Université de Liège

Learning probability densities in a distributed environment

by

Julien Nix

A thesis submitted in partial fulfillment for the degree of master civil engineer in computer science

in the

Faculty of Applied Sciences
Department of Electrical Engineering and Computer Science

Academic year 2014 - 2015

"You do not solve the problems of tomorrow with yesterday's tools."

Henry E. Schmuck

UNIVERSITÉ DE LIÈGE

Résumé

Faculté des sciences appliquées
Département d'Électricité, Électronique et Informatique

Thèse de Master

par Julien Nix

De nos jours, de plus en plus de données transitent à travers le réseau internet. Ces quantités de données faramineuses proviennent de sources multiples: réseaux sociaux, capteurs de données biométriques, streaming video, données scientifiques... Toutes ces données contiennent des informations intéressantes et attrayantes pour plusieurs raisons. En effet, en sciences, l'information permet de comprendre et donc de contribuer à des avancées scientifique. Dans le monde de l'entreprise, que ce soit du marketing, de l'assurance, du monde bancaire ou autre organisation visant à faire du profit, l'information, sur les clients dans ce cas, est d'une nécessité grandissante car elle permet de faire de la publicité ciblée en fonction des goût du client, vérifier son état de santé pour les assurances ou encore de diminuer les risques de prêt en fonction de ses habitudes d'achat.

Évidemment, pour traiter cette masse de données gigantesque, des systèmes particuliers ont été et sont mis en place. Dans cette thèse, le système étudié est un système dit distribué qui, d'une manière générale, vise à diviser pour mieux régner. L'idée est de répartir les données ainsi que les sources de calcul afin de les traiter en même temps. Ce concept est mis en oeuvre grâce à un outils open-source appelé Spark développé à l'université de Berkeley et connaissant un certain succès.

L'autre aspect de cette thèse concerne l'apprentissage automatique de ces données récoltées. Ces deux concepts se marient naturellement, l'un permettant de traiter une foule d'information, l'autre les analysant. Plus précisément, cette thèse se porte sur la caractérisation automatique de densités de probabilité à l'aide de modèles graphiques probabilistes.

L'apprentissage de tels modèles peut s'avérer fort complexe. La stratégie utilisée dans cette thèse, définie dans la thèse de doctorat de Franois Schnitzler, est d'utiliser un mélange d'arbres de Markov. Ces modèles étant de nature plus simple à construire et indépendants, leur création, selon la technique du "Perturb and Combine", dans un milieu distribué est adapté. L'idée est d'utiliser plusieurs arbres de Markov pour encoder des densités de probabilité plus complexes.

Leurs évaluations nous montrent l'utilité d'utiliser un mélange dans le but d'estimer des densités de probabilité. En effet, les résultats montrent de meilleures estimations surtout lorsque beaucoup de variables avec un nombre limité de données est utilisé.

Le développement d'algorithmes dans un milieu distribué ainsi que la mise en place de celui-ci est une tâche complexe. En effet, de nombreuses métriques doivent tre pris en compte. Qu'il s'agisse de hardware ou de software, l'optimisation pour un problème donné n'est pas tâche aisée et demande du temps et de l'expérience.

Les résultats en terme de performance nous montrent que d'une part, il est possible de traiter une quantité importante de données comme annoncé par Spark en distribuant celles-ci et les traitant individuellement. D'autre part, la capacité de Spark à utiliser plus de ressources lorsque le nombre de données disponible augmente tout en gardant de bonnes performances est caractérisée.

De manière général, la programmation distribué est complexe et nécessite de large connaissance en informatique. La création de modèle graphiques probabiliste avec Spark n'est donc pas chose aisée mais est réalisable. Évidemment, les solutions proposées sont toujours assujetties à des améliorations d'autant plus que Spark reste jeune et toujours en évolution.

UNIVERSITY OF LIEGE

Abstract

Faculty of Applied Sciences

Department of Electrical Engineering and Computer Science

Master thesis

by Julien Nix

Nowadays, more and more data are transmitted through the Internet. These huge amounts of data have multiple sources : social networks, biometric sensors, video streaming, sciences research ... All these data contain information interesting and attractive for several reasons. Indeed , in the scientific world, information is the key to understand and thus to contribute to scientific advances. In the corporate world , whether marketing, insurance, banking or other for-profit organizations , information about customers in this case, is a growing need as it allows for targeted advertising based on the customer's taste, check his health condition for insurance or reduce the risk of loan according to their buying habits.

Of course, to deal with this huge mass of data, specific systems have been and are put in place. In this thesis, the studied system is a distributed one which basically uses the concept of "divide and conquer". The idea behind this concept is to distribute the sources of computation and collected data in order to process them simultaneously. This concept is implemented through an open-source tool called Spark developed at the University of Berkeley and knowing some success.

The other aspect of this thesis concerns automatic learning of collected data. These two concepts combine well, one treats a wealth of information, the other analyzes it. Specifically, this thesis deals with the automatic characterization of probability densities using probabilistic graphical models.

Learning such models can be very complex. The strategy used in this thesis, defined in the thesis of François Schnitzler, is to use mixtures of Markov trees. These models are easier to build and can be built independently , with the " Perturb and Combine "

technique , which make them well-suited for a distributed environment. The idea is to use multiple Markov trees to encode more complex probability densities.

Their assessments show the utility of using such mixtures in order to estimate probability densities. Indeed, results show improvements in estimation especially when a lot of variables with a limited number of data is used.

The development of algorithms in a distributed environment as well as its establishment is a complex task. Indeed, many more metrics must be considered,. Whether we are looking at the hardware or software side, optimization for a given problem is not easy and requires time and experience.

Results in terms of performance show that on one hand, it is possible to treat a large amount of data as announced by Spark by distributing them and analyzing them "individually". Moreover, the Spark's ability to use more resources when the number of available data increases and keeping good performance is characterized.

Generally, distributed programming is complex and requires wide computer knowledge. The creation of probabilistic graphical with Spark is not easy but is achievable. Obviously, the proposed solutions are always subject to improvements , keeping in mind that Spark remains young and still evolving too.

Acknowledgements

First and foremost, I would like to express my gratitude to my Professor and supervisor Pierre Geurts for the useful comments, remarks and engagement through the learning process of this Master's thesis.

Furthermore I would like to acknowledge Xavier Tordoir for introducing me to the field of distributed computation as well as the support on the way.

I am also grateful to François Schnitzler for his precious help, wise advice and whose writings have been a wide source of inspiration.

I take this opportunity to express gratitude to my mates for their help and support. The sleepless nights we spent before deadlines and the fun we had will remain in my mermory.

Last but not the least, I would like to thank my family for the incredible support, always listening to my endless stories about my researches and courses.

Contents

List of Figures

List of Algorithms

List of Notations

$\boldsymbol{\mathcal{X}}$	A set of variables.
$D(\mathcal{X}_i)$	Sample of a variable $\mathcal{X}_i$ over a data set D.
n	The number of variables (in a model).
N	The number of observations over a data set D.
$\mathbf{x}_{D_i}$	The set of values in the i^{th} observation in the data set D taken by the set of variables $\mathcal{X}$.
$\mathcal{G}$	A simple graph or a graph representing a probabilistic model.
V	The number of vertices in a graph.
E	The number of edges in a graph.
$\mathcal{T}$	A simple tree graph or a learned tree structure representing a probabilistic model.
$E(\mathcal{G})$	The edges of the graph $\mathcal{G}$ where the number of edge is E.
$V(\mathcal{G})$	The vertices of the graph $\mathcal{G}$ where the number of vertices is V.
$\mathbb{P}$	A probability distribution.
$\mathbb{P}_D(.)$	The empirical probability distribution observed through the data set D.
$H_D(\mathcal{X})$	The Shannon's entropy of $\mathcal{X}$ according to $\mathbb{P}_D(\mathcal{X})$.
$I_D(\mathcal{X};\mathcal{Y})$	The mutual information between $\mathcal{X}$ and $\mathcal{Y}$ according to the $\mathbb{P}_D(\mathcal{X},\mathcal{Y})$.
$D_{KL}(\mathbb{P}_1 \parallel \mathbb{P}_2)$	The Kullback-Leibler divergence of $\mathbb{P}_2$ with respect to $\mathbb{P}_1$.

$Val(\mathcal{X})$ & $Val(\boldsymbol{\mathcal{X}})$	The set of all configurations that a single variable $\mathcal{X}$ or a set of variables $\boldsymbol{\mathcal{X}}$ can take.
$\lvert . \rvert$	The cardinality of a set or tuple.
$\boldsymbol{\mathcal{E}}$	A set of evidence for the inference.
m	The number of Markov trees in a mixture.
λ & μ	The weights of trees in a mixture.
$\mathcal{P}a_{\mathcal{G}}^{\mathcal{X}_i}$	The unique parent of the variable $\mathcal{X}_i \in \mathcal{G}$, can be specified without the $\mathcal{G}$.
θ	The set of parameters of a probability distribution.

Chapter 1

Introduction

The aim of this thesis is to learn discrete probability density estimation in a distributed environment. It will consist in implementing techniques to model probability distributions and to characterize the benefits of using a distributed environment. More precisely, those models will be constructed with the association of "Perturb and Combine" principle and automatic learning of probabilistic models which are used to encode a probability distribution. In this thesis, the models used are the Markov trees, a particular choice which will be motivated in Section 3.1. For an overview of the structure and the different topics introduced in this thesis, Section 1.3 describes their organisation.

1.1 Context & Motivation

In this section, the main research fields presented in this thesis such as machine learning, probabilistic graphical models and distributed computing are replaced into their context in order to have a general overview of their utility.

1.1.1 Machine Learning

Machine learning is a subfield of computer science. It comes from the study of pattern recognition and computational learning theory in artificial intelligence. It exploits empirical data and constructs models through algorithms that can learn from those data and make predictions. Therefore, the predictions depend on the model which depends itself on the data thus the predictions are not fixed, the more empirical data the system have, the more accurate models and predictions it produces.

Some examples of applications include the filtering of spam by the analysis of the e-mails content, facial pattern recognition from facial expression, search engines in order to give accurate answers, diseases from the analysis of genes...

All those tasks go through a massive amount of data, in order to produce models that will fit the "real" cases. The algorithms can exploit those data to answer different kinds of questions. For example, the probability that someone has a genetic disease, the probability that a spam arrives in your mail box, the probability that someone types the words "machine learning" in a search engine, is someone happy or sad from their facial expression... Those are predictive questions where you ask the model to predict what are the probabilities that an event occurs. Other kinds of queries can be asked to the model: what genes are relevant in a certain disease ? Are the genes related to each other or not ? Can we use a subset of elements to fit the data ?

Machine learning tasks are commonly classified into three wide categories in function of the nature of the learning set available. These are:

1. **Supervised learning**. A learning set composed of two sets $\boldsymbol{\mathcal{X}}$, the features or inputs, and $\boldsymbol{\mathcal{C}}$ the desired output, are given as input to the learning algorithm. Its goal is to learn general rules that maps input and their desired outputs. For example, matching people activities in function of their heart pulsation (like classification or regression).

2. **Unsupervised learning**. All given variables have the same status, leaving the learning algorithm on its own to find structure in its input. Unsupervised learning can be a goal in itself by discovering hidden patterns in data (like clustering or density estimation);

3. **In reinforcement learning**. In a dynamic environment, a computer program must perform a certain goal , such as playing chess (or any other game) against an other opponent, without an expert explicitly telling it whether it has come close to its goal or not but only judging on the finality (win or lose the chess game).

There is also a mixed of supervised and unsupervised learning called semi-supervised learning. In this subdivision, an incomplete learning set some missing outputs is given to the learning algorithm.

Machine learning can also be subdivided depending on the desired output of the problems:

1. classification: the inputs are linked to class (discrete values) and the model has to classify unseen inputs;
2. regression: same as classification but the outputs are linear;
3. clustering: grouping sets of variable;
4. **density estimation**: finding the distribution of input variables;
5. dimensionality reduction: simplifying the inputs by doing a projection in a lower-dimensional space.

In this thesis, only density estimation will be studied.

Therefore, machine learning is a research fields linked to a lot of other fields. It can model systems by learning from data and thus is a very suited and powerful way to produce algorithms that evolve by analysing more and more data.

Bias-Variance trade-off The bias-variance trade-off (or dilemma) is the problem of simultaneously minimising two sources of error. The two sources are either the bias and the variance. They limit learning algorithms from generalising beyond their learning set.

- The bias comes from the erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting). A bias learner has the tendency to learn the same wrong thing consistently;
- The variance comes from sensitivity to small random fluctuations in the learning set. High variance can cause overfitting: modelling the random noise in the learning data, rather than the intended outputs.

The bias-variance decomposition is a way of analysing a learning algorithm's expected generalisation error with respect to a particular problem as a sum of three terms, the bias, variance, and a quantity called the irreducible error, resulting from noise in the problem itself.

Section 2.5 will give intuitions about the bias-variance trade-off.

1.1.2 Probabilistic Graphical Models

A graphical model or probabilistic graphical model (PGM) is a probabilistic model for which a graph represents the conditional dependence structure between random variables. They are often used in the probability/statistics theory and in machine learning. They are composed of a compact graph representation and parameters. The graph pictures a set of independencies between the variables in the distribution while the parameters quantify those independencies. Two kinds of graphical representations of distributions which are commonly used exist, i.e. the Bayesian and Markov random fields.

In this thesis, we will focus on the Markov trees but Chapter 3 sums up in more detail the different types of probabilistic graphical models.

In order to understand better why those models are useful, let's take a simple example :

Figure 1.1 is a graph $\mathcal{G}$ composed of 8 labelled nodes. The nodes represent the variables $\mathcal{X}$ (binary in this case) while the arrows represent the dependencies. This structure gives us information about the probabilistic independencies of the problem and the parameters (not represented) quantify the conditional probability distribution of each variable $\mathcal{X}_i$ conditioned on its parents $\boldsymbol{\mathcal{P}}a_{\mathcal{G}}^{\mathcal{X}_i}$ in the graph. Therefore, the joint probability distribution encoded by this Bayesian network is given by a product of conditional distributions:

$$\mathbb{P}_{\mathcal{G}}(\mathcal{X}) = \prod_{i=1}^{n} \mathbb{P}_{\mathcal{G}}(\mathcal{X}_i | \boldsymbol{\mathcal{P}}a_{\mathcal{G}}^{\mathcal{X}_i}) \tag{1.1}$$

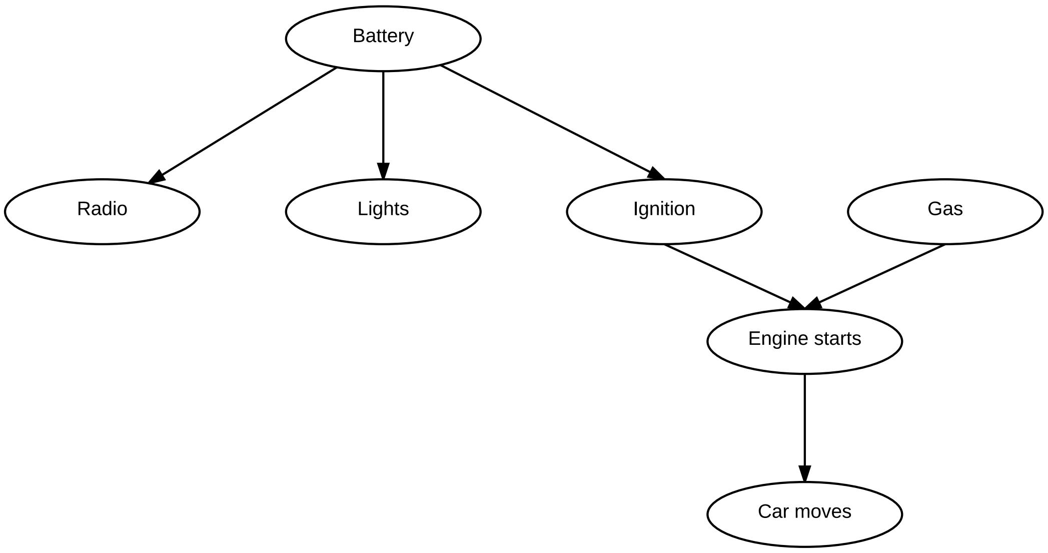


FIGURE 1.1: Bayesian network of a car

Thus, we have here:

$$P(B, R, L, I, G, E, C) = P(B)P(R|B)P(L|B)P(I|B)P(G)P(E|I, G)P(C|E) \qquad (1.2)$$

Moreover, the graph models a car with its different parts and their relationships. We can say that:

1. when the battery is on it may fuel the radio, the lights or/and the ignition process;
2. the engine may start if the ignition process is started and if there is gas left but both are independent;
3. when the engine starts, the car may move

Due to this structure, the graph of this Bayesian network provides even more information between the variables. For example, the "Lights", "Radio" and "Ignition" variables are independent given that the variable "Battery" as the latter is the parent of the other and there is only one path between them. The probability that a car moves when the engine is started is also not impacted by knowing if some gas are left nor if the ignition process is on.

Therefore, we can infer a lot information from those models but why are we using them and not computing a contingency table [1] for the join probability distribution. Because, it would require to compute $2^8 - 1 = 255$ parameters which is much more than our Bayesian network. Indeed, in this example, the Bayesian network only stores 14 independent probability values.

In practice, exploiting structures like Bayesian networks is necessary for a large set of variables and the sum is done quickly. Indeed, if a probability is defined as a 4 bytes float on a computer and assuming that only binary variables are used, a hard drive of 1TB could only store a contingency table of 38 variables. A RAM memory with 1GB could store a table of 28 variables. Therefore it seems obvious that using efficient probabilistic graphical models is needed. Nevertheless, Using the developed techniques with more than 1000 variables is still a challenge.

Furthermore, those PGMs can be constructed automatically or by an ”expert” specifying which variables should be linked together or not. In this thesis, only automatic learning is developed. Learning and inference are developed in more details in Chapter 3.

In this thesis, the available resources are much higher than a simple personal computer. Indeed, using clusters gives us the possibility of modelling distributions with more variables. The experimental data and resources are detailed in Part III.

This gives challenges to apply those techniques on high dimensional data. For example, gene expression measurements (often ten thousand of features) in the biology field, the traded portfolios in the finance field, the analysis of images, ...

