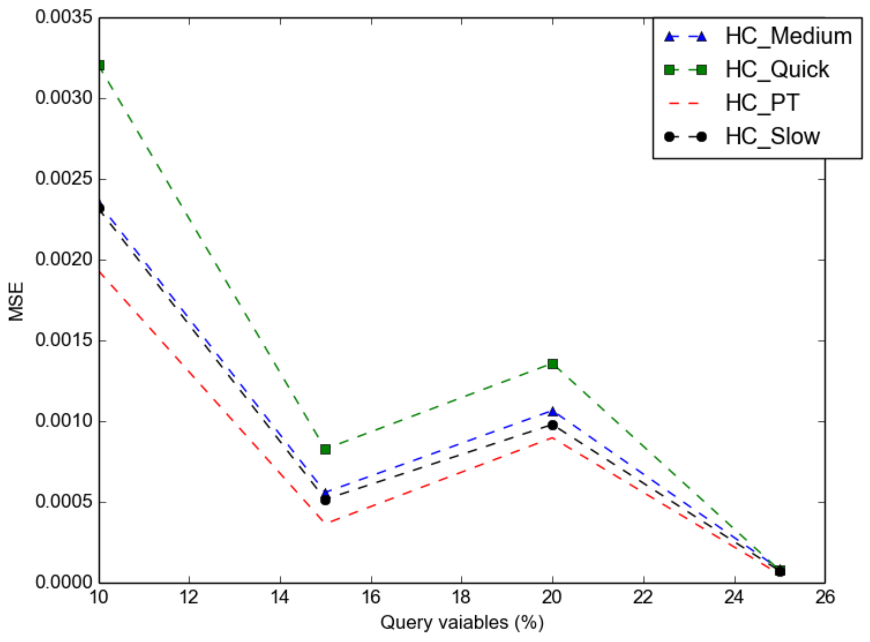


FIGURE 5.8: Inference in HEPAR II. MSE for 15 % evidence variables

Method	Num E (%)	NMLP	MSE	Mean Time (s)
HC_Quick	10	0.032	0.008	0.3113
HC_Quick	15	0.034	0.018	0.3113
HC_Quick	20	0.042	0.021	0.3088
HC_Quick	25	0.043	0.036	0.304
HC_Medium	10	0.029	0.007	1.5614
HC_Medium	15	0.033	0.016	1.5578
HC_Medium	20	0.04	0.021	1.5397
HC_Medium	25	0.043	0.037	1.5429
HC_Slow	10	0.028	0.007	3.1263
HC_Slow	15	0.032	0.016	3.1595
HC_Slow	20	0.038	0.02	3.2639
HC_Slow	25	0.041	0.037	3.026
HC_PT	10	0.019	0.002	0.0985
HC_PT	15	0.022	0.005	0.0926
HC_PT	20	0.028	0.008	0.0914
HC_PT	25	0.024	0.016	0.1173

Table 5.10: Inference in WIN95PTS. 15 % query variables

Method	Num Q (%)	NMLP	MSE	Mean Time (s)
HC_Quick	10	0.045	0.016	0.3117
HC_Quick	15	0.033	0.013	0.3132
HC_Quick	20	0.027	0.011	0.3134
HC_Quick	25	0.023	0.008	0.3143
HC_Medium	10	0.039	0.014	1.5575
HC_Medium	15	0.029	0.012	1.5594
HC_Medium	20	0.028	0.01	1.566
HC_Medium	25	0.021	0.007	1.5696
HC_Slow	10	0.038	0.014	3.0867
HC_Slow	15	0.028	0.011	3.0872
HC_Slow	20	0.029	0.01	3.085
HC_Slow	25	0.023	0.007	3.075
HC_PT	10	0.024	0.005	0.1253
HC_PT	15	0.02	0.004	0.0961
HC_PT	20	0.02	0.003	0.1443
HC_PT	25	0.017	0.003	0.1457