1.1.3 Perturb and Combine framework

The aim of machine learning is to create models which will fit at best all possible data sets. The bias-variance trade-off also applies for the learning of probabilistic graphical models. Of course, building a complex model would be very computationally expensive and would not necessarily be a *great* model. The idea in this thesis is to combine multiple simple probabilistic graphical models. The simple models are Markov trees as they are easy to build and easy to infer. Each Markov tree is associated to a weight that describes the importance of the tree. The probability density encoded by the mixture is

[1] a table containing the probability of every joint configuration of the variables

the weighted average of each densities encoded by each weighted tree:

$$\mathbb{P}(\boldsymbol{\mathcal{X}}) = \sum_{i=1}^{m} w_i \mathbb{P}_i(\boldsymbol{\mathcal{X}}) \tag{1.3}$$

where

$$\sum_{i=1}^{m} w_i = 1 \qquad \forall i : w_i \geq 0 \tag{1.4}$$

The combination of simple probabilistic graphical models may give the possibility to represent more elaborate classes of densities. Techniques are known to construct mixture that reduce the variance or the bias. In this thesis, two techniques are implemented, one to reduce the variance using bootstrapping (detailed in Section 4.2.1 and its implementation detail are in Section 18 about the MCL-B algorithm) and one to reduce the bias (detailed in Section 4.2.2 and its implementation details are in Section 19 about the MT-EM algorithm).

1.1.4 Distributed Programming

The distributed programming gives the possibility to compute on cluster larger data set. It also gives access to large amount of CPU, RAM and storage resources. Of course, distributed computing works in a different way than centralized computing, defining its own constraints (coordination, scheduling, ...) In this thesis, the tool used to implement distributed algorithms is called Spark.

Chapter 5 explains it in more detail. Introducing the concept of distributed computing and the Spark tool.

1.2 Goals of the Thesis

The research in this thesis are based on the algorithms defined in the PhD thesis of François Schnitzler [1]. The aim of this thesis is to re-use his research but re-modelled to work in a distributed environment. Different techniques to construct mixtures of Markov trees have been transformed to be efficient and scalable in a distributed environment, using message passing algorithms. Replacing the different techniques in their context, the goal is to characterize them in term of performance (computation time), scalability and accuracy.

1.3 Organization

This thesis is built of four main parts. The first one is the background where the general topics are approached in a more theoretical way. It introduces the notions of machine learning, probabilistic graphical models (PGMs), mixture models and Spark, the tool used to perform distributed computation. The second part describes and explains the different algorithms that have been implemented to construct mixtures of Markov trees. The third part deals with the different experimental results of the algorithms described in the second part. Finally, the fourth part is a discussion containing the possible improvements and a conclusion.

1.3.1 Part I: Background

This part presents the basic notions of the thesis. Chapter 2 gives simply notions of machine learning, for what it is used and how it is related to probabilistic graphical models. Chapter 3 introduces the probabilistic graphical models in a general way and define the model used in this thesis. It also defines how to build it, perform inference on it and the complexity of the presented algorithms. Then, Chapter 4 explains what is a mixture of trees and defines its framework. Finally, Chapter 5 is about distributed computing and the tool used, i.e. Spark and its particularities.

1.3.2 Part II: Distributed Algorithms

This part is all about the distributed algorithms that have been implemented thanks to Spark. Chapter 6 focuses on the distributed techniques to learn Markov trees as explained in the previous part. Then, Chapter 7 focuses on learning a mixture of trees. The algorithms are explained, sometimes with an illustration, and a pseudo-code is given as a support to understand them. Each of them are also evaluated in terms of time complexity.

1.3.3 Part III: Experimental Results

This part examines the experimental results of the correctness of mixtures and results of the previous algorithms in terms of computation time and scalability.

1.3.4 Part IV: Discussion

This part is composed of two chapters, the first one, Chapter 10, is about the possible improvements, what should be done, re-done or optimized. The second one 11 is the conclusion of this thesis highlighting the main resulting concepts of this thesis and discussing about the future of computing.

1.3.5 Appendix

The appendix provides some information about the source of the data used to perform the experiments. The representation is also discussed as, even if it seems a less important and simple subject, choosing one representation or another may lead to bad performance and misconception of how to build algorithms from it. My personal experience and general information about the distributed environment are also discussed.

Part I

Background theory

Chapter 2

Machine Learning Field

This chapter is a reminder about the goal of the machine learning; introducing some concepts and also what probabilistic graphical models are used for. The needed notions of machine learning are first introduced. Then, techniques to validate a model and related error between the input of a model and its output are discussed.

2.1 Brief Presentation

The core of a learning problem is to consider a set of n samples of a data set D and then try to predict properties of unknown data. A sample can be more than a single number and, for instance, a multi-dimensional entry (i.e. multivariate data), those are said to have several attributes or features. As there are many similarities between machine learning theory and statistical inference, although they use different terms, the machine learning field is sometimes called statistical learning or automatic learning.

2.2 From the data to the model

At first, in order to learn phenomenons, a data set D containing gathered observations of the studied variables has to be set up. Indeed, machine learning is about learning some properties of a data set and applying them to new data. Therefore, a common practice to evaluate an algorithm is to split the data set D at hand into two sets, one that is named the **learning set** on which we learn data properties and one that we call the **testing set** on which we test these properties.

More formally, a data set D is a sequence of N observations:

$$D = \mathbf{x}_{D_1}, ..., \mathbf{x}_{D_N}, \tag{2.1}$$

where an observation corresponds to a collection of measured variables:

$$\mathbf{x}_D \in Val(\boldsymbol{\mathcal{X}}) = \mathbf{x} = (\mathcal{X}_1 = x_1, ..., \mathcal{X}_n = x_n) \sim \mathbb{P}(\boldsymbol{\mathcal{X}}), \tag{2.2}$$

and, of course, where the properties variables are observed through a set of n random variables:

$$\boldsymbol{\mathcal{X}} = \mathcal{X}_1, ..., \mathcal{X}_n, \tag{2.3}$$

whose joint probability density is defined by $\mathbb{P}(\boldsymbol{\mathcal{X}})$. Note that there is no restriction on those variables. They can be continuous or discrete, categorical or numerical.

As said in the followed PhD. thesis, a core hypothesis is that the observations are all generated from the same density and this density do not change over time. Moreover, the samples are considered independent from each other. Remind that, such observations are denoted *iid* which stands for independent and identically distributed. Thus, the probability of a data set D is:

$$\mathbb{P}(D) = \prod_{i=1}^{N} \mathbb{P}(\boldsymbol{\mathcal{X}} = \mathbf{x}_{D_i}) \tag{2.4}$$

Once the learning and the test sets are defined, the learning set should be used as input of the learning algorithm. From those data contained in the learning set, the learning algorithm will produce what is called a model. In our case, probabilistic graphical model.

2.3 Subdivisions of Machine Learning tasks

As explained in the introduction, the machine learning tasks are subdivided in three main subdivisions. In this thesis, only unsupervised learning to perform density estimation and probabilistic inference is discussed.

2.3.1 Probabilistic Graphical Models in Density Estimation

In probability and statistics, density estimation is defined as the construction of an estimate $M_D = \widehat{\mathbb{P}}(\boldsymbol{\mathcal{X}})$ which is established from a data set D whose data are from an unobservable underlying probability density function that have generated the samples. This is exactly what **probabilistic graphical models** could be used to. They learn the structure of a density distribution from a learning set. The structure, constructed from the relationship between variables, can reduce the number of parameters used to characterize a density.

Just to give a slight vision of what is density estimation, remember this simple concept, and most people are already familiar with one common density estimation technique: the histogram. However, histograms are dependent on the choice of the "bins" in which the data are gathered. This can have a disproportionate effect on the resulting visualization and might lead to wrong interpretations of the data. Other techniques are used to estimate density, like uniform, Gaussian or Epanetchikov kernel. Figure 2.1 illustrates the idea of density estimation.

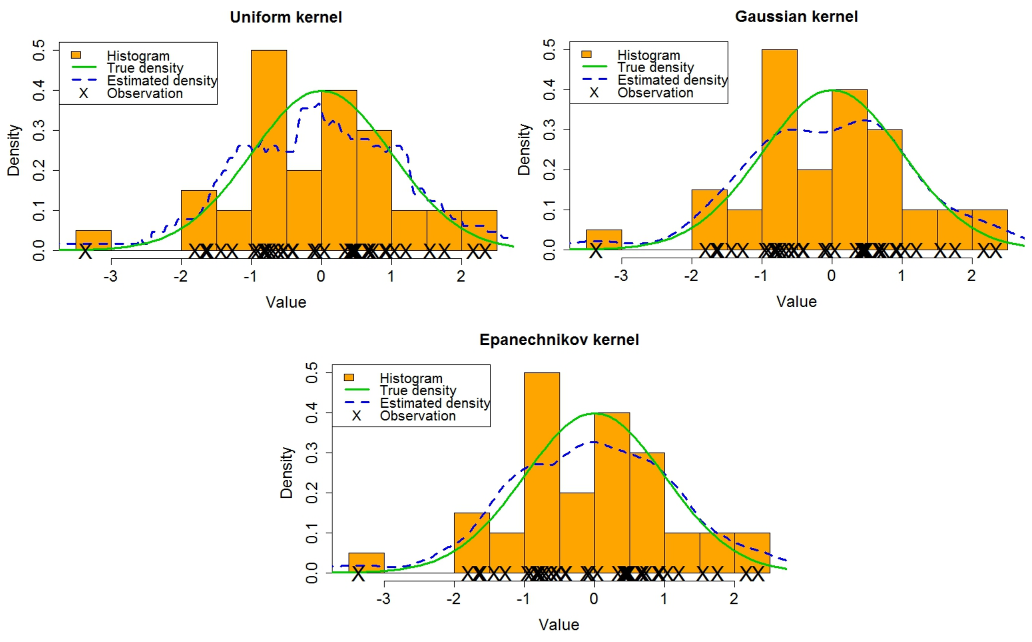


FIGURE 2.1: Estimation of a density through a histogram and a uniform, Gaussian and Epanetchikov kernel in function of the observation

As explained before, models can be used to perform probability inference. Various queries about the original probability density represented by the model can be done, here are some:

- the likelihood of observing a problem configuration $\mathbf{x}$;
- a conditional density $\widehat{\mathbb{P}}(\boldsymbol{\mathcal{C}}|\boldsymbol{\mathcal{X}}')$ which is exploited to predict the values of the output $\mathcal{C}$ in function of the input $\boldsymbol{\mathcal{X}}'$;
- the most likely configuration of an observation, $arg\ \underset{\mathbf{x}}{max}\ \widehat{\mathbb{P}}(\mathbf{x})$;
- the generation of new observations for simulations.

Those queries can be even used in parallel of other machine learning techniques, giving a way to generate observations and test on those synthetic data or even probabilistic information about the observations that have been made (greatly or not).

Furthermore, it should be kept in mind that any estimation can be associated to the error made from what is supposed to estimate. In statistical estimation, two kind of errors can be considered. The first kind is the error made at the parameter level. Indeed, to compare estimated parameters to true parameters, this estimation will be subjected to errors. The second kind is in the comparison of density itself. Therefore, in order to measure the correctness of the constructed models, we have to validate them. Validation will be described in the next section.

2.4 Validation

Creating models is one thing but assessing them is another one. One very important facet of the machine learning is the validation of the solutions provided by those models. Without it, it is impossible to verify if what you consider as an answer of a problem is really relevant. Note that the interpretability is also important, like the complexity of the techniques. The first topic is not discussed while the latter is approached in the different corresponding sections.

A distinction must be made between the validation of an algorithm and of a model on a specific problem. In this thesis, only the validation of models is discussed. As it is related to experimentation, the validation is approached in Part III and not defined in a specific and theoretical way. Just keep in mind that validation requires a learning set and a testing set[1]. From those sets, we can define the learning set error and the testing set error. Roughly speaking, the first error corresponds to the difference between the learning set output and their predictions (probability density in our case). The second error corresponds to the difference between the testing set output and their predictions.

In this thesis, the quality of a model is evaluated by its error, the likelihood ratio of a given variable according to the true density and the one encoded by the model. The inadequacy between two densities can be measured by the Kullback-Leibler divergence. Section 8 provides more details about it.

2.5 Bias and Variance

The aim of this section is just to give intuitions about what is the error called bias and the one called variance.

An analogy can be with the throwing darts games illustrated in Figure 2.2. Four different players throw darts and each are associated with a low or high bias and a low or high variance. The players can be seen as the model represented by the blue dots and the center is the target distribution represented by the red dots.

A model with a high bias will tend to always deviate from the right distribution. Therefore, it can be seen that the blue dots representing the model are shifted above the center representing the target distribution. It is like players with high bias are aiming at another point than the center.

The throwers with high variance have their darts more spread out. They are less able to successfully place the dart where they are aiming (the center in this case). In this case, the model does not fit exactly the target distribution in a different way.

In general, there is a trade-off between bias and variance meaning that if the bias is lowered by a particular technique, it variance will increase and vice versa. Thus models

[1]A validation set is also used in the cross-validation, see Section 2.5.1.

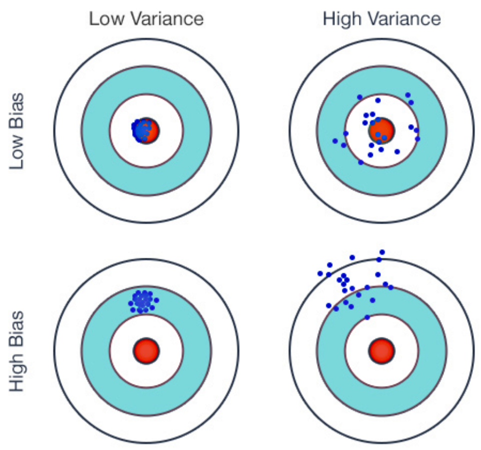


FIGURE 2.2: Bias-Variance trade-off analogy

with high bias can achieve lower variance, and models that have low bias will be at the expense of higher variance.

2.5.1 Underfitting, overfitting

As described before, both high bias and high variance can be considered as ”deviation” properties. Remind that a model is constructed from a learning set and depends on it. Typically, a model with high bias is said to *underfit* the data while a model with high variance and low bias is said to *overfit* the data.

Here is a simple regression example; assume that the aim is to represent a function $f(x)$ as a polynomial of a certain degree. By choosing the degree of the polynomial, a model can be defined to fit the function. Thus we have this approximation:

$$f(x) = a_0 + a_1 x + a_2 x^2 + ... + a_d x^d \tag{2.5}$$

The higher the degree is, the more complicated the function becomes. A first degree polynomial is a line, second degree one is a parabola, etc. In our case, three models are constructed to fit the data set: a simple line, a fourth degree polynomial, and a sixth degree polynomial. Figure 2.3 illustrates the problem.

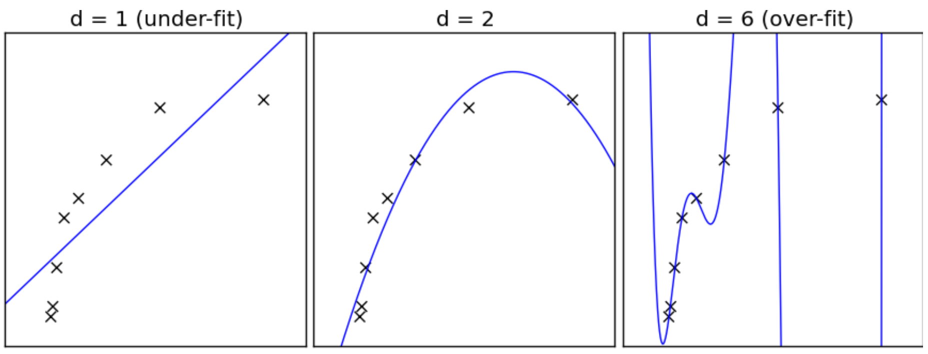


Figure 2.3: Approximation of a data set with polynomials of different degrees

The right one fits the data set perfectly and is not biased. However, we can see that the behavior of the curve is erratic, and seems different from the general pattern in the data. It has worked too hard to fit every single point, and has *overfit* the data. Thus, predictions from this model are likely to be inaccurate. Thus, we can say that this model has a low bias and a high variance. Note that if one point of the data is slightly moved, the curve would change a lot to fit all the point. Therefore, this type model has low bias but high variance.

We can say that matching exactly the learning set is not always wanted. Models should make good predictions on the learning set (as they have been build from it) but should also make good predictions on unseen, totally new, data. This is named the *generalization* of a model. Thus a more powerful model for a particular learning set is not necessary the best model.

Unlike the right polynomial, the left one has a high bias. This model will always be a line trying to fit the data as it can be with a line. We can say that this model *underfits* the data. Note that if one point of the data is slightly moved, the line model will not change a lot. Therefore, this high biased linear model has has low variance.

In this case, the center plot represents a pretty good fit as it does not over/underfit too much the data. Note that in machine learning, problems overfitting is more present (fitting the noise for example). Overfitting also applied for probabilistic graphical models to estimate probability densities.

Overfitting Detection and Cross-validation

In order to understand the problem of models' representation, some clues are given.

Multiple techniques exist to reduce overfitting like the bootstrap aggregation technique as explained in the section 4.2.1. A simple and effective technique to detect overfitting is called *cross validation*.

It involves to use a third additional set of data named the validation set. This new set can be constructed from the learning set by randomly splitting it into two subsets:

1. a smaller learning set (typically 75%) and,
2. a validation set (25%).

Then, the model is constructed over the new smaller learning set. The validation set is used to see if model generalizes well to unseen data as, after all, the aim is to predict future, unknown density probability.

Therefore, instead of just looking at the learning error, both learning and validation error are examined. From those errors, it is possible to see if a model is overfitting or underfitting. This is illustrated in Figure 2.4. Note that the model complexity corresponds to its adaptivity, its capacity to fit the learning set. This depends on the model of course. In the case of density estimation, the complexity increases with the number of independent parameters used to represent a model.

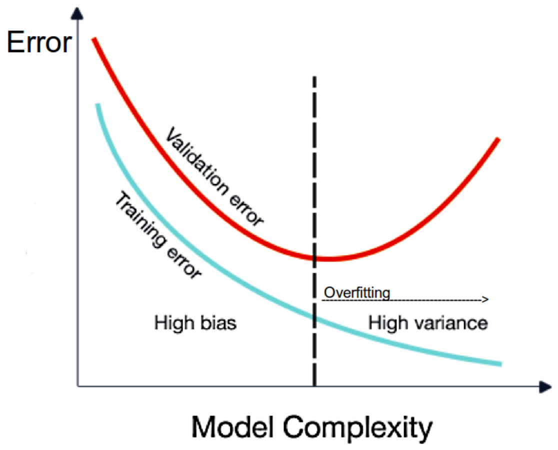


FIGURE 2.4: Representation of overfitting. Usually, the error of a model is a decreasing function of the model complexity when measured on the learning set.

As the techniques have random parts in it, the analysis should be performed multiple times to see the variance in the learning and validation errors.

The principle of using a validation test to evaluate the correctness of model is used for probabilistic models. The experiments done in Chapter 8 use this principle, using a learning set, a test set and a validation set.

The next chapter will focus on the core model of the thesis: probabilistic graphical models and more precisely Markov trees.

Chapter 3

Probabilistic Graphical Models

Probabilistic Graphical Models have been cited but what are they, why are they used and what do they "look like"? This chapter provides the main characteristic of them and shows different types of PGMs. Of course, the main focus of this thesis are the Markov trees. In order to distinguish those trees to other PGMs, we need to define them and express their particularities. This chapter also expresses the way to learn Markov trees and under which hypotheses it is performed. The algorithms are explained in a simple way, illustrated by an example; their complexity is also described and pseudo-codes are used as a simple support to present a way to implement those algorithms.

3.1 General Introduction

The probabilistic graphical models associate a graph representation and probabilities to encode joint probability distributions. Their representation is compact, can be interpreted easily and are less expensive to compute than computing a contingency table which could be intractable. More formally, they are composed of a structured graph $\mathcal{G}$ and its related parameters θ. The graph consists of a set of V vertices representing the studied variables $\boldsymbol{\mathcal{X}}$ which are linked by E edges representing the dependencies. Thus, on one hand, $\mathcal{G}$ encodes the independencies and the parameters θ quantify them. Note that a lot of different PGMs exist[1], they are classed in function of their properties; if

[1]Clique tree, chain graphs, conditional random field, etc.

they are directed, undirected, chained, cyclic, acyclic, etc. They can be differentiated also by the capacity of probabilistic dependencies that they can encode.

Figure 3.1 illustrates some of the main probabilistic graphical models related to this thesis. This figure gives a good overview of the relations between some constituent PGMs of Markov trees. As it is shown, the Markov trees are the intersection of the polytrees and Markov Forests, which are themselves included in other types of models. Therefore, to understand Markov trees, its constituents have to be presented. The next sections will explain their graphical differences and the related encoded independencies.

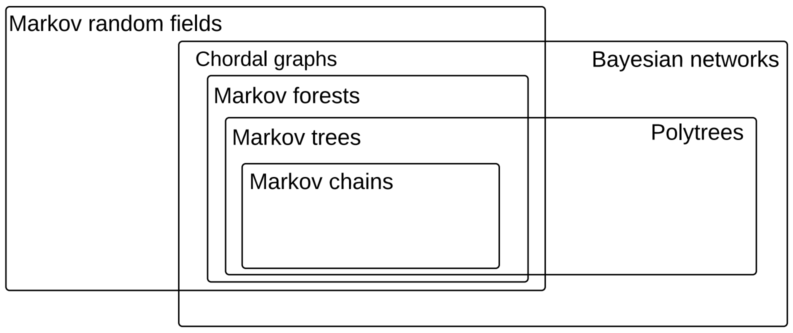
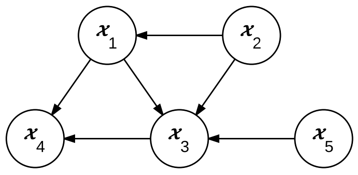


FIGURE 3.1: Relations between different classes of PGMs. The figure can be read as follows : Markov chains ⊂ Markov trees; Markov tree = (Markov forest ∩ Polytrees); Polytrees ⊂ Bayesian networks; Markov forests ⊂ Chordal graphs; Chordal graphs = (Markov random fields ∩ Bayesian networks).

3.1.1 Bayesian Networks



Representation, Properties and Interpretation As shown in the figure above, Bayesian networks (BNs) are directed acyclic graphs. As explained before, they encode joint probability distribution as the product of the marginal distribution of each variables

$\mathcal{X}_i$ conditionally to its parents $\boldsymbol{Pa}_{\mathcal{G}}^{\mathcal{X}_i}$ where the parameter θ quantifies it:

$$\mathbb{P}_{\mathcal{G}}(\mathcal{X}) = \prod_{i=1}^{n} \mathbb{P}_{\mathcal{G}}(\mathcal{X}_i | \boldsymbol{\mathcal{P}}a_{\mathcal{G}}^{\mathcal{X}_i}, \theta) \tag{3.1}$$

Now that the representation of a BN and its properties are described, the way to interpret it (i.e. inferring the independencies between variables) are given.

Consider three subsets $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}, \boldsymbol{\mathcal{C}}$ of the vertices in the graph $\mathcal{G}$. $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are said to be *d-separated* by $\boldsymbol{\mathcal{C}}$ (noted $(\boldsymbol{\mathcal{A}}|\boldsymbol{\mathcal{C}}|\boldsymbol{\mathcal{B}})$): the d-separation expresses the conditional independence between $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ conditionally to $\boldsymbol{\mathcal{C}}$. More formally, in a directed graph:

$$\forall\, \boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}, \boldsymbol{\mathcal{C}} \subset \text{disjointed Vertices}, (\boldsymbol{\mathcal{A}}|\boldsymbol{\mathcal{C}}|\boldsymbol{\mathcal{B}}) \Rightarrow \boldsymbol{\mathcal{A}} \perp \boldsymbol{\mathcal{C}} \,\big|\, \boldsymbol{\mathcal{B}}$$

Now, the d-separation should be described: the subsets $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are *d-separated* if and only if every chain of nodes from $\boldsymbol{\mathcal{A}}$ to $\boldsymbol{\mathcal{B}}$ is "blocked" by $\boldsymbol{\mathcal{C}}$. Consider three nodes of those subsets; $\mathcal{A} \in \boldsymbol{\mathcal{A}}, \mathcal{B} \in \boldsymbol{\mathcal{B}}$ and $\mathcal{C} \in \boldsymbol{\mathcal{C}}$; for these three node, a chain is blocked in the two following cases:

- the chain contains a sequence $\mathcal{A} \to \mathcal{C} \to \mathcal{B}$ or $\mathcal{A} \leftarrow \mathcal{C} \to \mathcal{B}$ or $\mathcal{A} \leftarrow \mathcal{C} \leftarrow \mathcal{B}$;
- the chain contains a v-structure sequence $\mathcal{A} \to \mathcal{Z} \leftarrow \mathcal{B}$ where $\mathcal{Z} \notin \boldsymbol{\mathcal{C}}$ and no descendant of $\mathcal{Z} \in \boldsymbol{\mathcal{C}}$.

The intuition is that $\mathcal{C}$ blocks the information between $\mathcal{A}$ and $\mathcal{B}$.

Formally, the conditional independence expressed above can be define like this:

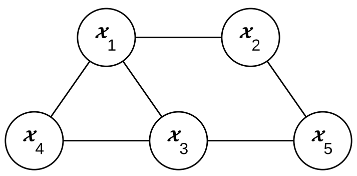
$$\mathcal{A} \perp \mathcal{B} \,\big|\, \mathcal{C} \iff \mathbb{P}(\mathcal{A} \,\big|\, \mathcal{B}, \mathcal{C}) = \mathbb{P}(\mathcal{A} \,\big|\, \mathcal{C}) \; \textit{and} \; \mathbb{P}(\mathcal{B} \,\big|\, \mathcal{A}, \mathcal{C}) = \mathbb{P}(\mathcal{B} \,\big|\, \mathcal{C})$$

Or

$$\mathcal{A} \perp \mathcal{B} \,\big|\, \mathcal{C} \iff \mathbb{P}(\mathcal{A}, \mathcal{B} \,\big|\, \mathcal{C}) = \mathbb{P}(\mathcal{A} \,\big|\, \mathcal{C}) \times \mathbb{P}(\mathcal{B} \,\big|\, \mathcal{B})$$

Hence, the way to interpret independencies in a BN and also how it encodes probabilities is given. This is the key to interpret BNs and also Markov trees.

3.1.2 Markov Random Fields

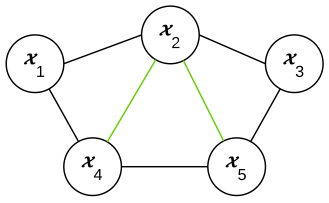


Representation, Properties and Interpretation As shown in the figure above, Markov random fields are undirected graphs. Of course, Markov random fields can encode sets of independence relationships while Bayesian networks cannot. Identically, the independencies can be read in the graph. In this case, edges are not oriented thus the sequence where v-structure appears is not possible. Therefore, the independencies (which are deduced from the global Markov properties in fact) are defined like this:

Consider three subsets $\boldsymbol{\mathcal{A}}, \boldsymbol{\mathcal{B}}, \boldsymbol{\mathcal{C}}$ of the vertices in the graph $\mathcal{G}$. $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are independent to $\boldsymbol{\mathcal{C}}$ if every path from any nodes $\mathcal{A} \in \boldsymbol{\mathcal{A}}$ to any nodes $\mathcal{B} \in \boldsymbol{\mathcal{B}}$ is blocked by at least one node $\mathcal{C} \in \boldsymbol{\mathcal{C}}$.

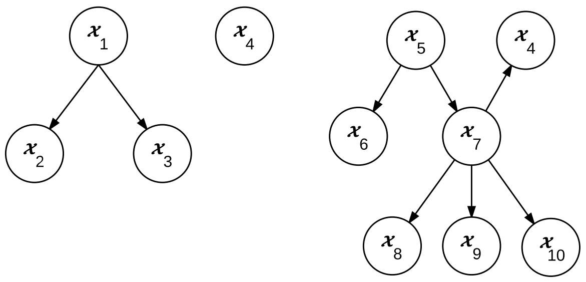
Note that as MRFs are not oriented (and thus have no v-structure), they cannot encode the fact that two variables $\mathcal{A}$ and $\mathcal{B}$ can be independent (i.e. $\mathcal{A} \perp \mathcal{B}$) and dependent conditionally to a third variable $\mathcal{C}$ (i.e. $\mathcal{A} \perp \mathcal{B}|\mathcal{C}$).

3.1.3 Chordal Graphs



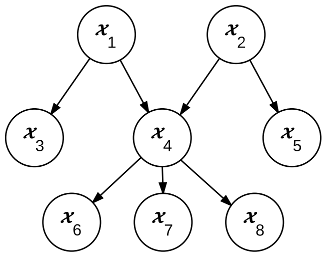
Representation, Properties and Interpretation As shown in the figure above, chordal graphs are graphs in which all cycles of four or more vertices have a chord. A chord is an edge which connects two vertices of the cycle but is not contained in it. Similarly, every cycle in $\mathcal{G}$ should be made up of, at most, three vertices. Chordal graphs are the intersection of Bayesian networks and MRFs; both can be used to represent chordal graphs and thus the set of independence relationships they should encode. Of course, in order to represent a chordal graph by a Bayesian network, edges have to be simply oriented but without creating any v-structure (would change the structural independencies) or directed cycle (would not be a Bayesian network). Chordal graphs can be interpreted like MRFs or like the corresponding Bayesian network. They can encode all distributions as complete graphs [2] are also chordal.

3.1.4 Markov Forests



Representation, Properties and Interpretation As shown in the figure above, Markov forests is a subclass of the chordal graphs. From the point of view of the chordal graphs, Markov forests are acyclic, disconnected graphs which can only contain $V - 1$ edges. It can also be seen as a Bayesian network whose variables have only one parent. Interpretation is also done identically. Moreover, as those models are more restrictive, they cannot model all distributions.

3.1.5 Polytrees



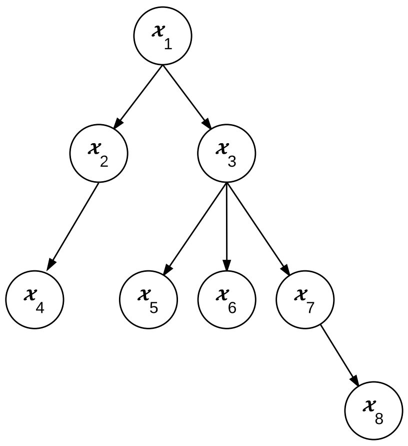
Representation, Properties and Interpretation As shown in the figure above, polytrees are directed acyclic graphs whose skeleton[3] is a tree. In other words, if the

[2]A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.

[3]The skeleton of a graph is its underlying undirected graph.

directed edges are replaced by undirected ones, the resulting graph is an undirected graph that is both connected and acyclic. It is a more constrained model than Bayesian network as it proscribes directed cycles. From those properties, we can also deduce that a polytree with V vertices (i.e. n variables) has $V-1$ edges. Independencies are interpreted the same way as in Bayesian networks. Of course, this more restrictive model can not encode any possible distribution.

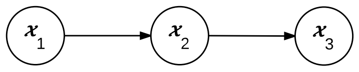
3.1.6 Markov Trees



Representation, Properties and Interpretation As shown in the figure above, Markov trees are acyclic, connected graphs spanning all variables that have one and only one parent. Markov tree are defined as the intersection of polytrees and Markov forests. Therefore, it contains $V-1$ edges. They can be transformed from a MRF into a Bayesian network by choosing randomly a root and orienting the edges outwards the root (used and explained in Section 6.1.3).

Markov tree can be interpreted, in terms of independencies, like in a Bayesian network. Markov trees are easy to interpret as they are more constrained (limiting it in term of distributions that can be encoded). In Markov trees, like in the figure above, parents are independent of the descendants of their children conditionally to their children and vice versa. Mixtures in this thesis will use this model, it cannot encode all possible distributions but is relatively easier to be built than a real Bayesian network for example.

3.1.7 Markov Chains



Representation and Properties As shown in the figure above, Markov chains are a subclass of Markov trees where each variable is connected to at most two variables.

Figure 3.2 is an illustration of the different PGMs and their properties.

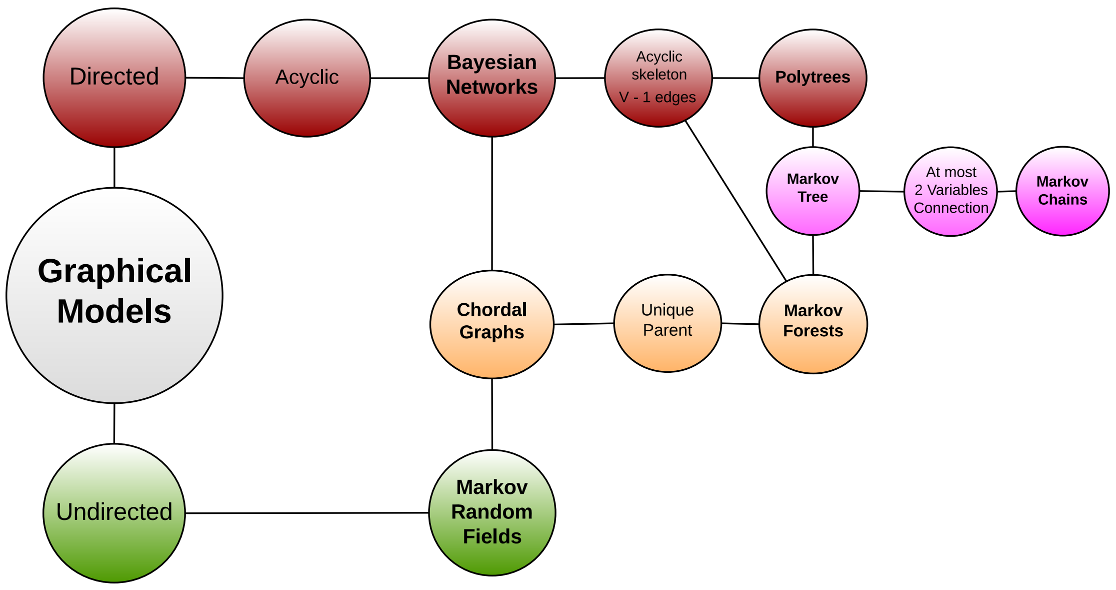


FIGURE 3.2: Summary of the PGMs and their properties

As said earlier, PGMs are used to answer probabilistic queries about the studied variable. Probabilistic inference in Markov trees is discussed in next Section 3.2.

3.2 Inference in Markov Trees

Once a PGM has been specified or constructed, it can be exploited to answer probability queries about the distribution. Indeed, inference is used to compute any marginal probabilities in PGMs and, in our case, in Markov tree which can be done in an efficient way.

The algorithm commonly used to perform inference is the belief propagation which is a message passing algorithm (also called sum-product message passing). This algorithm was first proposed by Judea Pearl in 1982, who formulated this algorithm on trees, and was later extended to polytrees. While it performs exact inference on tree, it has since been shown to be a useful approximate algorithm on general graphs.

Remember that a joint distribution encoded by Markov trees can be factorized as the product of the marginal of each variable $\mathcal{X}_i$ conditionally to its parent. Consider also that an observation (or multiple), commonly called *evidence*, has been made on one of the studied variables and you want to infer the impact of this evidence on the other variables. Thus, the idea is to compute inferences of the form $\mathbb{P}_{\mathcal{G}}(\boldsymbol{\mathcal{X}}|\boldsymbol{\mathcal{E}} = \mathbf{e})$ where $\boldsymbol{\mathcal{E}}$ is a set of evidence and of course, $\boldsymbol{\mathcal{X}}$ is the set of the other studied variables.

Therefore, computing the marginals $P(\mathcal{X}_i|\mathcal{E}_j = e)$ where the evidence is $\{\mathcal{E}_j = e\}$ would be very computationally expensive if we had to re-compute all the marginals. Fortunately, using the structure of the Markov tree and by sending messages from nodes to nodes, the belief propagation algorithm allows the marginals to be computed much more efficiently.

3.2.1 Description of the Belief Propagation

We will first describe the initialization of the belief propagation [2] with some illustrations. First, Figure 3.3 gives a quick way to visualize the messages passing construction. A node $\mathcal{X}$ has a unique parent P and a set of ancestors $\boldsymbol{\mathcal{A}}_{\mathcal{X}}$ (i.e. parents of this parent). Of course, it has a set of Children C and also a set of descendant $\boldsymbol{\mathcal{D}}_{\mathcal{X}}$ (i.e. children of the children). In the belief propagation, we will go through only the parent and the children level per level.