Table 5.11: Inference in WIN95PTS. 15 % evidence variables

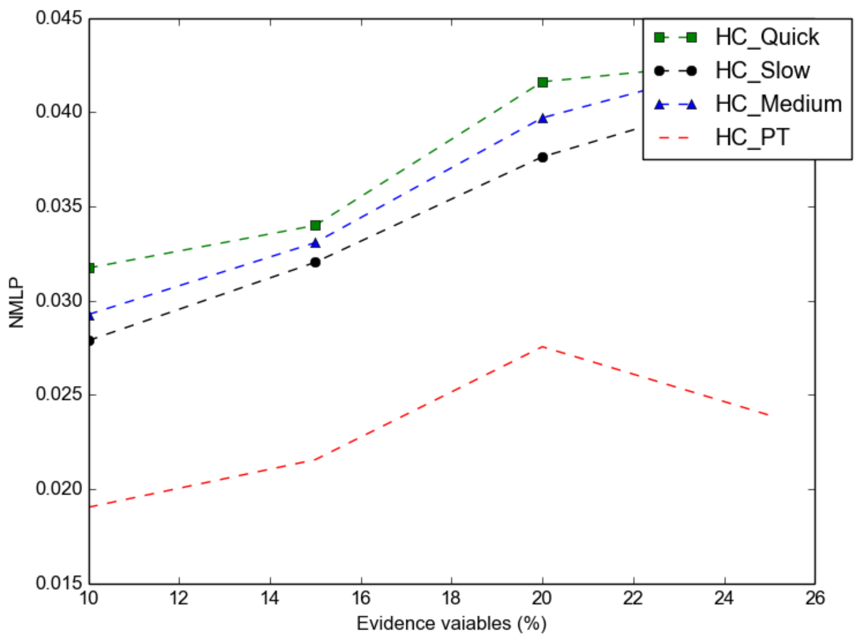


FIGURE 5.9: Inference in WIN95PTS. NMLP for 15 % query variables

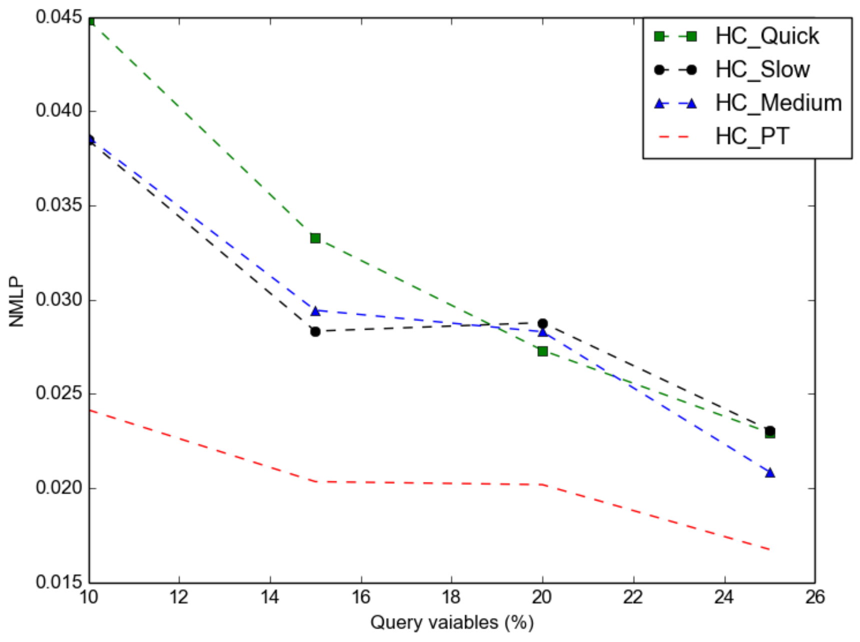


FIGURE 5.10: Inference in WIN95PTS. NMLP for 15 % evidence variables

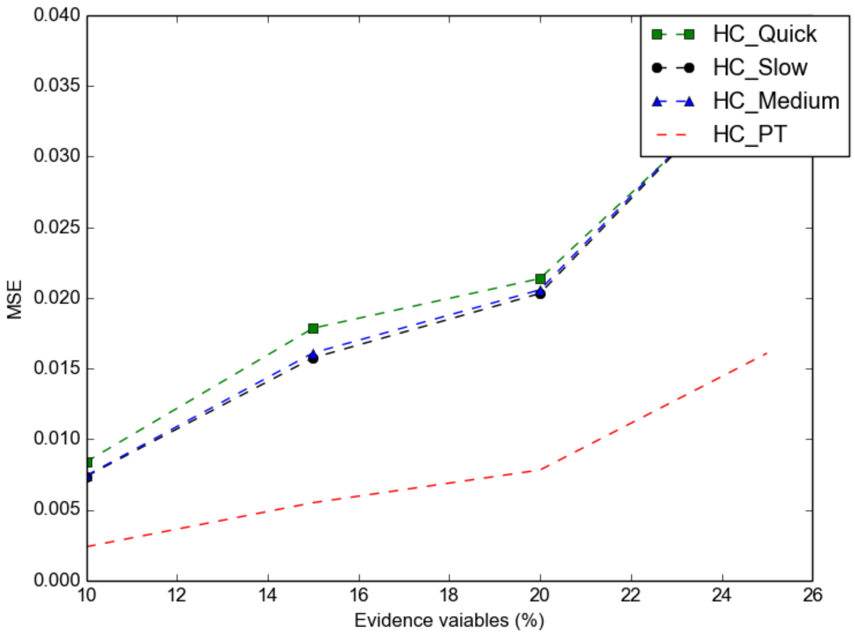


FIGURE 5.11: Inference in WIN95PTS. MSE for 15 % query variables

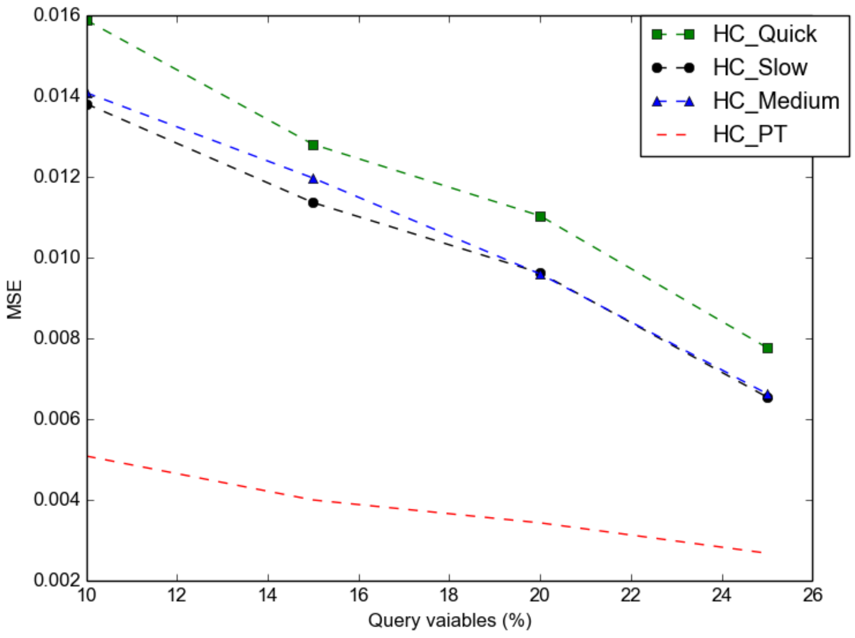


FIGURE 5.12: Inference in WIN95PTS. MSE for 15 % evidence variables

To analyse the results, we need to consider both the inference time and the inference accuracy of the models. Given that our model focus on both learning and inference, it is essential to see if the new model can produce both efficient and accurate answers. The results obtained for the different datasets are similar, so it is possible to draw some conclusions that all the test have in common.

1. First, it is possible to see that the models learned with HCPT have a tractable inference complexity, given that the time required to evaluate each query is lower than all the time required for LW methods. The inference time required for quick LW is about three times the inference time required for the PT, and the time inference time required for slow LW is about 30 times the inference time required for the PT.

2. Second, comparing the accuracy levels obtained with the different methods, it is possible to see that the inference error measured by NMLP or MSE is almost always smaller when we use exact inference over PTs than when we use the LW algorithm. The only exception is the results provided by the NMLP using 25 % of the variables of the HEPAR II network as query variables. Given that the NMLP is not defined when the result returned by the inference procedure is 0, when the number of query variables is too large many query samples can be rejected, and therefore the results may be spoiled.

In this experiments, the performance obtained by PTs has matched our expectations. The method works well in time and accuracy, showing that using exact inference over thin models can produce good results, and that the use of incremental compilation in PTs for learning BNs from data achieves a relevant reduction on the inference complexity of the model providing also accurate answers.

Chapter 6

Conclusions and Future Research

This master thesis has the objective of creating a compact model for the representation of network polynomials that we call PTs. The new model is used as a complementary representation for BNs, and its main purpose is to provide a tractable exact inference framework, allowing also an efficient evaluation of the inference complexity of the model.

First, we presented the model properties, and the inference and evaluation procedures for PTs. Then, we presented the methods required for the incremental compilation of PTs, and finally, a method for learning PTs from data. The main motivation for learning PTs was to learn models with a low inference complexity, allowing exact inference in some situations where otherwise (using other learning methods) it would be intractable. The methods presented also provide a flexible framework that allows an easy adaptation of most of the state-of-the-art score+search methods for learning BNs and facilitates the creation of new methods for learning PTs.

From the experimental results presented in this report, we can conclude that:

- The incremental compilation of PTs does not suppose a relevant increment in the time required for the learning process.
- The models learned with HCPT lose a bit of faithfulness but achieve a huge reduction in the inference complexity.
- The time required to perform exact inference in the PTs learned by HCPT is clearly smaller than the time required to perform approximate inference using LW in the BNs learned by score+search methods.