Despite the general representation of a node, let's note that there are three types of nodes:

1. the root which has no parent;
2. the intern nodes which have an unique parent and one or multiple children;

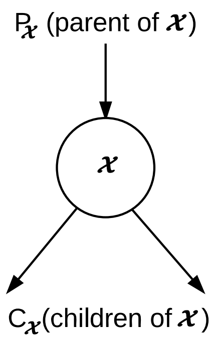

$P_{\mathcal{X}}$ (parent of $\mathcal{X}$)
$\mathcal{X}$
$C_{\mathcal{X}}$(children of $\mathcal{X}$)

FIGURE 3.3: Representation of a node in the belief propagation algorithm

3. the leaves which have no children.

From the set of evidences, each variable has to be initialized, we define $\boldsymbol{\mathcal{E}}$ the set of initialized variables and the evidence can be divided in correspondence with the parent and the children of the variables:

$$\boldsymbol{\mathcal{E}} = \boldsymbol{\mathcal{E}}_{\boldsymbol{\mathcal{A}}_{\mathcal{X}}} \cup \boldsymbol{\mathcal{E}}_{\boldsymbol{D}_{\mathcal{X}}} \tag{3.2}$$

Moreover, two types of messages λ & π are used to compute the belief (i.e. marginal probabilities) and are denoted like this:

- $\lambda(\mathcal{X}) \propto P(C_{\mathcal{X}}|\mathcal{X})$
- $\pi(\mathcal{X}) \propto P(\mathcal{X}|P_{\mathcal{X}})$

Then, we can show that belief is proportional to the multiplication of the $\lambda(\mathcal{X})$ & $\pi(\mathcal{X})$:

$$Belief = P(\mathcal{X}|\mathcal{E} = e) \propto \lambda(\mathcal{X})\pi(\mathcal{X}) \tag{3.3}$$

From those assumptions, we can now describe the way that the messages π & λ are send in order to compute the belief.

In Markov trees, belief propagation works in two phases:

1. sending λ messages upwards (from the lowest level of the tree to the root)
2. sending π message downwards (from the root to the lowest level of the tree)

First Phase In the first phase, as said before, messages are sent inwards: starting at the leaves, each node passes a message along the unique edge towards the root node. The tree structure guarantees that it is possible to obtain messages from all other adjoining nodes before passing the message on. As storing received messages in nodes is not recommended (as it would take a lot of memory and as Spark, the distributed programming tool do not recommend it), the messages have to be passed level by level as this way, it ensures that all nodes have received the λ messages from its children to compute its own $\lambda(\mathcal{X})$ which is mandatory. This continues until the root has obtained messages from all of its adjoining nodes.

The first thing is to initialize the $\lambda(\mathcal{X})$ & $\pi(\mathcal{X})$ at each node by its evidence if any:

$$\lambda(\mathcal{X}) = \mathbb{P}(\mathcal{X} = e) = 1 \tag{3.4}$$

If the node is not defined through an evidence, the leaves can be initialized. We have that

$$\lambda(\mathcal{X}) = \mathbf{1} \tag{3.5}$$

Then, λ messages can be sent to compute the λ of each parent. For each children C of the parent P, we have that:

$$\lambda_C(\mathcal{X} = x) = \sum_{child} \mathbb{P}(C = c|\mathcal{X} = x)\lambda(\mathcal{Y} = y) \tag{3.6}$$

When the intern nodes or, at the end, the root, have received all their λ children messages at a step, we can compute λ on those nodes. We have that:

$$\lambda(\mathcal{X} = x) = \prod_{C \in Child(\mathcal{X})} \lambda_C(\mathcal{X} = x) \tag{3.7}$$

Figure 3.4 represents each step in a simple example.

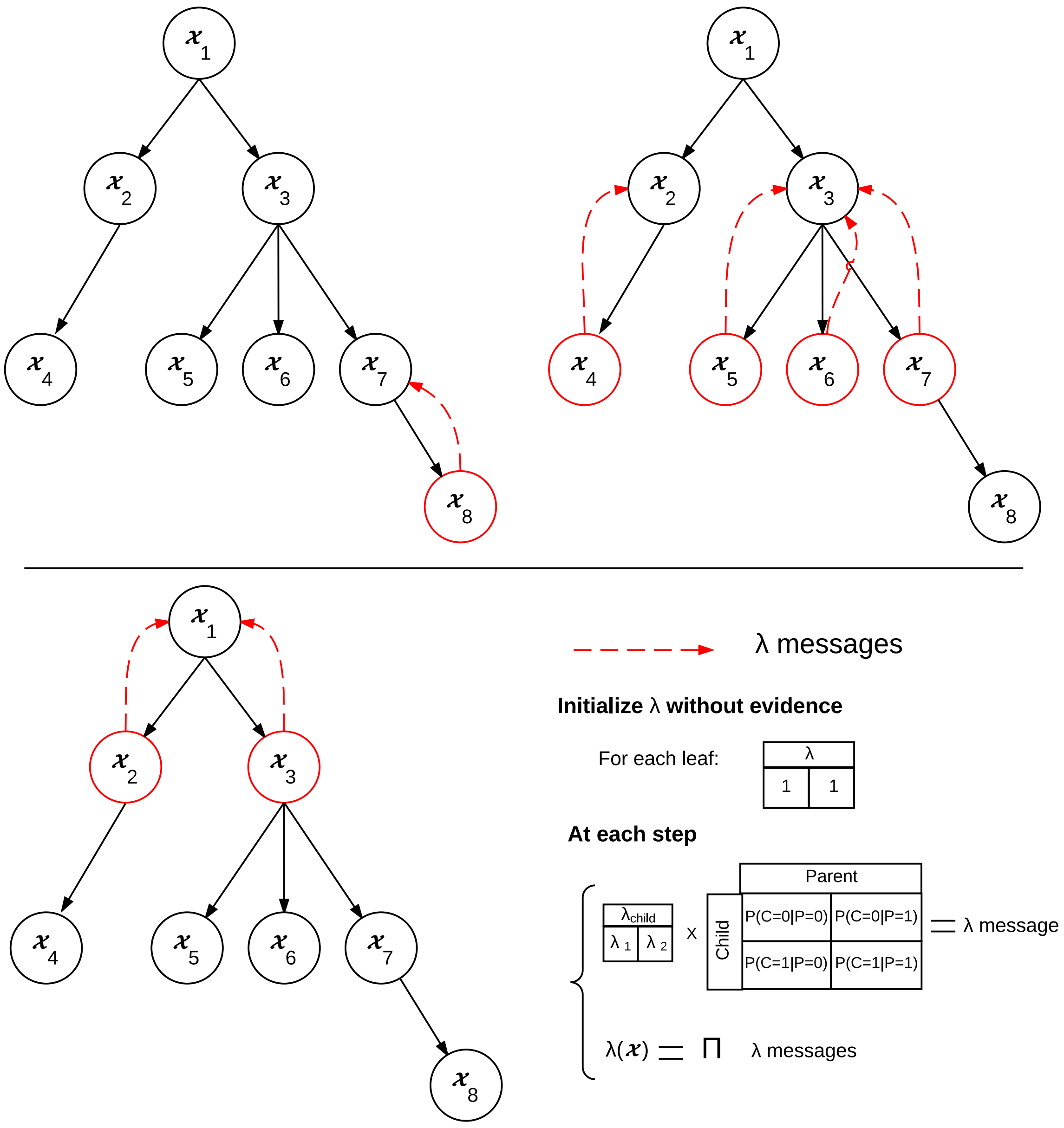


FIGURE 3.4: First phase of the belief propagation with 8 binary variables and no evidence.

Second Phase The second phase involves passing the messages outwards: starting from the root, messages are passed in the reverse direction step by step. The algorithm is completed when all leaves have received their π messages.

If the root was not defined through an evidence, it can be initialized. We have that

$$\pi(\mathcal{X}) = \mathbb{P}(\mathcal{X}) \tag{3.8}$$

Then, π messages can be sent to compute $\pi(\mathcal{X})$ of each children. For the unique parent P of C, we have that:

$$\pi_{\mathcal{X}}(P = p) = \pi(P = p) \prod_{U \in Child(P) \backslash \mathcal{X}} \lambda_U(P = p) \tag{3.9}$$

Note that the parent has to receive all the $\lambda(\mathcal{X})$ of each of its children in order to compute the π message for its children.

When the intern nodes or the leaves have received all their π messages from their unique parent P at a certain step, we can compute π on those nodes. We have that:

$$\pi(\mathcal{X} = x) = \sum_p \mathbb{P}(\mathcal{X} = x | P = p) \pi_X(P = p) \tag{3.10}$$

Finally, all the λ & π are computed, thus the belief can be computed as explained before by multiplying the $\lambda(\mathcal{X})$ and $\pi(\mathcal{X})$.

Figure 3.5 represents each step in a simple example.

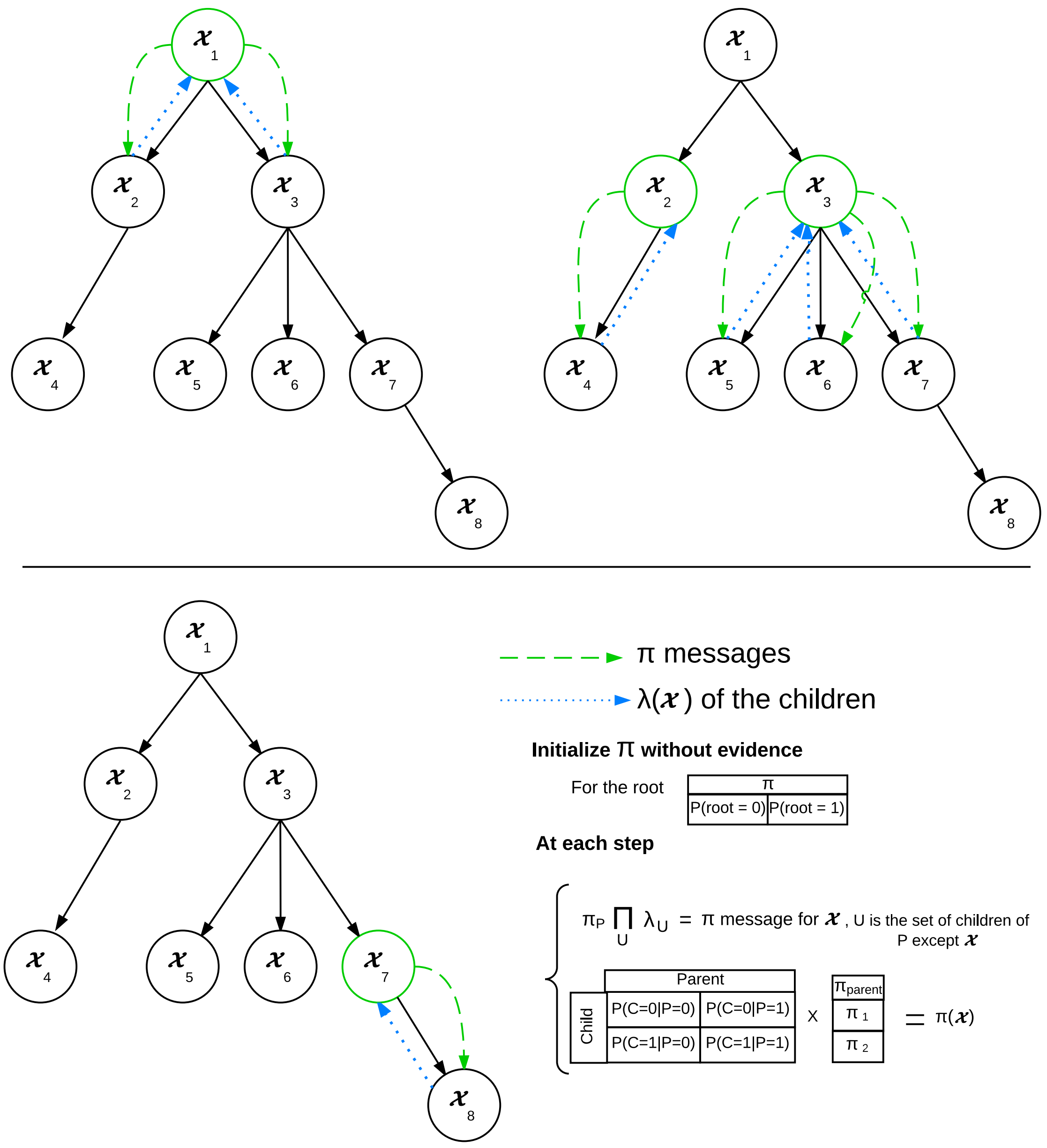


FIGURE 3.5: Second phase of the belief propagation with 8 binary variables and no evidence.

The distributed version of the belief propagation algorithm and its complexity are defined in the section 6.2.

3.2.1.1 Complexity of Inference

In this thesis, only inference on Markov trees is performed. The complexity to infer probability distribution is linear in the number of variables. In comparison with Bayesian network for example, inference requires an enormous memory size, calculation becomes very complex and even sometimes can not be completed. A version of the belief propagation algorithm named *loopy belief propagation* is an approximate algorithm whose convergence is not guaranteed used when the network is too complex. Note that probabilistic inference in general has been proven to be NP-hard.

This section has provided basic notions and concepts about statistical inference in probabilistic graphical models. Now, next Section 3.3 is about learning Markov trees automatically from data sets.

3.3 Learning Markov Trees

There are multiple ways to learn a probabilistic graphical model. It can be learned automatically, which is the assumption of this thesis and also are its parameters. It can be defined by an expert which only defines the structure and then automatically infers probabilistic parameters from the data. Moreover, it can be totally specified by an expert, to perform inference for example.

Moreover, in the case of learning, we have to define a data set to learn from the data and another to perform test to validate the models. Thus, a data set D is often split to define a learning and testing set. As this is more related to the machine learning fields, Chapter 2 has explained it in more details.

In this chapter, techniques to learn Markov trees will be discussed, explained and evaluated in term of complexity. In the context of this thesis and following the same path as the PhD thesis of François Schnitzler, learning a Markov tree can be considered as an optimization problem. We want to find the graph $\mathcal{G}$ and its parameters θ that maximize a particular score. As the aim is to learn a join probability over all variables, a common score to minimize is the Kullback-Leibler divergence of the learned model.

Let's not forget that the performances of the different methods, in a distributed environment, are one of the big goal of the thesis. Determining the complexity of the methods are a great way to have a view of the performance. Empirical tests are performed too, see Part III for more details.

3.3.1 Hypotheses

In order to learn PGMs, and in this case Markov trees, some hypotheses are set:

- *observability*: it can be partial or full, it expresses the fact that the values of some variables in $\mathcal{X}$ might be missing in a data set. The distinction must be made between the "missing" (or abnormal) variables and the variables that are called "hidden" or "latent" that might not have been measured. Partial means that only some of the values are missing; full means that no value is missing.
- *identically distributed (iid)*: it describes the fact that the observations of the learning set are considered to be independent and identically distributed. It assures that each observation ought to be viewed as a different random variable. This also means that the realizations of those different variables are independent from each other, and drawn from the same probability distribution $\mathbb{P}$.
- *faithfulness*: It means that, at least, one model in the class of candidate models can encode perfectly the conditional independence relationships contained in the data holding a probability distribution $\mathbb{P}$. Note that this hypothesis is mostly relevant for learning Bayesian network structures based on the independence relationships inferred from the learning set.

In this thesis, the assumptions of full observability and identically distributed observations are considered valid. The faithfulness assumption is not considered as Markov trees are too simple structures. They do not have enough parameters to encore all distributions exactly. This simplicity can be considered as a downside but can also be considered as a upside because it makes learning easier and faster.

Moreover this assumption is often used to establish asymptotic results when the number of available observations tends to infinity. Therefore, it may not be relevant in the context of this thesis and it is good to note that constructing models under constraints is a way to limit overfitting too.

3.3.2 Learning a Tree Structure with the Chow-Liu Algorithm

Learning a tree structure is an essential step in this thesis as it constitutes the main methods to build a Markov tree, also the main element in a mixture of tree. Learning the best tree given a data set of observations is done by using the Chow-Liu algorithm first described in a paper by Chow and Liu in 1968.

The algorithm builds a tree structure maximizing the likelihood of a learning set. It corresponds to minimizing the Kullback-Leibler divergence of the model to the empirically observed distribution $\mathbb{P}_D(\boldsymbol{x}) \triangleq \frac{N_D(\mathbf{x})}{N}$.

Practically speaking, Chow and Liu provides a simple algorithm for constructing the optimal tree; it will compute the pairwise mutual information between all the variables and construct a tree by adding maximum mutual information pair to the tree. For the sake of completeness, in this thesis, it has to be said that the Chow-Liu algorithm computes the opposite mutual information pair and adds the minimum mutual information pair to the tree (i.e. construct the minimum weight spanning tree where the weights are the opposite mutual information pair). The resulting tree is exactly the same.

Note that a more efficient tree construction algorithm for the common case of sparse data was outlined in the PhD thesis of Marina Meila in 1999 [3].

The mutual information between two variables is a measure of the variables' mutual dependence. Unlike the correlation coefficient, it is not limited to real-valued random variables, hence, mutual information is more general. It measures the similarity of the joint distribution $\mathbb{P}(\mathcal{X},\mathcal{Y})$ and the products of factored marginal distribution $\mathbb{P}(\mathcal{X})\mathbb{P}(\mathcal{Y})$. Formally, the mutual information of two discrete random variables X and Y can be defined as:

$$I_D(\mathcal{X};\mathcal{Y}) = \sum_{y\in\mathcal{Y}}\sum_{x\in\mathcal{X}} \mathbb{P}(x,y)\log\left(\frac{\mathbb{P}(x,y)}{\mathbb{P}(x)\,\mathbb{P}(y)}\right), \tag{3.11}$$

It can also be decomposed like this:

$$I_D(\mathcal{X};\mathcal{Y}) = H_D(\mathcal{X}) - H_D(\mathcal{X}|\mathcal{Y}) \tag{3.12}$$

with

$$H_D(\mathcal{X}) = -\sum_{x\in\mathcal{X}} \mathbb{P}(x)\log_2 \mathbb{P}(x) \tag{3.13}$$

and

$$H_D(\mathcal{X}|\mathcal{Y}) = -\sum_{x\in\mathcal{X},y\in\mathcal{Y}} \mathbb{P}(\mathcal{X}=x,\mathcal{Y}=y)\log_2 \mathbb{P}(\mathcal{Y}=y|\mathcal{X}=x) \tag{3.14}$$

where $H_D(\mathcal{X})$ is the Shannon's entropy of $\mathcal{X}$ and $H_D(\mathcal{X}|\mathcal{Y})$ is the conditional entropy of $\mathcal{X}$ with respect to $\mathcal{Y}$ over the data set D. Note that the unit of measurement of mutual information is the bit here as we use the logarithm in base 2.