- The answers provided by the PTs learned with HCPT are most of the times more accurate than the answers provided by approximate inference methods in BNs learned by score+search methods.

Although the results obtained in this master thesis are satisfactory, there is a huge potential of improvement and further research work related to PTs. One of the main difficulties of learning ACs was that its complex representation made the task of learning extremely challenging. PTs have a much simpler representation, so the changes in the structure of the model are easier and less expensive. The creation of new learning methods for PTs should be challenging but feasible.

Future research work may focus on improving the incremental compilation and optimization methods presented in this master thesis. These methods were created using heuristics and a further research in the properties of the PTs could help with the creation of methods that obtain more accurate models and more efficiently. Another interesting topic related to learning PTs would be studying the parameter values for the scoring function presented in this master thesis. Here, the parameters were set using the best values obtained in a limited set of tests, so a further analysis is desirable.

Given that PTs encode network polynomials, and their similarities with the AC model, it should be simple to adapt some useful methods used for ACs to PTs. For example, it would be interesting and relatively simple to create methods for obtaining the partial derivatives of the network polynomials encoded in PTs. The partial derivatives are very useful for diverse problems such as obtaining all the marginal probabilities of the polynomial given an evidence or for sensitivity analysis. Another interesting task would be to study the use of PTs for solving abduction problems, in particular the MPE and the MAP problem. The MPE problem could be easily solved by changing the sums for maximizers in the inference procedure *query* presented in Chapter 4.

Finally, it would be very useful to learn PTs in combination with undirected graphical models instead of BNs, so there may be future work related to this topic.

Appendix A

Proof of Theorem 1

The work presented in this master thesis depends heavily on *Theorem 1.* Basically, it assures that the use of incremental compilation of PTs in combination with any *Bayesian network* structure learning algorithm that starts from scratch and makes local changes in the network by the addition, deletion or reversal of directed arcs will produce sound PTs. This is essential, because if a PT is not sound it means that there will be cases where the indicators of the parents of a node in the tree will not be set, and the inference process will fail.

The purpose of this appendix is to prove *Theorem 1*, to assure that all the PTs obtained with the learning methods proposed in this master thesis are sound, and they obtain PTs that are capable of answering any joint probability query that could be asked to its corresponding BN.

A.1 Properties of the Polynomial trees and notation

In order to prove *Theorem 1*, let us first mention the properties of the PTs that are essential for this task.

A PT $\mathcal{P}$ over $\mathcal{X}_P = \{*\} \cup \mathcal{X}$ is a compact representation of a network polynomial corresponding to a BN $\mathcal{B}$ over $\mathcal{X}$. Obviously, $\mathcal{P}$ is a tree, so any node $X_i \in \mathcal{X}$ has exactly one parent in $\mathcal{P}$, with the exception of the root node $*$, that is the ancestor of all the other nodes and has no parents.

All the lemmas presented next are based on Definition 7. It essentially says that $\mathcal{P}$ is sound regarding to $\mathcal{B}$ if $\forall X_i \in \mathcal{X}$ all the parents of X_i in $\mathcal{B}$ are also predecessors of X_i in $\mathcal{P}$. If this property is fulfilled, and given that $\mathcal{P}$ has a tree structure, then any joint probability query that could be answered by $\mathcal{B}$ can also be answered by $\mathcal{P}$.

Let us consider that the changes made to $\mathcal{P}$ and $\mathcal{B}$ during the learning process are produced by the application of the methods *addArc*, *reverseArc*, *deleteArc* or *optimize*. For the first three, it is necessary to check that the result of applying one of these operations to $\mathcal{B}$ and $\mathcal{P}$ will produce a sound PT $\mathcal{P}'$. The method *optimize* changes the representation of $\mathcal{P}$ without changing the probability distribution that it represents, so no changes are applied to $\mathcal{B}$. Therefore, we will need to prove that the PT $\mathcal{P}'$ is sound regarding to the old network $\mathcal{B}$.

The notation used in this appendix is presented next:

- $X_{out} \rightarrow X_{in}$: Arc from X_{out} to X_{in}, where X_{out} is the output node and X_{in} is the input node.
- $\mathcal{B}$: Current BN.
- $\mathcal{B}'$: BN obtained after adding, reversing, or deleting an arc $X_{out} \rightarrow X_{in}$ in $\mathcal{B}$.
- $\mathcal{P}$: Current PT.
- $\mathcal{P}'$: PT obtained after modifying $\mathcal{P}$.
- M: First common predecessor of X_{out} and X_{in} in $\mathcal{P}$. In other words, the deepest node in $\mathcal{P}$ such that $M \in Pred(\mathcal{P}, X_{out})$ and $M \in Pred(\mathcal{P}, X_{in})$.
- $\mathcal{D}_{out}$: $Desc(\mathcal{P}, X_{out})$.
- $\mathcal{D}_{in}$: $Desc(\mathcal{P}, X_{in})$.

Lemma 1. *Let $\mathcal{P}$ be a PT over $\mathcal{X}_P = \{*\} \cup \mathcal{X}$ and $\mathcal{B}$ be a Bayesian network over $\mathcal{X}$. If $\mathcal{P}$ is sound regarding to $\mathcal{B}$, then the PT $\mathcal{P}'$ obtained after applying $addArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$ is also sound regarding to $\mathcal{B}'$, where $\mathcal{B}'$ is the result of adding the arc $X_{out} \rightarrow X_{in}$ to $\mathcal{B}$, and the addition of $X_{out} \rightarrow X_{in}$ to $\mathcal{B}$ does not produce a cycle in $\mathcal{B}'$.*

Proof. According to Definition 7, we know that $\mathcal{P}'$ will be sound regarding to $\mathcal{B}'$ after the addition of $X_{out} \rightarrow X_{in}$ if for any $X_i \in \mathcal{X}$, all the nodes $X_j \in Pa(\mathcal{B}', X_i)$ are predecessors of X_i in $\mathcal{P}'$. Lemma 1 assumes that $\mathcal{P}$ is sound regarding to $\mathcal{B}$, so it will be enough to check that the node X_{in} has X_{out} as one of its predecessors in

$\mathcal{P}'$ and that the nodes involved in the recompilation of the network have not lost any predecessor that were one of their parents in $\mathcal{B}$.

To prove Lemma 1, it is necessary to observe the three different cases that can occur when compiling an arc addition $X_{out} \rightarrow X_{in}$ in $\mathcal{P}$. These cases depend on the position of X_{out} and X_{in} in $\mathcal{P}$.

Case 1 (Scenario 1 of *addArc*): $X_{out} \in Pred(\mathcal{P}, X_{in})$.

In this case no operations are applied to $\mathcal{P}$ by $addArc(X_{out}, X_{in})$, so $\mathcal{P}' = \mathcal{P}$. Therefore, $\mathcal{P}'$ is sound regarding to $\mathcal{B}$. X_{in} is the only node that does not have the same parents in $\mathcal{B}$ and $\mathcal{B}'$. Given that the parents of X_{in} in $\mathcal{P}'$ are $Pa(\mathcal{P}', X_{in}) = Pa(\mathcal{P}, X_{in})$, and $X_{in} \in Pred(\mathcal{P}', X_{out})$, then $\mathcal{P}'$ is sound regarding to $\mathcal{B}'$.

Case 2 (Scenario 2 of *addArc*): $X_{in} \in Pred(\mathcal{P}, X_{out})$.

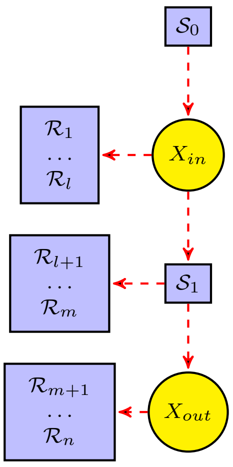

$\mathcal{S}_0$
X_{in}
$\mathcal{R}_1$ ... $\mathcal{R}_l$
$\mathcal{S}_1$
$\mathcal{R}_{l+1}$ ... $\mathcal{R}_m$
X_{out}
$\mathcal{R}_{m+1}$ ... $\mathcal{R}_n$

FIGURE A.1: Case 2 of Lemma 1. Before $addArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$.

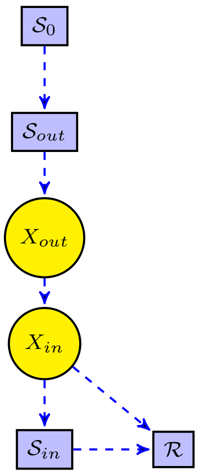


FIGURE A.2: Case 2 of Lemma 1. After $addArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$.

This case is the most complex addition, because it covers a big amount of movements from $\mathcal{P}$ to $\mathcal{P}'$ and part of the tree must be reconfigured. The strategy to prove the soundness of $\mathcal{P}'$ will be to group the nodes and check the soundness of those involved in any change in $\mathcal{P}'$ or $\mathcal{B}'$.

First, lets define some sets of nodes that we will use during the proof:

- Let $\mathcal{S}_0$ be the set of nodes in $\mathcal{X}$, such that $\forall X_i \in \mathcal{S}_0$, $X_i \notin Desc(\mathcal{P}, X_{in})$ *and* $X_i \neq X_{in}$.
- Let $\mathcal{S}_1$ be the set of nodes in $\mathcal{X}$, such that $\forall X_i \in \mathcal{S}_1$, $X_i \in Desc(\mathcal{P}, X_{in})$ *and* $X_i \in Pred(\mathcal{P}, X_{out})$

- Let $\mathcal{S}_{in}$ be the set of nodes $X_i \in \mathcal{S}_1$ such that $X_i \in \mathit{Desc}(\mathcal{B}, X_{in})$.
- Let $\mathcal{S}_{out}$ be the set of nodes $X_i \in \mathcal{S}_1$ such that $X_i \notin \mathit{Desc}(\mathcal{B}, X_{in})$. This set of nodes includes all the nodes in $\mathcal{S}_1$ that also belong to $\mathit{Pred}(\mathcal{B}, X_{out})$, because otherwise the addition of the arc $X_{out} \to X_{in}$ in $\mathcal{B}$ would produce a cycle in $\mathcal{B}'$.
- Let $R = \{\mathcal{R}_1, \mathcal{R}_2, \dots, \mathcal{R}_n\}$ be the branches that hang from the nodes in $\{X_{out}\} \cup \mathcal{S}_1 \cup \{X_{in}\}$ and that do not contain X_{out} or any predecessor of X_{out} in $\mathcal{P}$.

It is trivial that all the nodes in $\mathcal{S}_0$ keep being sound, because all of them have the same predecessors in $\mathcal{P}$ and $\mathcal{P}'$ and they are not affected with the addition of $X_{out} \to X_{in}$ in $\mathcal{B}'$.

After applying $\mathit{addArc}(X_{out}, X_{in})$ all the nodes in $\mathcal{S}_{out}$ keep their deepness order, which means that $\forall X_k, X_l \in \mathcal{S}_{out}$, if X_k was a predecessor of X_l in $\mathcal{P}$ then it is also a predecessor of X_l in $\mathcal{P}'$. The same happens for all the nodes in $\mathcal{S}_{in}$.

For every node $X_i \in \{X_{in}\} \cup \mathcal{S}_{in}$ the predecessors of X_i are $\mathit{Pred}(\mathcal{P}', X_i) = \mathit{Pred}(\mathcal{P}, X_i) \cup \{X_{out}\} \cup \mathcal{S}_{out}$. Given that every X_i keeps in $\mathcal{P}'$ all the predecessors that it had in $\mathcal{P}$, and that $X_{out} \in \mathit{Pred}(\mathcal{P}', X_{in})$, then every node $X_i \in \{X_{in}\} \cup \mathcal{S}_{in}$ is sound in $\mathcal{P}'$.

For every node $X_i \in \{X_{out}\} \cup \mathcal{S}_{out}$ the predecessors of X_i are $\mathit{Pred}(\mathcal{P}', X_i) = \mathit{Pred}(\mathcal{P}, X_i) \setminus (\{X_{in}\} \cup \mathcal{S}_{in})$. Given that no node in $\{X_{in}\} \cup \mathcal{S}_{in}$ is a predecessor of X_{out} in $\mathcal{B}'$ if $\mathcal{B}'$ is a valid network, then every node $X_i \in \{X_{out}\} \cup \mathcal{S}_{out}$ is sound.

Lastly, we have to consider the branches in $\mathcal{R}$. To do that, it is necessary to mention that each branch $\mathcal{R}_i \in \mathcal{R}$ will hang in $\mathcal{P}'$ from the deepest node X_j in $\mathcal{P}$, such that $X_j \in \mathcal{S}_{in} \cup \{X_{in}\}$, and $\forall X_k \in \mathcal{R}_i$, $X_j \in \mathit{Pred}(\mathcal{P}, X_k)$. This means that for every node $X_i \in \mathcal{R}$ the predecessors of X_i in $\mathcal{P}'$ are $\mathit{Pred}(\mathcal{P}', X_i) = \mathit{Pred}(\mathcal{P}, X_i) \cup \{X_{out}\} \cup \mathcal{S}_{out}$. Therefore, all the nodes in $\mathcal{R}$ are sound in $\mathcal{P}'$.

We have checked that all the nodes in $\mathcal{X}$ are sound in $\mathcal{P}'$, and therefore $\mathcal{P}'$ is sound regarding to $\mathcal{B}'$ for this case.