This being defined, the different parts needed to construct the algorithm are given. We need to determine the complexity of the algorithm to assure that it is a fast way to build a tree as said before. Note that the construction of the minimum weight spanning tree is defined in its own Section 3.3.3.

The complexity of the Chow-Liu and a pseudo-code in Algorithm 1 are described in the next paragraph.

Complexity and Algorithms

The Chow-Liu algorithm can be split in two phases:

1. compute the $n \times n$ pairwise mutual information between variables. It requires $\mathcal{O}(n^2N)$ operations.
2. Build the MWST from those mutual information. Depending on the presented MWST algorithm and the structure used, the complexity may vary from $\mathcal{O}(n^2\log n)$ to $\mathcal{O}(n^2 + n\log n)$[4].

Therefore, the Chow-Liu algorithm has a total complexity of $\mathcal{O}(n^2N)$.

[4]It may be less complex, nearly linear in function of the number of edges but this method is not presented in this thesis.

Algorithm 3.3.1: Chow-Liu (CL)

```
Input  : $\boldsymbol{\mathcal{X}}$ and $D$
         Procedure MutualInformation and MinimumWeightSpanningTree
Output: Minimum Spanning Tree
Initialize : Mutual Information Matrix : MI = $[0]_{n \times n}$
for $i \leftarrow 0$ to $n-1$ do
    for $j \leftarrow i+1$ to $n$ do
        $MI[i,j] = MI[j,i] = -$MutualInformation$(\mathcal{X}_i, \mathcal{X}_j, D)$
    end for
end for
$\mathcal{T}_{CL} =$ MinimumWeightSpanningTree$(MI)$
return $\mathcal{T}_{CL}$
```

Algorithm 3.3.2: MutualInformation

```
Input  : $\mathcal{X}_{var}$, $\mathcal{X}_{cond}$ and $D$
         Procedure Entropy and ConditionalEntropy
Output: Mutual Information between two variables
return Entropy$(\mathcal{X}_{var}, D)$ − ConditionalEntropy$(\mathcal{X}_{var}, \mathcal{X}_{cond}, D)$
```

Algorithm 3.3.3: Entropy

```
Input  : $\mathcal{X}$ and $D$
Output: Shannon's entropy $H_D(\mathcal{X})$
Initialize : a structure to compute the probabilities of $\mathcal{X}$ : $P = [0]_{cardinality_{\mathcal{X}}}$
for $i \leftarrow 0$ to $nbSamples$ do                    /* compute the occurrences */
    $P[D(\mathcal{X}_i)] = P[D(\mathcal{X}_i)] + 1$
end for
$P = P/nbSamples$                                          /* compute the probabilities */
$result = 0$
for $i \leftarrow 0$ to $cardinality_{\mathcal{X}}$ do
    $result = -P[i] * \log_2(P[i])$
end for
return $result$
```

```
Algorithm 3.3.4: ConditionalEntropy
Input  : $\mathcal{X}_{var}, \mathcal{X}_{cond}$ and $D$
Output: Conditional Shannon's entropy $H_D(\mathcal{X}_{var}|\mathcal{X}_{cond})$
Initialize :
  • a structure to compute the joint probabilities of $\mathcal{X}_{var}$ and $\mathcal{X}_{cond}$
    $JP = [0]_{cardinality_{\mathcal{X}_{var}} \times cardinality_{\mathcal{X}_{cond}}}$
  • a structure to compute the probabilities of $\mathcal{X}_{cond}$
    $P = [0]_{cardinality_{\mathcal{X}_{cond}}}$
for $i \leftarrow 0$ to $nbSamples$ do
    /* compute the co-occurrences of $\mathcal{X}_{var}$ and $\mathcal{X}_{cond}$ */
    $JP[D(\mathcal{X}_{var_i}), D(\mathcal{X}_{cond_i})] = JP[D(\mathcal{X}_{var_i}), D(\mathcal{X}_{cond_i})] + 1$
    /* compute the occurrences of $\mathcal{X}_{cond}$ */
    $P[D(\mathcal{X}_{cond_i})] = P[D(\mathcal{X}_{cond_i})] + 1$
end for
$JP = JP/nbSamples$                /* compute the joint probabilities */
$P = P/nbSamples$                  /* compute the probabilities */
$result = 0$
for $i \leftarrow 0$ to $cardinality_{\mathcal{X}_{var}}$ do
    for $j \leftarrow 0$ to $cardinality_{\mathcal{X}_{cond}}$ do
        $result = -JP[i,j] * \log_2(JP[i,j]/P[j])$
    end for
end for
return $result$
```

3.3.3 Minimum Weight Spanning Tree (MWST) Algorithms

This section is to remind the main algorithms to construct a MWST. As they have been a source of inspiration and work, explaining those three algorithms (Kruskal, Prim and Boruvka) is a great way to have a clear view of the construction of a MWST. Note that those algorithms have been thought in a centralized fashion (opposed to distributed fashion), they can not be directly transposed to a distributed environment. In Section 6.1.2, some explanations are provided about the way those algorithms could be transformed to fit the distributed Spark tool; moreover, a message passing algorithm, ideal for a distributed environment in order to construct the MWST, is provided in this section but it may be easier to begin with explanations on more traditional MWST algorithms.

Node that, beyond the scope of this thesis, MWST have also practical applications. For example[5], it can be used by telecommunications company laying cable to a new neighborhood. If it is constrained to bury the cable only along certain paths (e.g. along roads), then this system could be represented by a graph which points are connected by those paths. As some paths might be more expensive, because they are longer, or require to be buried the cable deeper in the ground. Then, these paths would be represented by edges with larger weights. Those weights could be a particular currency which would be an acceptable unit. Therefore, a spanning tree for that graph would be a subset of those paths containing no cycles but still connects to every house. If some edges have the same weight, the minimum spanning tree is not unique, weights could be differentiated[6] to build an unique tree. The constructed minimum weight spanning tree would be one with the lowest total cost and thus would represent the least expensive path for laying the cable.

With a practical example in mind, Kruskal's algorithm will be the first to be described. Then Prim's and Boruvka's algorithm will be defined.

3.3.3.1 Kruskal's Algorithm

Kruskal's algorithm is a greedy algorithm in graph theory that finds a minimum weight spanning tree for a connected weighted graph. This algorithm has been developed by Joseph Kruskal and first appeared in Proceedings of the American Mathematical Society in 1956 [4].

Constructed from a graph (complete in our case), this algorithm follows two main steps:

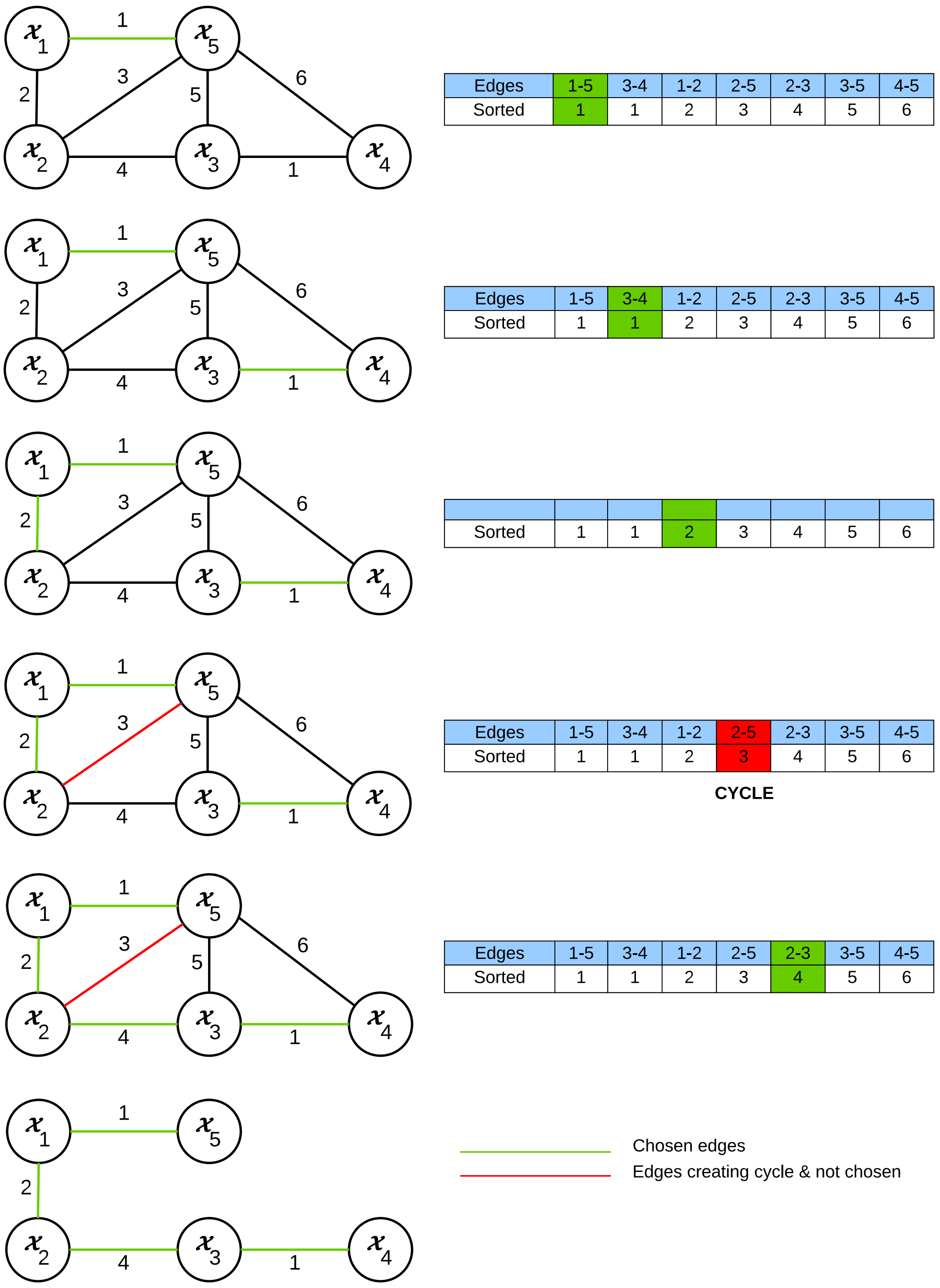
1. sort $E(\mathcal{G})$ in function of their weight, from the highest to the lowest;
2. go through the sorted edges and construct the tree by taking the edges in order unless it creates a cycle in the graph.

[5]This example refers to the minimum weight spanning tree algorithm. In order to go from a minimum spanning tree to a maximum one, simply taking the opposite weight of each edge to construct the tree makes the trick.

[6]Lexicographic order of their endpoints can be used

The algorithm stops when the tree is spanning (i.e. if the number of taken edges equals the number of vertices minus one). Note that the sorting is well-suited to be performed in parallel, the rest of the algorithm is unfortunately not much suited as it needs to perform iterative steps and using "Find" and "Union" operations which may be expensive in the case of distributed computing. Figure 3.6 illustrates a simple example of the algorithm.

Complexity of Kruskal's algorithm and also its pseudo-code in Algorithm 5 are given in the next paragraph.



Edges	1-5	3-4	1-2	2-5	2-3	3-5	4-5
Sorted	1	1	2	3	4	5	6

Edges	1-5	3-4	1-2	2-5	2-3	3-5	4-5
Sorted	1	1	2	3	4	5	6

Sorted	1	1	2	3	4	5	6

Edges	1-5	3-4	1-2	2-5	2-3	3-5	4-5
Sorted	1	1	2	3	4	5	6

CYCLE

Edges	1-5	3-4	1-2	2-5	2-3	3-5	4-5
Sorted	1	1	2	3	4	5	6

🟩 Chosen edges
🟥 Edges creating cycle & not chosen

FIGURE 3.6: Example of Kruskal's algorithm

Complexity and Algorithm

Consider a graph $\mathcal{G}$ containing V vertices and E edges. It has be shown that Kruskal's algorithm can run in $\mathcal{O}(E \log E)$ time using simple structures. Moreover, Its running time is also equivalent to $\mathcal{O}(E \log V)$ because we have that $\log E = \log V$ as E is at most V^2 & $\log V^2 = 2 \log V \equiv O(logV)$.

Of course, the complexity can be deduced from its different parts:

Sorting the edges by weight using a comparison sort has a complexity of $\mathcal{O}(E \log E)$.

Then, using a particular structure (e.g. disjoint-set) to memorize which vertices are in which components, $\mathcal{O}(V)$ operations are performed. Indeed, in each iteration two vertices are supposed to be connected through one edge in the spanning tree if it does not form a cycle. To detect cycles, this requires to watch each component to find if the vertices are not already connected and to union them if not. Thus the total complexity is $\mathcal{O}(E \log E) = \mathcal{O}(E \log V)$.

Algorithm 3.3.5: Kruskal

Input : Symmetric matrix M of positive weights

Output: Minimum Weight Spanning Tree

Initialize :

- $\mathcal{T}$ an empty graph with V vertices where V =the number of rows in M
- sort M by decreasing order

```
foreach (u,v) taken in a decreasing order in $M$ do
    if $\mathcal{T}$ is fully connected then                 /* i.e. $V-1$ edges added */
        return $\mathcal{T}$
    else if vertices u and v are not connected yet then
        $\mathcal{T} = \mathcal{T} \bigcup (u,v)$                      /* connect both vertices */
end foreach
```

3.3.3.2 Prim's Algorithm

Prim's algorithm is also a greedy algorithm. It has been developed in 1930 by a Czech mathematician named Vojtech Jarnik and later independently by computer scientist

Robert C. Prim in 1957 [5] and rediscovered by Edsger Dijkstra in 1959. It follows rather simple steps like the Kruskal's algorithm.

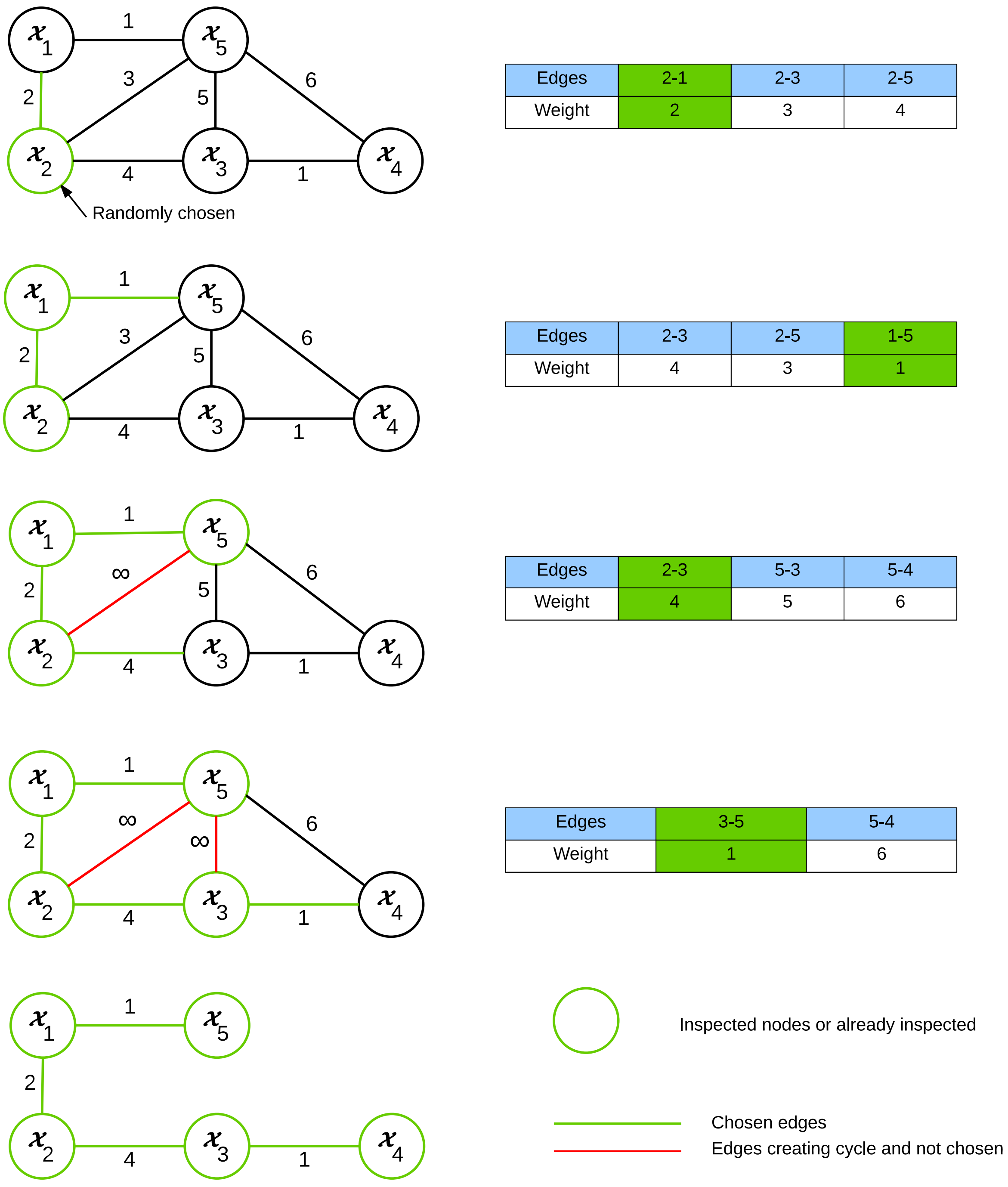
From a complete graph, the algorithm follows those steps:

1. pick a single vertex, chosen arbitrarily from the graph;
2. among the set of edges linking vertices not yet in the tree, add the minimum weight edge in the tree;
3. repeat step 2 until all vertices are in the tree.

Then the tree is build. Note that the idea of computing the minimum edges for the set of edges at step 2 could be easily done in parallel.

Figure 3.7 illustrates a simple example of the algorithm.