Case 3 (Scenario 3 of *addArc*): $X_{out} \notin Pred(\mathcal{P}, X_{in})$ and $X_{in} \notin Pred(\mathcal{P}, X_{out})$.

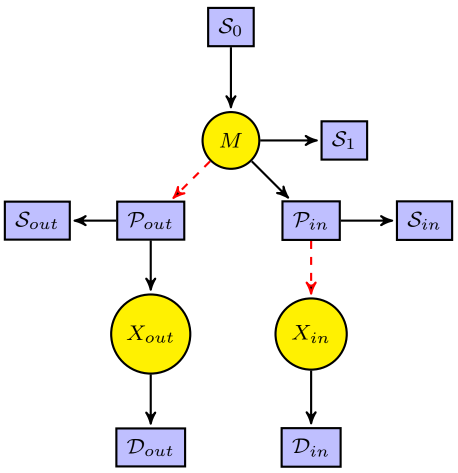


FIGURE A.3: Case 3 of Lemma 1. Before $addArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$.

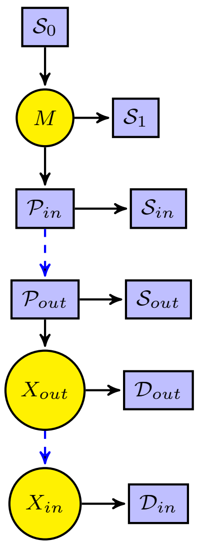


FIGURE A.4: Case 3 of Lemma 1. After $addArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$.

First, lets define some sets of nodes that we will use during the proof of this case:

- Let $\mathcal{P}_{out}$ be the set of nodes in $\mathcal{X}$, such that $\forall X_i \in \mathcal{P}_{out}$, $X_i \in Pred(\mathcal{P}, X_{out})$ *and* $X_i \in Desc(\mathcal{P}, M)$.
- Let $\mathcal{P}_{in}$ be the set of nodes in $\mathcal{X}$, such that $\forall X_i \in \mathcal{P}_{in}$, $X_i \in Pred(\mathcal{P}, X_{in})$ *and* $X_i \in Desc(\mathcal{P}, M)$.
- Let $\mathcal{S}_{out}$ be the set of nodes in $\mathcal{X}$, such that $\forall X_i \in \mathcal{S}_{out}$, $X_i \notin \{X_{out}\} \cup Desc(\mathcal{P}, X_{out})$ and $\exists X_j \in \mathcal{P}_{out} \mid X_i \in Desc(\mathcal{P}, X_j)$.
- Let $\mathcal{S}_{in}$ be the set of nodes in $\mathcal{X}$, such that $\forall X_i \in \mathcal{S}_{in}$, $X_i \notin \{X_{in}\} \cup Desc(\mathcal{P}, X_{in})$ and $\exists X_j \in \mathcal{P}_{in} \mid X_i \in Desc(\mathcal{P}, X_j)$.
- Let $\mathcal{S}_0$ be the set of nodes in $\mathcal{X}$, where $\forall X_i \in \mathcal{S}_0$, $X_i \notin Desc(\mathcal{P}, M)$ *and* $X_i \neq M$.
- Let $\mathcal{S}_1$ be the set of nodes in $\mathcal{X}$, where $\forall X_i \in \mathcal{S}_1$, $X_i \in Desc(\mathcal{P}, M)$ *and* $\forall X_j \in \mathcal{P}_{out} \cup \mathcal{P}_{in}$, $X_i \notin \{X_j\} \cup Desc(\mathcal{P}, X_j)$.

The method $addArc(X_{out}, X_{in})$ removes the connections $M \to M_{out}$ (M_{out} is the child of M that belongs to $\mathcal{P}_{out}$), and $Pa(\mathcal{P}, X_{in}) \to X_{in}$ in $\mathcal{P}'$. Then it adds the arcs $Pa(\mathcal{P}, X_{in}) \to M_{out}$ and $X_{out} \to X_{in}$ in $\mathcal{P}'$. Only node X_{in} has a new parent in $\mathcal{B}'$. The only nodes that could have their predecessors changed in $\mathcal{P}'$ are X_{out},

X_{in}, and the nodes belonging to $\mathcal{P}_{out}$, $\mathcal{S}_{out}$, $\mathcal{D}_{out}$ or $\mathcal{D}_{in}$. The rest of the nodes have exactly the same predecessors in $\mathcal{P}'$ as they had in $\mathcal{P}$, so they are also sound in $\mathcal{P}'$.

For any node $X_i \in \mathcal{P}_{out} \cup \mathcal{S}_{out} \cup \{X_{out}\} \cup \mathcal{D}_{out}$, we have $Pred(\mathcal{P}', X_i) = Pred(\mathcal{P}, X_i) \cup \mathcal{P}_{in}$. Therefore any node X_i has at least the same predecessors in $\mathcal{P}'$ as it had in $\mathcal{P}$, so every X_i is sound in $\mathcal{P}'$.

For any node $X_i \in \{\mathcal{X}_{in}\} \cup \mathcal{D}_{in}$, we have $Pred(\mathcal{P}', X_i) = Pred(\mathcal{P}, X_i) \cup \mathcal{P}_{out} \cup \{X_{out}\}$. Therefore any node X_i has at least the same predecessors in $\mathcal{P}'$ as it had in $\mathcal{P}$, and $X_{out} \in Pred(\mathcal{P}', X_{in})$, so every X_i is sound in $\mathcal{P}'$.

All the nodes in $\mathcal{X}$ are sound in $\mathcal{P}'$ after applying Case 2 of *addArc*, so $\mathcal{P}'$ is also sound.

$\square$

Lemma 2. *Let $\mathcal{P}$ be a PT over $\mathcal{X}_P = \{*\} \cup \mathcal{X}$ and $\mathcal{B}$ be a Bayesian network over $\mathcal{X}$. If $\mathcal{P}$ is sound regarding to $\mathcal{B}$, then the PT $\mathcal{P}'$ obtained after applying $deleteArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$ is also sound regarding to $\mathcal{B}'$, where $\mathcal{B}'$ is the result of removing the arc $X_{out} \to X_{in}$ from $\mathcal{B}$.*

Proof. The method $deleteArc(X_{out}, X_{in})$ only removes X_{out} from $Pa(\mathcal{B}, X_{in})$. Therefore, in $\mathcal{B}'$ X_{out} is no longer a parent of X_{in}. In $\mathcal{P}$ there are no changes applied, so $\mathcal{P}' = \mathcal{P}$. The predecessor of any node $X_i \in X$ in $\mathcal{P}'$ are the same that they were in $\mathcal{P}$. We have then that $Pred(\mathcal{P}', X_i) = Pred(\mathcal{P}, X_i)$. As there are no additional arcs in $\mathcal{B}'$, then $\mathcal{P}'$ is sound regarding to $\mathcal{B}'$. $\square$

Lemma 3. *Let $\mathcal{P}$ be a PT over $\mathcal{X}_P = \{*\} \cup \mathcal{X}$ and $\mathcal{B}$ be a Bayesian network over $\mathcal{X}$. If $\mathcal{P}$ is sound regarding to $\mathcal{B}$, then the PT $\mathcal{P}'$ obtained after applying $reverseArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$ is also sound regarding to $\mathcal{B}'$, where $\mathcal{B}'$ is the result of reversing the arc $X_{out} \to X_{in}$ in $\mathcal{B}$, and $X_{in} \to X_{out}$ does not produce a cycle in $\mathcal{B}'$.*

Proof. The operations performed in $reverseArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$ to $\mathcal{P}$ can be divided into two steps, an arc deletion and Case 3 of an addition. The reversal consists then of the deletion of arc $X_{out} \to X_{in}$ and the addition of the arc $X_{in} \to X_{out}$.

Let us consider that we use an intermediate state to make the transition from $\mathcal{P}$ to $\mathcal{P}'$ and from $\mathcal{B}$ to $\mathcal{B}'$. The PT and the BN will be called $\mathcal{P}_m$ and $\mathcal{B}_m$ respectively in the intermediate state.

1. $\mathcal{B}_m, \mathcal{P}_m \leftarrow deleteArc(\mathcal{B}, \mathcal{P}, X_{out}, X_{in})$.
2. $\mathcal{B}', \mathcal{P}' \leftarrow addArc(\mathcal{B}_m, \mathcal{P}_m, X_{in}, X_{out})$. This operation corresponds to Case 3 of Lemma 1, because X_{out} is a predecessor of X_{in} in $\mathcal{P}_m$.

We can conclude that $\mathcal{P}_m$ is sound regarding to $\mathcal{B}_m$ (Lemma 2), and that $\mathcal{P}'$ is sound regarding to $\mathcal{B}'$ (Lemma 1).

□

Lemma 4. *Let $\mathcal{P}$ be a PT over $\mathcal{X}_P = \{*\} \cup \mathcal{X}$ and $\mathcal{B}$ be a Bayesian network over $\mathcal{X}$. If $\mathcal{P}$ is sound regarding to $\mathcal{B}$, then the PT $\mathcal{P}'$ obtained after applying $optimize(\mathcal{B}, \mathcal{P}, \mathcal{X}_{OPT})$ is also sound.*

Proof. Unlike the algorithms related to the application of local changes in $\mathcal{B}$, the optimization process has the objective of modifying the structure of $\mathcal{P}$, but without changing the probability distribution that it represents, which means that to prove this lemma it is necessary to demonstrate that $\mathcal{P}'$ is sound regarding to $\mathcal{B}$.

As it is shown in Algorithm 4.7, the method $optimize(\mathcal{B}, \mathcal{P}, \mathcal{X}_{OPT})$ consists of a sequence of changes to $\mathcal{P}$. Let $\mathcal{P}_i \in \{\mathcal{P}, \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n, \mathcal{P}'\}$ be each intermediate state from $\mathcal{P}$ to $\mathcal{P}'$. Every PT $\mathcal{P}_i$ is the result of applying the method $pushUpNode(\mathcal{B}, \mathcal{P}_{i-1}, X_{opt})$, where X_{opt} is the variable to optimize in each change. Lets take $\mathcal{P}'_a$ as the result of applying $pushUpNode(\mathcal{B}, \mathcal{P}_a, x)$, where $\mathcal{P}_a$ can be any sound PT in $\{\mathcal{P}, \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$. Proving that any $\mathcal{P}'_a$ is sound regrading to $\mathcal{B}$ would also prove Lemma 4.

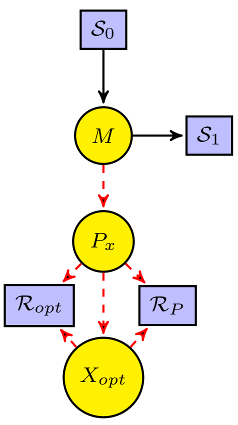


FIGURE A.5: $\mathcal{P}_a$.

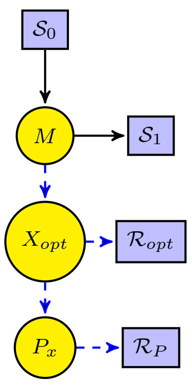


FIGURE A.6: Case 1.

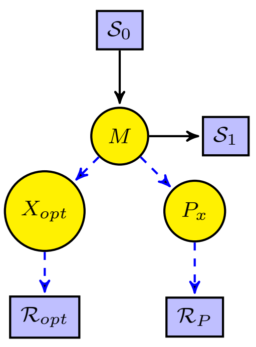


FIGURE A.7: Case 2.

First, lets define a some nodes and set of nodes used during this proof:

- X_{opt}: Node to optimize.
- P_x: Parent of X_{opt} in $\mathcal{P}_a$.
- M: Parent of P_x, that is the first common predecessor of X_{opt} and P_x.
- $\mathcal{S}_0$: Contains all the nodes $X_i \in \mathcal{X}$, such that $X_i \notin Pred(\mathcal{P}_a, M)$ and $X_i \neq M$.
- $\mathcal{S}_1$: Contains all the nodes $X_i \in \mathcal{X}$, such that $X_i \in Pred(\mathcal{P}_a, M)$, $X_i \neq P_x$, and X_i is $X_i \notin Pred(\mathcal{P}_a, P_x)$.
- $\mathcal{R}$: Set of branches hanging from P_x or X_{opt}, such that $\forall X_i \in B$, $X_i \neq X_{opt}$.
- $\mathcal{R}_P$: Set of branches $\{\mathcal{R}_{P_1}, \mathcal{R}_{P_2}, \ldots, \mathcal{R}_{P_k}\} \in \mathcal{R}$, such that $\forall \mathcal{R}_{P_i} \in \mathcal{R}_P$, $\exists X_i \in \mathcal{R}_P$ such that $P_x \in Pred(\mathcal{P}_a, X_i)$.
- $\mathcal{R}_{opt}$: Set of branches $\{\mathcal{R}_{X_1}, \mathcal{R}_{X_2}, \ldots, \mathcal{R}_{X_l}\} \in \mathcal{R}$, such that $\forall \mathcal{R}_{X_i} \in \mathcal{R}_{opt}$, $\mathcal{R}_{X_i} \notin \mathcal{R}_P$.