Complexity of the Prim's algorithm and also its pseudo-code in Algorithm 6 are defined in the next paragraph.



x_1
x_2
x_3
x_4
x_5
1
2
3
4
5
6
∞
Randomly chosen
Inspected nodes or already inspected
Chosen edges
Edges creating cycle and not chosen

Edges	2-1	2-3	2-5
Weight	2	3	4

Edges	2-3	2-5	1-5
Weight	4	3	1

Edges	2-3	5-3	5-4
Weight	4	5	6

Edges	3-5	5-4
Weight	1	6

FIGURE 3.7: Example of Prim's algorithm

Complexity and Algorithm

The complexity of Prim's algorithm is very dependent on the data structure used. Indeed using an adjacency matrix graph representation and linearly searching in this array of weights the minimum weight edge requires $\mathcal{O}(V^2)$ running time. An adjacency list representation could be great for sparse graph but not in our case. However, heaps are a great structure to find minimum weight edges in the algorithm's second step. As it is not the goal of this thesis and as heaps [7] are not actual usable structure in the distributed computing fields (as it would create too many links between the distributed element and then oblige to send messages between them), the heap structure will not be described.

Algorithm 3.3.6: Prim

```
Input  : Symmetric matrix $M$ of positive weights
Output: Minimum Weight Spanning Tree
Initialize :
  • $\mathcal{T}$ an empty graph with V vertices where V =the number of rows in $M$
  • a set of vertices $V_{new} = \{x\}$, where x is an arbitrary node (starting point) $\in$ V
repeat
    /* if multiple edges have the same weight, pick one edge randomly */
    Find the minimum weight edge {u, v} s.t. u $\in V_{new}$ and v $\notin V_{new}$
    $V_{new}$ ++ v                              /* add v to the set $V_{new}$ */
    $\mathcal{T} = \mathcal{T} \bigcup (u, v)$                 /* connect the two vertices */
until $Vnew = V$
```

3.3.3.3 Boruvka's Algorithm

Boruvka's algorithm is a greedy algorithm that has been developed by Boruvka in 1926 [6], long before computers even existed as a method of constructing an efficient electricity network (similarly to the example explained ealier).

The algorithm first considers each vertex as a sub-tree of the final tree. Then, it starts by checking each sub-tree (i.e. vertex) and adding its minimum weight edge in the graph, without considering the already added edges. It goes on, grouping growing sub-trees

[7]Complexity using heap can be decreased to $\mathcal{O}(E + V \log V)$.

(vertices) together through their minimum weight edges until a single group, constituting a minimum weight spanning tree, is left.

Note that the step where each vertices search for its minimum weight edge is well-suited to be performed in a distributed environment while regrouping the sub-trees is quite complex.

Figure 3.8 illustrates a simple example of the algorithm.

Complexity of the Prim's algorithm and also its pseudo-code in Algorithm 7 are defined in the next paragraph.

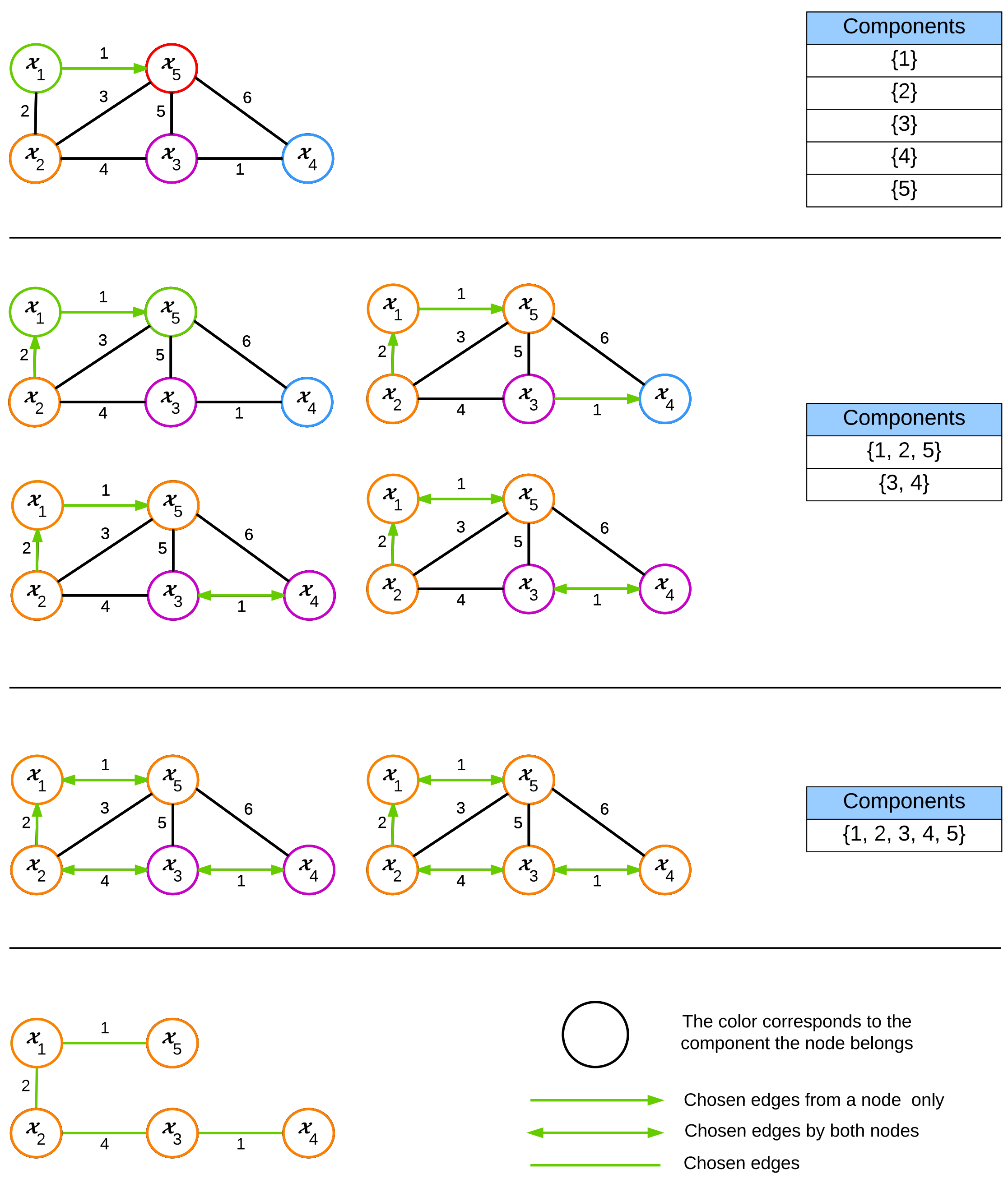

Components
{1}
{2}
{3}
{4}
{5}
Components
{1, 2, 5}
{3, 4}
Components
{1, 2, 3, 4, 5}
The color corresponds to the component the node belongs
Chosen edges from a node only
Chosen edges by both nodes
Chosen edges

FIGURE 3.8: Example of Boruvka's algorithm

Complexity and Algorithm

For the complexity, for each pass, the number of sub-trees (i.e. groups of vertices) is reduced by a factor two. Thus, there are $\mathcal{O}(\log V)$ passes. Each pass takes time $\mathcal{O}(E)$ to figure out which sub-tree each vertex is in, then finding the minimum edge connecting to sub-trees. Therefore, the complexity of the algorithm is $\mathcal{O}(E \log V)$

Algorithm 3.3.7: Boruvka

```
Input  : Symmetric matrix $M$ of positive weights
Output: Minimum Weight Spanning Tree
Initialize : a forest $\mathcal{F}$ to be a set of one-vertex trees, one for each vertex of the graph
while $\mathcal{F}$ has more than one tree do
    foreach tree $\mathcal{T}$ of $\mathcal{F}$ do
        $\mathcal{S}$ an empty set of edges
        foreach vertex v in $\mathcal{T}$ do
            Find e1 the edge with the minimum weight from v to a vertex outside of $\mathcal{T}$
            $\mathcal{S}$ ++ e1                                /* add e1 to $\mathcal{S}$ */
        end foreach
        Find e2 the edge with the minimum weight in $\mathcal{S}$
        Connect two trees by e2 in $\mathcal{F}$
    end foreach
end while
return $\mathcal{F}$
```

3.3.4 Construct Markov Trees & Compute their Parameters

In order to create a Markov tree, we have to orient the edges of the tree constructed by the Chow-Liu algorithm. The idea is to choose randomly a node which will be considered as the root and all the edges will be oriented outwards the root. This phase is explained in more details in Section 6.1.3.

From this structure, parameters will be learned from the data set. Of course, the parameters could also be set by an expert to tune the structure but it is not the aim of this thesis. Those parameters are in fact the values of the conditional probabilities defined by the edges of the graph $\mathcal{G}$:

$$\theta_{i,x_i|\mathbf{a}} = \mathbb{P}_{\mathcal{G}}(\mathcal{X}_i = x_i|\mathcal{P}a_{\mathcal{G}}^{\mathcal{X}_i} = \mathbf{a}) \tag{3.15}$$

Essentially, we want to find a most likely value of θ given a data set D, that corresponds to $arg \max_{\theta} \mathbb{P}(\theta|D, \mathcal{G})$. According to Bayes Rule, we have

$$\mathbb{P}(\theta|D, \mathcal{G}) = \frac{\mathbb{P}(D|\mathcal{G}, \theta)\mathbb{P}(\theta|\mathcal{G})}{\mathbb{P}(D)} \tag{3.16}$$

where the terms have the following meanings:

- $\mathbb{P}(\theta|D, \mathcal{G})$: Posterior
- $\mathbb{P}(D|\mathcal{G}, \theta)$: Likelihood
- $\mathbb{P}(\theta|\mathcal{G})$: Prior
- $\mathbb{P}(D)$: Evidence

From that, two[8] major ways to compute the parameters of the Markov tree:

- the maximum likelihood estimation (MLE) where the goal is to select the value of the parameters maximizing the likelihood of the observed samples

  $$\theta_{ML} = arg \max_{\theta} \mathbb{P}(D|\mathcal{G}, \theta); \tag{3.17}$$

  which are, practically speaking, defined like this:

  $$\theta_{i,x_i|\mathbf{a}} = \frac{N_D(\mathbf{a}, x_i)}{\sum_{x_i \in Val(\mathcal{X})} N_D(\mathbf{a}, x_i)} \triangleq \mathbb{P}_D(x_i|\mathbf{a}) \tag{3.18}$$

  Where the numerator corresponds to the number of samples of the data set D where the parent $\mathcal{P}a_{\mathcal{G}}^{\mathcal{X}_i} = \mathbf{a}$ and $\mathcal{X}_i = x_i$. The denominator corresponds to the number of samples of the data set D.

[8]The maximization a posteriori (MAP), a Bayesian estimate is also often used

- the Bayesian parameter estimation refers to the Bayes rule, defining a prior distribution $\mathbb{P}(\theta|\mathcal{G})$. In this case, the goal is to select the maximum a posteriori value of the parameters:

  $$\theta_{MP} = arg \underset{\theta}{max} \mathbb{P}(D|\mathcal{G},\theta)\mathbb{P}(\theta|\mathcal{G}) \tag{3.19}$$

  which are also defined like this:

  $$\theta_{i,x_i|\mathbf{a}} = \frac{1 + N_D(\mathbf{a}, x_i)}{|Val(\mathcal{X}_i)| + \sum_{x_i \in Val(\mathcal{X})} N_D(\mathbf{a}, x_i)} \triangleq \mathbb{P}_D(x_i|\mathbf{a}), \tag{3.20}$$

  where using such a prior distribution assures that the probabilities encoded are strictly positive. It is performed by adding one in the numerator and adding the cardinality of the variable in the denominator.

Here is an intuitive example of each parameter estimation:

Let's imagine that you are a doctor. You have a patient who presents some odd symptoms. You search in an encyclopedia and find out that the disease could be either a common cold or lupus.

It is written in your book that if a patient has lupus then the probability that he will present these symptoms is about 90% which is quite high.

You also read that if the patient has a common cold then the probability that he will show these symptoms is only 10%, which is pretty worrying.

Which disease does your patient is more likely to have?

Well, there are three approaches to consider. If you used MLE, you would declare:

"The patient has lupus. Lupus is the disease which maximizes the likelihood of presenting these symptoms."

You, as a doctor, referring to your encyclopedia.

However, a wise doctor would remember that lupus is very rare (the prior probability). This means that the prior probability of anyone having lupus is very low (almost 5 cases per 100K people) compared to common colds. Using a Bayesian estimate you should decide that the patient is more likely to have a common cold than lupus.

In fact, Bayesian assumes a pre-knowledge for the problem and uses it whereas MLE uses present data only and tries to find the best descriptive model.

Note that, in this thesis, both techniques have been implemented but the Bayesian techniques is used.

Before defining mixture of Markov tree, the complexity of learning a Markov tree is discussed in the next paragraph.

Complexity of Learning a Markov Tree

Unlike Bayesian networks, learning Markov tree is easier. As suggested in the previous sections, using multiple simple structures as Markov tree could give a good way to model more complex systems.

Learning a Bayesian network could be very complex; exponential in the number of variables even if strong constraints are defined to restrict the structure (e.g. limit the number of parents of one variable).

Therefore, using the Chow-Liu method which has a complexity of $\mathcal{O}(n^2N)$ to create a mixture seems a good way to tackle the complexity issue of learning much more complex PGMs while using multiple of them to create a mixture.

Chapter 4

Mixtures of Markov Trees

The aim of this chapter is to define how to build mixtures of Markov tree and how they are a way to model more complex distributions. First, the use of mixture to model probability will be explained and then the technique to construct a mixture.

4.1 Presentation

The principle of mixture is simply the idea of creating a set of models which are able to provide predictions for a specific problem. Typically, the predictions of a mixture are an average of each model in it. Therefore, the resulting predictions of a set of variables $\mathcal{X}$ are computed like this:

$$\mathbb{P}(\mathcal{X}) = \sum_{i=1}^{m} \lambda_i \mathbb{P}_i(\mathcal{X}), \qquad \sum_{i=1}^{m} \lambda_i = 1, \lambda_i > 0 \ \forall i \tag{4.1}$$

where λ_i corresponds to the weight of each model and, of course, $\mathbb{P}_i(\mathcal{X})$ corresponds to the probability of the i^{th} model of the mixture.

A simple example to understand the usefulness of a mixture is to approximate a complex densities by a mixture of two weighted normal densities[1]. Here is an illustration in Figure 4.1

which give a simple example of the combination of two probability distributions.

[1]A normal density is characterized by its mean μ and its variance σ^2 written $\mathcal{N}(\mu, \sigma)$

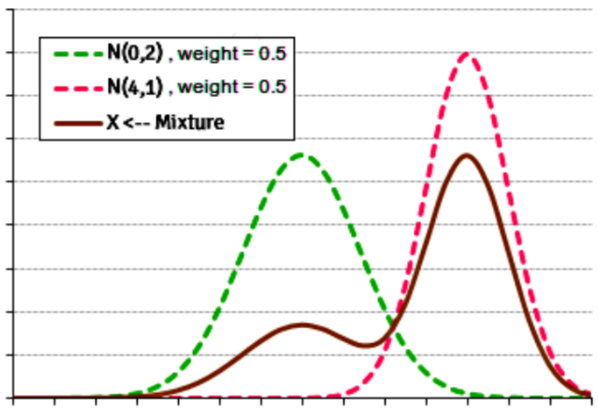


FIGURE 4.1: Representation of a mixture of two normal densities with the same weight to approximate a more complex density.

Therefore, combining multiple models can increase the modelling power and thus, reduce the bias or the variance. Note that multiple techniques to create mixture exist, two ways to create a mixture to decrease the bias and the variance are explained in the next section.

4.2 Mixture to Reduce the Bias and the Variance

4.2.1 Perturb and Combine to Create Mixture to Reduce the Variance

This section focuses on the way to generate multiple models from a given learning set with the perturb and combine techniques. As its name suggests, this technique supposes to first **perturb** either the learning set or the learning algorithm with randomness and then to **combine** the perturbed models in a certain way into an aggregated one. Therefore, with this framework, an ensemble of models can be constructed and combined; this is called a **mixture**. Each model alone will maybe have a higher variance and bias but the combination of each of them could produce a "better" model with lower variance. The randomization can be placed in the algorithm by replacing a value by a random one; the data set can also be modified randomly (see the next paragraph about bootstrap aggregation). Of course, the challenge in this context is to find the way to use this randomization at the right element to produce a better model with lower variance without increasing the bias too much (see bias-variance dilemma in the section 2.5). A well-known technique of perturb and combine is the *Bootstrap Aggregation*. It will now be described.

Bootstrap Aggregation

Bootstrap aggregating, also called bagging, is a machine learning ensemble meta-algorithm designed to improve the stability and accuracy of machine learning algorithms. As said before, the randomness is induced through a modification of the learning set. It may be used when only a small amount of data is available; this technique would compensate this lack of data by creating multiple bootstrap replicates of the original learning set. Next, the learning algorithm is applied on those replicates. The final prediction consists in averaging the predictions of each created models.

A bootstrapped replicates D' of size N is obtained by taking N' randomly observations of the original data set D. More formally, a random number r_i will be drawn N' times and D' is constructed by taking

$$\mathbf{x}_{D'_i} = \mathbf{x}_{D_{r_i}} \qquad \forall i \in [1, N'], \tag{4.2}$$

where $\mathbf{x}_{D_i}$ (resp. $\mathbf{x}_{D'_i}$) refers to the i^{th} observation of D (reps. D').

When the size of the replicate is the same than the original set (i.e. $N = N'$), then the replicate is expected to have a fraction ($\approx 63.2\%$) of the observations in D.

It also reduces variance and helps to avoid overfitting. Bagging is a special case of the model averaging approach.

Figure 4.2 illustrates the perturb and combine techniques with bootstrap aggregation to create a mixture.

This technique is used in this thesis to build the MCL-B algorithm, see section 7.1 for more details.

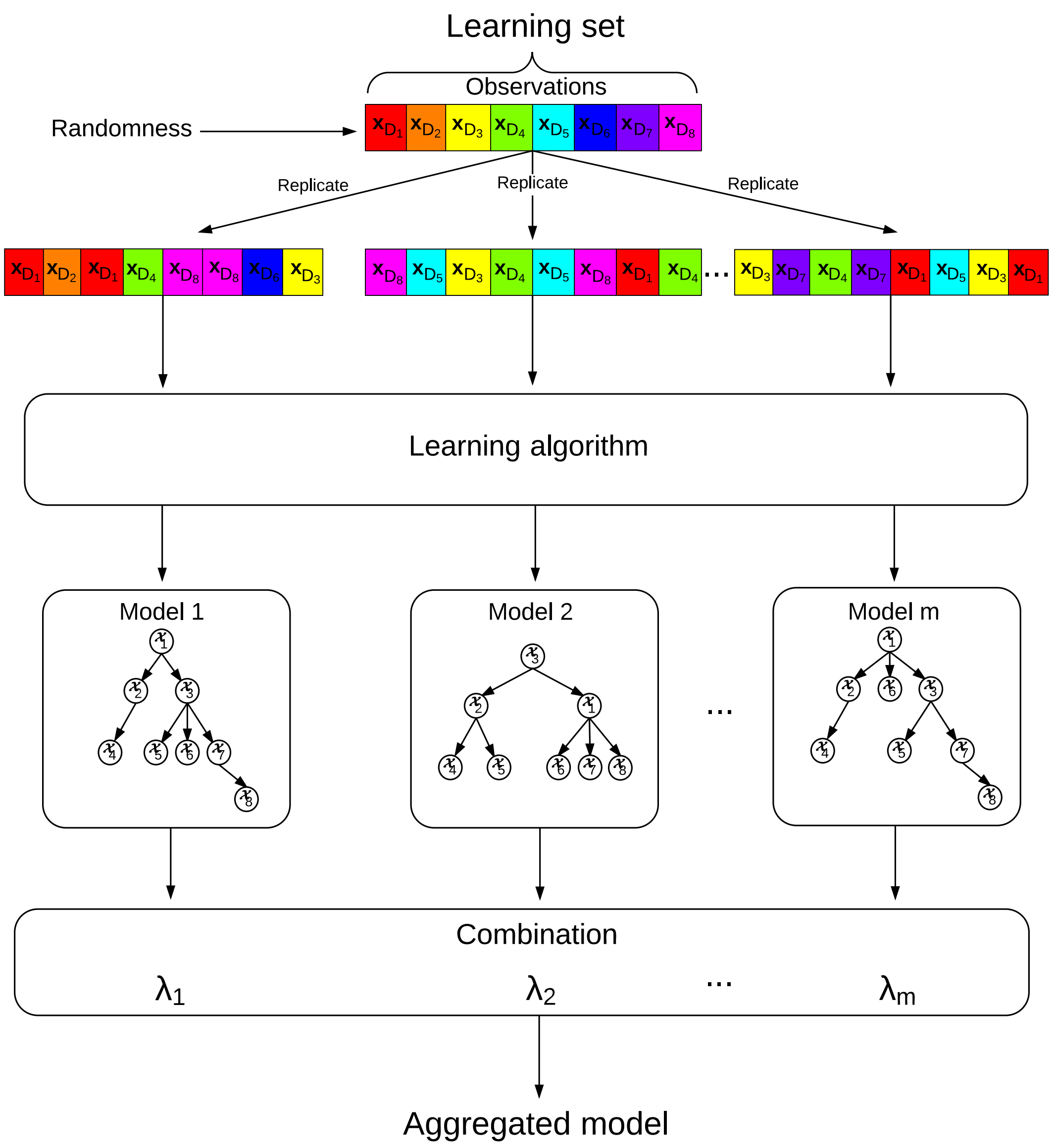


FIGURE 4.2: Representation of the perturb and combine framework performing bootstrap aggregation to create a mixture of models. The bootstrap replicates are randomly chosen with replacements from the original set of observations (learning set) with $N = N'$.

4.2.2 Expectation-Maximization to Reduce the Bias

One way to reduce the bias is the Expectation-Maximization (EM) algorithm. EM is an iterative algorithm used to search maximum likelihood or maximum a posteriori (MAP) estimates of parameters where model is function of unobserved hidden variables. The idea of the algorithm is to alternate between two phases:

1. *the expectation phase*: a function is created to evaluate the expectation of the log-likelihood thanks to the current estimate for the parameters.
2. *the maximization phase*: parameters that maximize the expected log-likelihood found on the E step are computed.

The new parameters are then used to determine a new distribution of the hidden variables in the next expectation phase. Those two phases will be performed iteratively until the new parameters do not change from a pass to another.

This algorithm is at the base of the MT-EM algorithm described in Section 7.2.

Chapter 5

Distributed Computing & Spark

The aim of this chapter is to introduce the main concepts of the distributed computing. It is also to present the tool used in this thesis: Spark.

5.1 Introduction

The word distributed in terms such as "distributed system", "distributed programming", and "distributed algorithm" originally referred to computer networks where individual computers were physically distributed within some geographical area. The terms are nowadays used in a much wider sense, even referring to autonomous processes that run on the same physical computer and interact with each other by message passing. While there is no single definition of a distributed system, the following defining properties are commonly used:

- There are several autonomous computational entities connected over the network, each of them has its own local memory;
- The entities communicate with each other by message passing.

In this thesis, the computational entities are named computers or clusters (clusters can be defined as an ensemble of computers working together sometimes called "node").

A distributed system may have a common goal, such as solving a large computational problem. Alternatively, each computer have to deal with its individual computation processes, and the purpose of the distributed system is to coordinate the use of those computers (cluster) and their resources with communication techniques in order to compute at a large scale a problem that has been distributed.

Other typical properties of distributed systems include the following:

- the system can cope with failures and recover from them;
- the system is made of different kinds of computers linked in a network. The structure of the network is not known in advance. The system may also vary during the execution of a distributed program;
- each cluster has incomplete and a limited view of the whole system. Only some part of the input are known by each cluster.

A typical architecture is shown in figure 5.1. A user connects himself to a master (computer) and sends him a processing request. The master will distribute and coordinate the processing request to workers (also computers) send by the user.

Furthermore, to have a clearer view of the often associated terms, defining the difference between "parallel programming" (also named "concurrent programming") and " distributed programming" may be useful:

- **parallel computing**: A shared memory is available for each (or some) processor and give the possibility to exchange information between processors;
- **distributed computing**: each processor owns its private memory which is distributed. Information is exchanged only by passing messages between the processors.

Keep in mind parallel and distributed are two different concepts even if they share some common notions[1]; high-performance parallel computation in a shared-memory

[1]Parallel is often use to described multiple processes done at the same time. Thus, it is one reason why those terms are sometimes misused.

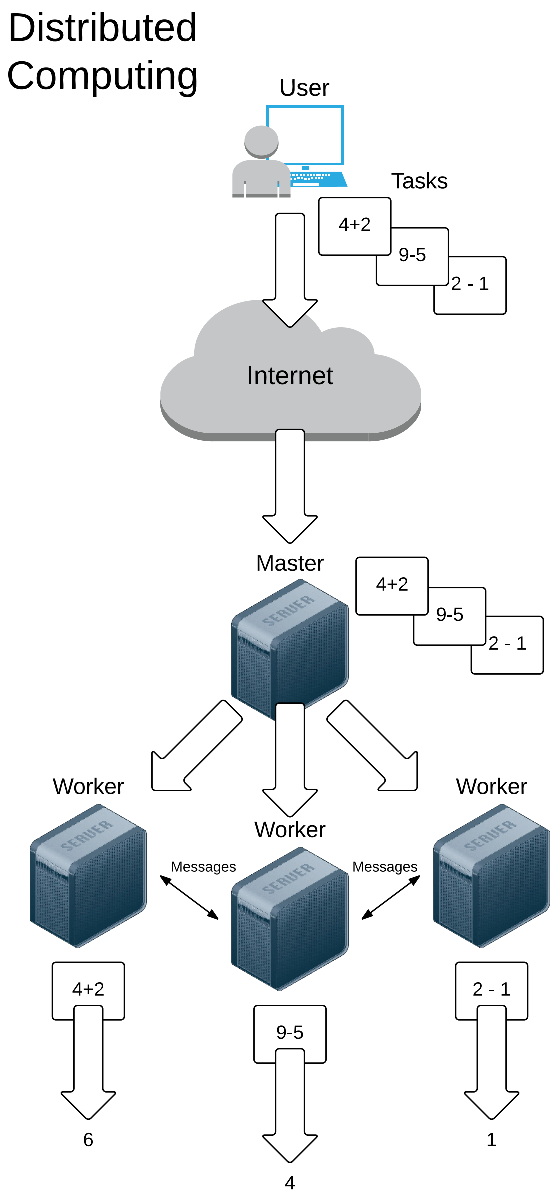


FIGURE 5.1: Illustration of a server (cluster) architecture.

multiprocessor uses parallel algorithms while the coordination of a large-scale distributed system uses distributed algorithms.

Figure 5.2 illustrates the two different contexts of the distributed and parallel computing.

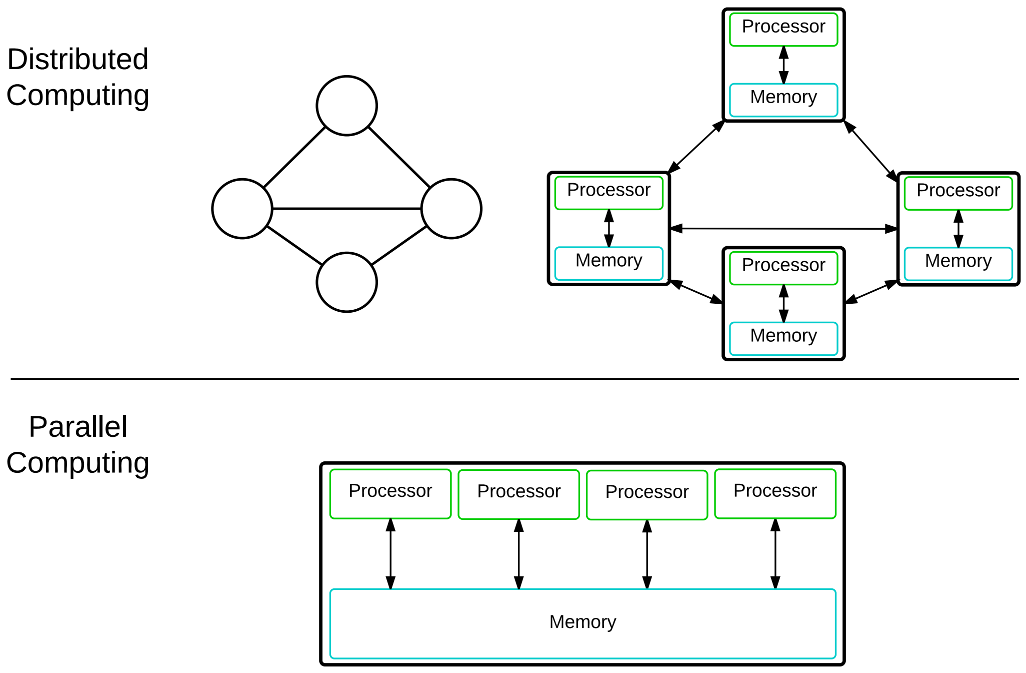


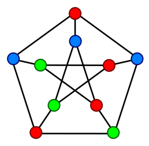
FIGURE 5.2: Illustration of the differences between distributed and parallel computing. The upper left figure represents a schematic view of a distributed system, it is a network topology in which each node is a cluster and are connected to communicate; the right one is the same topology in more details. The bottom figure represents a parallel system with processors and their shared memory.

In distributed algorithms, like in this thesis, computational problems are often related to graphs (see Section 5.3 about GraphX, the computational graph library that has been used). The topology of the network is often the same as the graph or may be at least simulated to approach it; the more the topology looks like the graph, the more the algorithms would be efficient and distributed (and scalable) as the communication between nodes would be shorter (as locality of the different computer node in the network influences the communication costs).

Now, an example, at a high level of abstraction, about a graph algorithm is given. It is used to give a rough idea about the differences of distributed, parallel and centralized[2] graph algorithms in order to understand in a pragmatic way the nuances of those concepts:

[2]Opposed to distributed, it is made up of one central computer which may be accessible by multiple terminals.

Consider the computational problem, in its simplest form, of coloring the vertices of a graph $\mathcal{G}$ such that adjacent vertices do not share the same color; this problem is called the vertex coloring.



Centralized algorithms $\mathcal{G}$ could be encoded as a string, and the string is given as input to a computer. The computer program finds a coloring the vertices of $\mathcal{G}$ by simply looking over the string containing all the needed information. Then, it encodes the coloring as a string and returns it.

Parallel algorithms

- $\mathcal{G}$ is also encoded as a string. Nevertheless, the string is accessible in parallel by multiple computers. Then, each computer might compute a coloring for only one part of $\mathcal{G}$;
- the aim here is to exploit the processing power of multiple computers in parallel.

Distributed algorithms in message-passing model

- The network topology corresponds to the structure of $\mathcal{G}$ or is, at least, simulated it if not enough resources are available. Each node in $\mathcal{G}$ corresponds to a computer and the edges are the communication links. First, each computer only knows about its direct neighbours in $\mathcal{G}$. Then, the computers exchange messages with each other to discover the structure of $\mathcal{G}$. Finally, each computer must return its own color. The thing to understand is that each node has its own information, memory, resources and have to communicate in order to perform this algorithm;
- the aim here is also to coordinate the operation of an arbitrary distributed system;
- in distributed computing, the complexity is more often measured through the number of communication operations than computational steps.

Hence, the three different approaches described are rather different. This give an idea on how graph algorithms in a distributed environment have to be developed.

Of course, distributed algorithms can also solve non graph related problems but the vision of graph is a good way to understand the constraints of a distributed algorithm.

One of the major challenges in developing and implementing distributed algorithms is successfully coordinating the behavior of the independent parts of the algorithm in the face of processor failures and unreliable communications links. The choice of an appropriate distributed algorithm to solve a given problem depends on both the characteristics of the problem, and characteristics of the system the algorithm will run on such as the type and probability of processor or link failures, the kind of inter-process communication that can be performed, and the level of timing synchronization between separate processes

Furthermore, the distributed approach may give access to higher computational resources but has more constraints as computers have to communicate between them with message, a coordination has to be taken into account. Another important aspect of distributed computing is the network. All computers are connected and communicate through a network which is limited by its bandwidth. Therefore, the speed of the network is an important aspect that has to be taken into account when performance of algorithms is evaluated.

Moreover, challenges in implementing distributed algorithms is to greatly deal with the coordination of the computers, face process failures and unreliable communications links. Fortunately, some tools managing such challenges have been developed and one of them, cited multiple times, is Spark.

Therefore, as basic notions are now defined, the next section will describe the tool Spark used in this thesis to perform distributed computing and its particularities.

5.2 Spark

Spark

Spark is a cluster computing platform designed to be fast and general-purpose. It is used for distributed data analysis; a field that has been growing during the last decade. Spark is becoming one of the most popular open-source platform.

To support more types of computations (like interactive queries and/or stream processing), Spark uses the fast and popular MapReduce model (see Section 5.2.1). As it has been developed to process large data sets, speed is very important meaning the distinction between interactivity and delayed activity.

Spark is also designed to cover a wide range of workloads. This generality includes for example batch applications, iterative/recursive algorithms, interactive queries, and streaming. Spark integrates all these workloads in the same platform, making it easy and inexpensive to combine different processing types, which is often necessary in production data analysis pipelines. In addition, as every thing is grouped in one place, it decreases the need of management of separated tools.

Spark is built to be accessible with simple APIs[3] in Scala, Java, Python or even SQL. Scala is the one chosen and used in this thesis (see Section B.1 where Scala is briefly described). Big Data tools are integrated too, Spark can run in *Hadoop* clusters, which are well-known type of cluster, and access to any file stored in the *Hadoop Distributed File System* (i.e. distributed data set).

Hadoop is an open source, Java-based programming framework drawn from GoogleFS. It supports the processing of large data sets in a distributed computing environment.

The core of Apache Hadoop consists of a storage part (*Hadoop Distributed File System* (HDFS)) and a processing part (Spark using the MapReduce framework).

Hadoop splits files into large blocks, replicates and distributes them amongst the workers. Hence, it provides the possibility to access those data in parallel and the fault-tolerant property as if a worker fails to perform a task for some reasons, the data can be switch to another worker which will perform the task.

To process the data, Spark using MapReduce will transfer the tasks for workers to compute them, based on the data each worker need to process. The concept of MapReduce is explained in its dedicated paragraph in Section 5.2.2.

[3]Application Programming Interface.

In order to have a clear view of Spark and the architecture of the tool, here is a small explanation of its construction.

Multiple integrated components are contained in Spark. Of course, there is the core of Spark which is the computational engine. It deals with the scheduling, distributing and monitoring applications consisting of multiple tasks across multiple workers or a computing cluster. This core is behind many higher-level components which are specific to other tasks such as machine learning or graph computation. This creates the strength and the generality of Spark, giving the possibility to juggle with multiple components each built for a specific task.

Figure 5.3 is an illustration of the architecture of the whole Spark [7]:

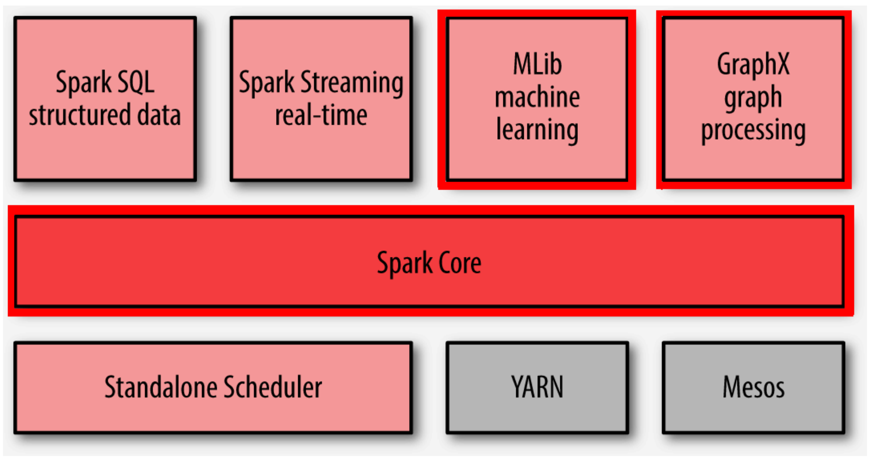


FIGURE 5.3: Composition of the Spark tool

The components used in this thesis are related to the MLlib[4] components. The GraphX (and of course the Spark core) component is used in this thesis to perform distributed computing on the Markov trees.