The strategy now is to prove that all the nodes in $\mathcal{P}'_a$ are sound regarding to $\mathcal{B}$ by checking the changes in their predecessors lists. Let us divide the nodes in $\mathcal{P}_a$ in three groups to prove that every node of $\mathcal{P}'_a$ is sound regarding to $\mathcal{B}$. the sets of nodes will be $\mathcal{S}_M = \{M\} \cup \mathcal{S}_0 \cup \mathcal{S}_1$, $\mathcal{S}_{OPT} = \{X_{opt}\} \cup \mathcal{R}_{opt}$ and $\mathcal{S}_P = \{P_x\} \cup \mathcal{R}_P$. This three sets contain all the nodes in $\mathcal{P}_a$, so $\forall X_i \in \mathcal{P}_a$, $X_i \in \mathcal{S}_M \cup \mathcal{S}_{OPT} \cup \mathcal{S}_P$.

1. $\forall X_i \in \mathcal{S}_M$, $Pred(\mathcal{P}'_a, X_i) = Pred(\mathcal{P}_a, X_i)$, so X_i is sound in $\mathcal{P}'_a$.
2. $\forall X_i \in \mathcal{S}_{out}$, $Pred(\mathcal{P}'_a, X_i) = Pred(\mathcal{P}_a, X_i) \setminus \{P_x\}$. As P_x cannot be a parent of X_i in $\mathcal{B}$ (otherwise there would be a cycle in $\mathcal{B}$), X_i is sound in $\mathcal{P}'_a$.
3. For the set of nodes $\mathcal{S}_P$, there are two possible cases depending on if there is a node in $\mathcal{S}_P$ that is a descendant of X_{opt} in $\mathcal{B}$.

   Case 1: $\exists X_i \in \mathcal{S}_P$ such that X_i is a descendant of X_{opt} in $\mathcal{B}$.

   $\forall X_i \in \mathcal{S}_P$, $Pred(\mathcal{P}'_a, X_i) = Pred(\mathcal{P}_a, X_i) \cup \{X_{opt}\}$, so the node X_i in $\mathcal{P}'_a$ is sound regarding to $\mathcal{B}$.

   Case 2: $\nexists X_i \in \mathcal{S}_P$ such that X_i is a descendant of X_{opt} in $\mathcal{B}$.

   $\forall X_i \in \mathcal{S}_P$, $Pred(\mathcal{P}'_a, X_i) = Pred(\mathcal{P}_a, X_i) \setminus \{X_{opt}\}$. As X_i cannot be a descendant of $\{X_{opt}\}$ in $\mathcal{B}$, the node X_i in $\mathcal{P}'_a$ is sound regarding to $\mathcal{B}$.

We have proved that all the nodes in $\mathcal{P}'_a$ are sound if $\mathcal{P}_a$ was sound, so $\mathcal{P}'_a$ is sound. We can conclude that if $\mathcal{P}$ is sound regarding to $\mathcal{B}$, all the PTs in $\{\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n, \mathcal{P}'\}$ are also sound, so $\mathcal{P}'$ is sound.

□

Theorem 1. *Let Alg be any algorithm for learning in parallel a PT $\mathcal{P}$ and a BN $\mathcal{B}$ from scratch that, to make any change in $\mathcal{P}$ or $\mathcal{B}$ during the learning process, only uses the methods addArc, reverseArc, deleteArc or optimize. A PT $\mathcal{P}'$ is always sound regarding to a BN $\mathcal{B}'$ if $\mathcal{P}'$ and $\mathcal{B}'$ are the output of Alg.*

Proof. By induction.

Base case: When learning a PT and a Bayesian network from scratch, the initial models assume full independence between the variables. Let $\mathcal{B}_0$ be the initial Bayesian network over $\mathcal{X} = \{X_0, X_1, \dots, X_n\}$ and let $\mathcal{P}_0$ be the initial PT over $\mathcal{X} \cup \{*\}$. We have that $\forall X_i \in \mathcal{X}$, $Pa(\mathcal{B}_0, X_i) = \emptyset$, so X_i has no parents in $\mathcal{B}_0$, which means that we can say that all the nodes in $Pa(\mathcal{B}_0, X_i)$, that are none, are also predecessors of X_i in $\mathcal{P}_0$. Therefore, $\mathcal{P}_0$ is sound regarding to $\mathcal{B}_0$.

Induction step: Let us consider that we start from a PT $\mathcal{P}$ and a BN $\mathcal{B}$ such that $\mathcal{P}$ is sound regarding to $\mathcal{B}$. Let $\mathcal{P}'$ and $\mathcal{B}'$ be the resultant PT and Bayesian network obtained after applying a change to $\mathcal{P}$ and $\mathcal{B}$ respectively.

All the changes that could be applied to get $\mathcal{P}'$ and $\mathcal{B}'$ are made by the methods *addArc*, *reverseArc*, *deleteArc* or *optimize*, so $\mathcal{P}'$ and $\mathcal{B}'$ are the result of applying one of these four methods to $\mathcal{P}$ and $\mathcal{B}$.

Case 1: If *addArc* is applied, $\mathcal{P}'$ is sound regarding to $\mathcal{B}'$ by Lemma 1.

Case 2: If *deleteArc* is applied, $\mathcal{P}'$ is sound regarding to $\mathcal{B}'$ by Lemma 2.

Case 3: If *reverseArc* is applied, $\mathcal{P}'$ is sound regarding to $\mathcal{B}'$ by Lemma 3.

Case 4: If *optimize* is applied, there are no changes in $\mathcal{B}$, so $\mathcal{B}' = \mathcal{B}$. Also $\mathcal{P}'$ is sound regarding to $\mathcal{B}$ by $Lemma 4$. So, given that $\mathcal{B}' = \mathcal{B}$, $\mathcal{P}'$ is sound regarding to $\mathcal{B}'$.

We can conclude that in all the possible cases $\mathcal{P}'$ is sound.

□

Bibliography

Akaike, H. (1974). A new look at the statistical model identification. *IEEE Transactions on Automatic Control 19*(6), 716–723.

Bach, F. R. and M. I. Jordan (2001). Thin junction trees. In *Advances in Neural Information Processing Systems*, pp. 569–576.

Beinlich, I. A., H. J. Suermondt, R. M. Chavez, and G. F. Cooper (1989). The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks. In *Proceedings of the 2nd European Conference on Artificial Intelligence in Medicine*, pp. 247–256. Springer-Verlag.

Beygelzimer, A. and I. Rish (2004). Approximability of Probability Distributions. In *Advances in Neural Information Processing Systems*, pp. 377–384.

Bouckaert, R. R. (1993). Probabilistic network construction using the minimum description length principle. In *Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pp. 41–48. Springer.

Carvalho, A. M. (2009). Scoring functions for learning Bayesian networks. *INESC-ID Tec. Rep. 54/2009*.

Chan, H. and A. Darwiche (2001). When do numbers really matter? In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, pp. 65–74. Morgan Kaufmann Publishers Inc.

Chechetka, A. and C. Guestrin (2008). Efficient principled learning of thin junction trees. In *Advances in Neural Information Processing Systems*, pp. 273–280.

Chevrolat, J., F. Rutigliano, and J. Golmard (1994). Mixed Bayesian networks: A mixture of Gaussian distributions. *Methods of Information in Medicine 33*(5), 535–542.

Chickering, D. M. (1996). Learning Bayesian networks is NP-complete. In *Learning from Data*, pp. 121–130. Springer.

Cooper, G. F. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial Intelligence 42*(2), 393–405.

Cooper, G. F. and E. Herskovits (1991). A Bayesian method for constructing Bayesian belief networks from databases. In *Proceedings of the Seventh Conference on Uncertainty in Artificial Intelligence*, pp. 86–94. Morgan Kaufmann Publishers Inc.

Cooper, G. F. and E. Herskovits (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning 9*(4), 309–347.

Dagum, P. and M. Luby (1993). Approximating probabilistic inference in Bayesian belief networks is NP-hard. *Artificial Intelligence 60*(1), 141–153.

Darwiche, A. (2003). A differential approach to inference in Bayesian networks. *Journal of the Association for Computing Machinery 50*(3), 280–305.

Darwiche, A. (2009). *Modeling and Reasoning with Bayesian Networks.* Cambridge University Press.

De Campos, L. M. (2006). A scoring function for learning Bayesian networks based on mutual information and conditional independence tests. *The Journal of Machine Learning Research 7*, 2149–2187.

Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 1–38.

Elidan, G. and S. Gould (2009). Learning bounded treewidth Bayesian networks. In *Advances in Neural Information Processing Systems*, pp. 417–424.

Flores, M. J. (2005). *Bayesian networks inference: Advanced algorithms for triangulation and partial abduction* . Ph. D. thesis.

Fung, R. M. and K.-C. Chang (1989). Weighing and integrating evidence for stochastic simulation in Bayesian networks. In *Uncertainty in Artificial Intelligence*, pp. 209–220.

Gámez, J. A., J. L. Mateo, and J. M. Puerta (2011). Learning Bayesian networks by hill climbing: efficient methods based on progressive restriction of the neighborhood. *Data Mining and Knowledge Discovery 22*(1-2), 106–148.

Gámez, J. A. and J. M. Puerta (2005). Constrained score+ (local) search methods for learning Bayesian networks. In *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, pp. 161–173. Springer.

Geiger, D. and D. Heckerman (1994). Learning Gaussian networks. *Uncertainty in Artificial Intelligence*, 235–243.

Geiger, D., T. Verma, and J. Pearl (1990). Identifying independence in Bayesian networks. *Networks 20*(5), 507–534.

Hastings, W. K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika 57*(1), 97–109.

Heckerman, D., D. Geiger, and D. M. Chickering (1995). Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning 20*(3), 197–243.

Henrion, M. (1988). Propagation of uncertainty by probabilistic logic sampling in Bayesian networks. In *Uncertainty in Artificial Intelligence*, Volume 2, pp. 149–164.

Hrycej, T. (1990). Gibbs sampling in Bayesian networks. *Artificial Intelligence 46*(3), 351–363.

Jaeger, M., J. D. Nielsen, and T. Silander (2006). Learning probabilistic decision graphs. *International Journal of Approximate Reasoning 42*(1), 84–100.

Korb, K. B. and A. E. Nicholson (2003). *Bayesian artificial intelligence.* cRc Press.

Kullback, S. and R. A. Leibler (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, 79–86.

Lam, W. and F. Bacchus (1994). Learning Bayesian belief networks: An approach based on the MDL principle. *Computational intelligence 10*(3), 269–293.

Lauritzen, S. L. and D. J. Spiegelhalter (1988). Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 157–224.

Lowd, D. and P. Domingos (2008). Learning arithmetic circuits. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence.*

Lowd, D. and P. Domingos (2010). Approximate inference by compilation to arithmetic circuits. In *Advances in Neural Information Processing Systems*, pp. 1477–1485.

Lowd, D. and A. Rooshenas (2013). Learning Markov networks with arithmetic circuits. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pp. 406–414.

Mahdi, R. and J. Mezey (2013). Sub-local constraint-based learning of Bayesian networks using a joint dependence criterion. *The Journal of Machine Learning Research 14*(1), 1563–1603.

Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller (1953). Equation of state calculations by fast computing machines. *The Journal of Chemical Physics 21*(6), 1087–1092.

Onisko, A. (2003). *Probabilistic causal models in medicine: Application to diagnosis of liver disorders.* Ph. D. thesis, Institute of Biocybernetics and Biomedical Engineering, Polish Academy of Science, Warsaw.

Park, J. D. (2002). MAP complexity results and approximation methods. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pp. 388–396. Morgan Kaufmann Publishers Inc.

Park, J. D. and A. Darwiche (2001). Approximating MAP using local search. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pp. 403–410. Morgan Kaufmann Publishers Inc.

Park, J. D. and A. Darwiche (2004). A differential semantics for jointree algorithms. *Artificial Intelligence 156*(2), 197–216.

Pearl, J. (1986). Fusion, propagation, and structuring in belief networks. *Artificial Intelligence 29*(3), 241–288.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann.

Pearl, J. (1995). Causal diagrams for empirical research. *Biometrika 82*(4), 669–688.

Schwarz, G. (1978). Estimating the dimension of a model. *The Annals of Statistics 6*(2), 461–464.

Shachter, R. D. and C. R. Kenley (1989). Gaussian influence diagrams. *Management Science 35*(5), 527–550.

Shachter, R. D. and M. A. Peot (1989). Simulation approaches to general probabilistic inference on belief networks. In *Uncertainty in Artificial Intelligence*, pp. 221–234.

Shahaf, D. and C. Guestrin (2009). Learning thin junction trees via graph cuts. In *International Conference on Artificial Intelligence and Statistics*, pp. 113–120.

Spirtes, P., C. N. Glymour, and R. Scheines (2000). *Causation, prediction, and search*, Volume 81. MIT press.

Tsamardinos, I., L. E. Brown, and C. F. Aliferis (2006). The max-min hill-climbing Bayesian network structure learning algorithm. *Machine Learning 65*(1), 31–78.

Vats, D. and R. D. Nowak (2014). A junction tree framework for undirected graphical model selection. *The Journal of Machine Learning Research 15*(1), 147–191.

Wang, T., J. W. Touchman, and G. Xue (2004). Applying two-level simulated annealing on Bayesian structure learning to infer genetic networks. In *Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference*, pp. 647–648. IEEE Computer Society.

Xue, G.-L. (1993). Parallel two-level simulated annealing. In *Proceedings of the 7th international conference on Supercomputing*, pp. 357–366. Association for Computing Machinery.

Yi, W. and Z. Li (2011). Processing of missing values using Gibbs Sampling. In *Proceedings of the Third International Conference on Measuring Technology and Mechatronics Automation-Volume 02*, pp. 927–930. IEEE Computer Society.