Spark can use three different schedulers: "Standalone Scheduler", "YARN" and "MESOS". Those are systems that will manage, schedule and coordinate the resource in a cluster.

A few words on YARN (standing, in a self-deprecating fashion, for "Yet Another Resource Negotiator"), it is a data operating system which will manage resources of the cluster. It is the core of the new generation of Hadoop systems and it has been used for the tests done in a cluster.

[4]Machine Learning library

Therefore, the biggest advantage of Spark is its integration giving the possibility to associate, seamlessly , multiple different processing models. One of the largest advantages of tight integration is the ability to build applications that seamlessly combine different processing models.

5.2.1 MapReduce Framework

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

The idea is to combine two main procedures:

1. a *map* procedure that will filter, sort or apply a function to each element of an ensemble;
2. a *reduce* procedure that will count, sum or reduce elements of an ensemble in one simple information.

It distributes the different processes by gathering the distributed clusters, computing the various tasks on the multiple clusters, managing the messages to communicate, transferring data between the various parts of the system and granting redundancy and fault tolerance.

The MapReduce framework is inspired by the map and reduce functions used in functional programming. However, the real idea behind the MapReduce framework is to provide scalability and fault tolerance capability. Indeed, it can be seen as defining functions and to perform them on each distributed computers (workers).

As previously said, MapReduce is a framework for processing problems with large data sets using a large number of computers, clusters or even a grid [5]. Processes can be run on data, structured (in a data base) or unstructured (any file system). Of course, as communication could be a bottleneck in term of performance and as data could be geographically dispersed, data are processed on or near the storage assets to reduce the transmission costs (remember the locality influence).

[5]An ensemble of clusters are in a grid if the clusters are shared across geographically and administratively distributed systems, and use heterogeneous hardware

As shown previously in Figure 5.1, a distributed network is made up of a master and workers, from this more detailed view, the MapReduce framework is also detailed. Indeed, it can be split into three steps including the two previously shown (map and reduce):

1. the *map step*: the map procedure is applied to the distributed data on each worker and the output is written to a temporary storage. The master orchestrates the distribution of the processes and their respective data. Note that for redundant copies of distributed data, only one is processed;
2. the *Shuffle step*: the output data produced by the map step are redistributed such that all data are located on the workers that will perform the reduce step;
3. the *reduce step*: each group of output data are processed by their respective workers, per key, in a distributed fashion.

Those steps are assumed to be run in a sequence, each step starts only after the previous step is finished. However, they can be interleaved as long as the final result is not affected.

The main advantage of MapReduce is that it supports distributed computing. The map operations can be done independently on multiple workers in parallel even though, in practice, parallelization is constrained by the number of independent data or even the number of CPUs per workers. Identically, the reduce operations can be distributed provided that all outputs of the map operations that are gather at the same time to their respective reducer (reduce worker) or that the reduce operation is associative. At first sight, MapReduce may seem more restrictive (distribution of the tasks to the workers, coordination, schedule...) than common sequential algorithms and therefore less efficient but is in fact and in practice, well-suited to manage large data sets. For example, a large server farm can use MapReduce to sort a petabyte of data in only a few hours. Another advantage is that it is able to recovers from partial failure of clusters. Indeed, if a map or reduce operation fails, the work can be re-schedule by the master (if the input data is still available of course).

Note that Spark is a framework that uses the same concepts as the MapReduce framework.

Now the distributed objects of Spark are to be defined and described, it is done in next Section 5.2.2 while more detail information about how the system works is described in Section 5.2.3.

5.2.2 Programming with Resilient Distributed Data

Spark's core has defined a particular object: the resilient distributed data set (RDD). It is a distributed collection of elements. In Spark, everything is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result. Under the hood, Spark automatically distributes the data contained in RDDs across your cluster and parallelizes the operations you perform on them. There are two ways to create RDDs. The first one is to parallelize an existing collection in your driver program. The second one is to refer to a data set in an external storage system. RDDs are split into multiple pieces called *partitions*, which may be computed on different workers.Then, the idea is to perform Map and Reduce operations which depend on passing in functions that are used by Spark to compute data on RDDs.

However, one of the major constraints using RDDs is that it is not allowed to access two RDDs at the same time (e.g. to perform operations in one depending on another). In order to use information stocked in two (or more) RDDs, they have to be aggregated into one bigger RDD. This constraint is partially discussed in Paragraph 5.2.2 about working with key-value RDDs. Another important thing to have in mind is that the order of data contain in a file or a collection will not be guaranteed (see Section A.2).

The aim here is not to define how to use Spark but to describe the essential properties of the RDDs and particularities of the system. See Section 5.2.3 for more details about the systems internal design. The main properties are defined in the following paragraphs.

Immutable Objects In functional and oriented-object programming, an immutable object is an object whose state cannot be modified after it is created. The use of immutable objects is mandatory. Using mutable objects could lead to error as mutable objects are not directly updated and can be in an "illegal" state. The idea is to always use immutable objects whose states are always defined. Of course, RDDs are immutable objects. This is well-suited for distributed computing as it requires synchronization which may be difficult to perform with mutable objects. With the immutability property, when a function returns an object, its state is clearly defined. Therefore, there is no risk that a process may have change the object at the same time, this ensures that object are always in a determined state.

Transformations/Actions Two types of operations can be done on RDDs.

The first one is called *transformation*. When a transformation is applied on an RDD, it returns a new RDD. As discussed in the next paragraph, Sparks uses the lazy computation, it means that the RDDs are computed only when an action is performed on them. Many transformations are element-wise; that is, they work on one element at a time; but this is not true for all transformations.

As an example of transformation, filtering an RDD is a transformation: it creates a new RDD with only the elements that satisfy a predicate. There is also the union of two RDDs, joining two RDDs. In fact, any map operation is a transformation that apply a function to each element of a data set and returns a new RDD.

The second type of operation is called *action*. Actions are the operations that return a final value to the driver program or write data to an external storage system. Indeed, when an action is triggered, transformations of the required RDDs are evaluated as they are needed to actually compute the output.

As an example of actions, counting the number of elements in an RDD is an action. Any reduce operation is an action that aggregates all the elements of an RDD using some function and returns the final result to the driver program.

Lazy Computation Lazy evaluation (also named "call-by-need") means that when a transformation is called on an RDD (e.g. calling a map operation), the operation is not immediately performed. Instead, Spark internally stocks the different transformations that have been requested on the RDD. Therefore, all created RDDs can be seen as a sequence of transformation used to create new RDDs. This concept is named "lineage information".

Indeed, by using the lazy evaluation, Spark can reduce the number of passes it has to take over data by grouping operations together. Therefore, Spark can schedule more easily, be more efficient. Note that there is no real benefit to write a single complex map instead of chaining together many simple operations. It gives users the possibility to develop their applications with smaller and more manageable operations.

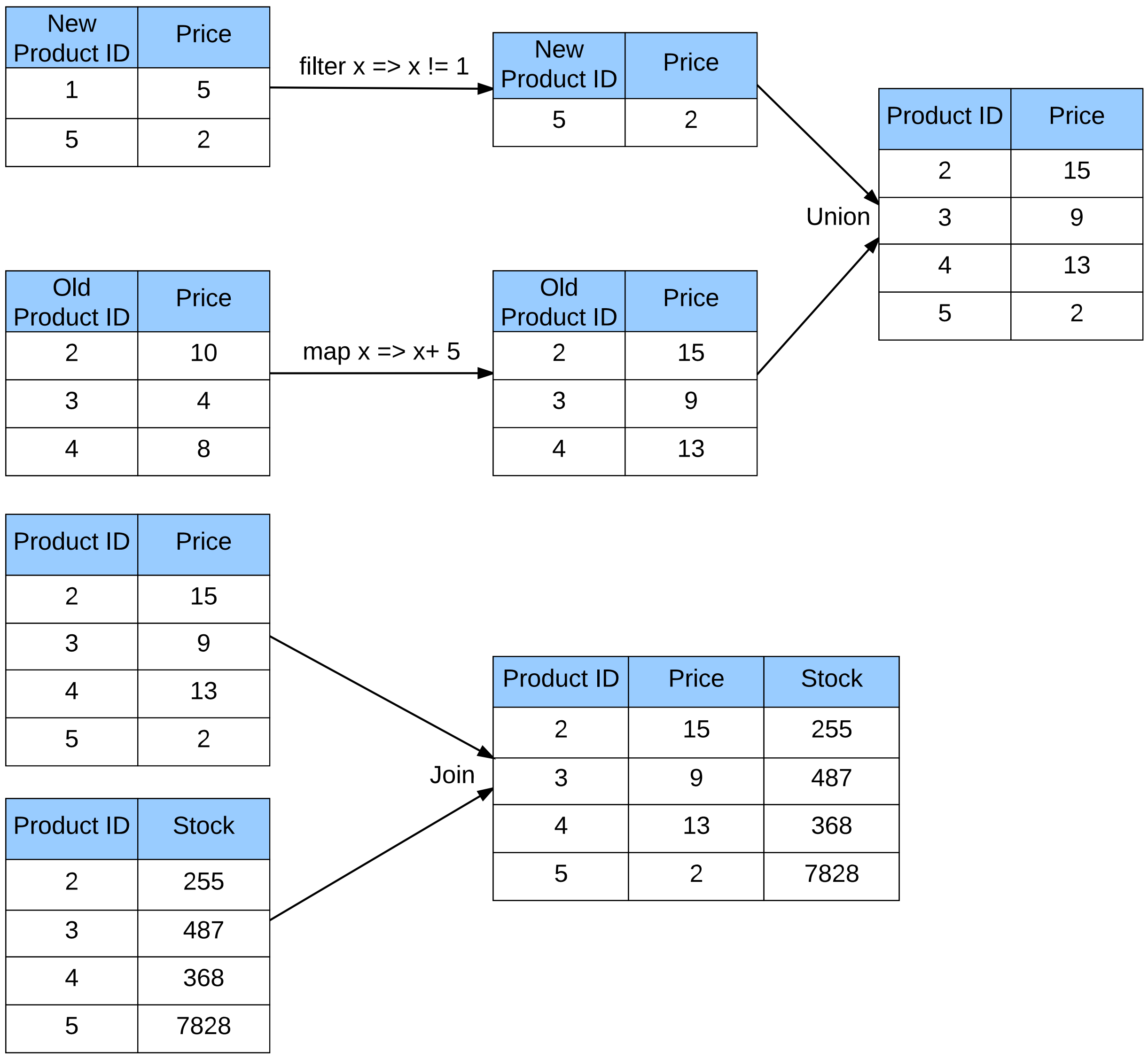
Caching As said earlier, Sparks RDDs are lazily evaluated. Of course, the same RDD may be used multiple times and Spark would re-compute the RDD and all of its dependencies whenever an actions is called on it. This may be computationally expensive for iterative algorithms which need to look at the data at each iteration.

For example, imagine that you perform a transformation on an RDD of integers to build a new RDD composed of the square of each element of the first RDD. Then, you count the number of element in the new RDD (i.e. perform an action) and then you compute the sum of each element (i.e. also an action) on the new RDD. In this example, the new RDD will be re-compute twice for each action, therefore, it may be interesting to cache it before performing the actions. Note that Spark can cache the data in different at many levels (on the java heap, serialized or not, in memory, on disk...).

It may seem odd not to persist RDD by default but the idea behind is that, for big data set, if an RDD will not be re-used, it would be a huge waste of storage space while, instead, Spark could go through the data once and re-compute the result. This ability to re-compute RDDs is in fact why the are said *resilient*.

Working with Key-value Pair Key-value RDDs are very often used, they are also named pair RDDs. As its name suggests, a pair RDD is composed of keys matching values. As said earlier, pair RDDs are used to perform aggregate operations by key or aggregate values of multiple RDDs by key. They give new possibilities of operations on single RDDs or on two RDDs. For a single pair RDD, it is possible to group all the data with the same key or counting elements by key. For multiple pair RDDs, it is possible to group together two RDDs. The join is a common transformation that takes two RDDs and groups them by key, it will be often used. There is also the transformation ReduceByKey which is often used too. It aggregates data separately for each key using a function and return an RDD.

Figure 5.4 represents simple RDD operations.



New Product ID	Price
1	5
5	2

Old Product ID	Price
2	10
3	4
4	8

New Product ID	Price
5	2

Old Product ID	Price
2	15
3	9
4	13

Product ID	Price
2	15
3	9
4	13
5	2

Product ID	Price
2	15
3	9
4	13
5	2

Product ID	Stock
2	255
3	487
4	368
5	7828

Product ID	Price	Stock
2	15	255
3	9	487
4	13	368
5	2	7828

FIGURE 5.4: First example shows two pair RDDs; they correspond to old and new product ids and their respective prices. The RDD with the old product is filtered to filter too old product and the prices of the new products are increased of 5 units. Finally, an union operation gather the two RDDs into one. The second example corresponds to a simple join between two pair RDDs. The first one is composed of product ids and their prices while the second one is composed of product ids and their stocks. Finally, the resulting RDD is the aggregation by key of both RDDs.

5.2.3 Systems Internal Design

In order to understand the way Spark works, we have to take a look at a lower level, in its internal design.

Spark Architecture As said before, distributed architecture and also Spark is composed of one master and workers. Figure 5.5 illustrates Spark architecture. The *driver program* is the one running on the master node which coordinates the workers. It has to communicate with all workers which are called *executors*. Master and workers are defined in their own Java process. The whole architecture composed of a master and workers is called *application*. Applications are run on a set of computers thanks to a *cluster manager*. Multiple cluster managers exist like YARN presented previously and Mesos (both are open-source). Spark has also its own built-in cluster manager which can also be used on Amazon EC2 clusters (see Section B.2 about Amazon).

Worker Node is the same as Slave Node. Executor is the program that is launched on the Worker when a *job* starts execution.

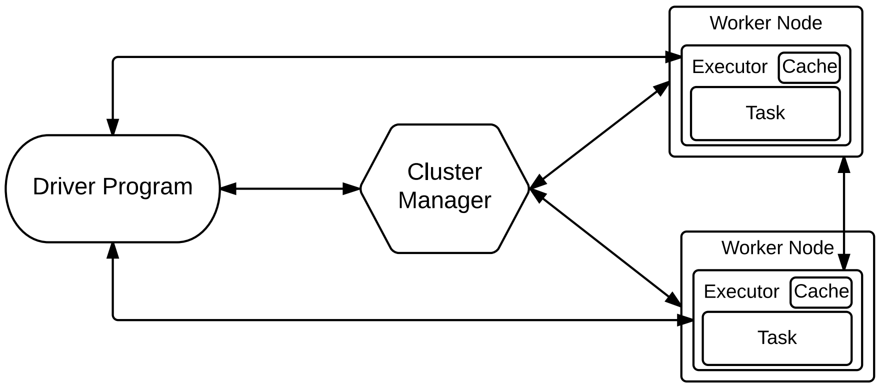


FIGURE 5.5: Spark architecture

Each elements performs particular tasks and each of them will be presented to give an overview of the functioning of the different parts of the Spark architecture.

The **driver program** is the main element of the architecture. It will create RDDs, performs transformations/actions, schedule the different tasks translated from the user code for the executors (see next paragraph about Spark execution).

Executors are entities that run individual tasks of a Spark job. They are responsible for two main roles:

1. run tasks from an application, return results to the driver;
2. provide in-memory storage in case of caching.

Moreover, if an executor failed to perform tasks, those tasks are re-scheduled to another executor. This way, Spark is fault tolerant. One of the main difference in Spark architecture and former architectures is that RDDs are cached directly inside of executors. This gives the possibility for executors to run tasks only on cached data.

Cluster managers are particular entities that are used by Spark and are responsible for starting executor processes. They also allocate resources across applications.

The three main components of the Spark architecture have been defined. Now the way operations are done in Spark is described in the next paragraph.

Spark Execution The idea of Spark is that it will try to distributed as much as possible the operations done on RDDs. More precisely, when operations have to be executed on RDDs, Spark will transform the lineage information (i.e. the sequence of transformations) into a real execution plan by merging those operations into tasks. In fact, the lineage information is a directed acyclic graph composed of RDD objects that will be computed later once an action occurs.

For example, if a map and a filter operations are done on an RDD, Spark will only retain a DAG like this: Filtered RDD $\leftarrow$ Mapped RDD $\leftarrow$ RDD.

When an action like count is triggered, Spark has to compute all partitions of the considered RDD and transfers it to the driver program. The directed acyclic graph will be used to plan what operations have to be done, going recursively from the last entry (Filtered RDD) through the transformed RDDs (Mapped RDD in our case) to the actual RDD containing the data (RDD).

Spark schedules the needed operations into a physical plan called *stages* (e.g. multiple map and/or filter operations). In our very simple example, the scheduler may schedule a computation stage for each RDD; each stage has multiple tasks for each partition of the considered RDD. Of course, stages are executed in the reverse order, from the initial RDD to the required one. Each Task launched by a physical stage are independent and will all do the same thing but in a specific partition of data.

Note that whenever an action is called, one or multiple stages may be created. This set of stages is referred as a *job*.

In order to summarize a Spark execution, here are the different phases:

1. From a set of operations, a directed acyclic graph of RDDs is constructed;
2. Whenever an action is called, the directed acyclic graph is used to create a physical execution plan defining a job which is a set of stages which are themselves a set of tasks;
3. Each tasks is schedule and executed on a cluster on a partition of an RDD;
4. The action is over when the last stage of the job is computed.

Spark goes through those four phases for each execution trying to distribute as much as possible the operations.

Note that Spark differentiate itself from "older" Hadoop MapReduce framework on two very high level differences: Spark stores data in-memory whereas Hadoop has to stores data on disk. Secondly, Spark's data storage model, resilient distributed data sets, uses a clever way of guaranteeing fault tolerance that minimizes network I/O. This may give better performance as data do not have to go through disks but it requires also more RAM on each workers to store data.

This section gave an idea on how Spark works at a certain point of abstraction. Now, as in this thesis, graph computation is performed, the GraphX component of Spark is described in next Section 5.3.

5.3 GraphX

GraphX is a library for manipulating graphs (e.g., a social networks friend graph) and performing graph-parallel computations. Like Spark Streaming and Spark SQL, GraphX extends the Spark RDD API, allowing us to create a directed graph with arbitrary properties attached to each vertex and edge. Indeed, a graph is build from an RDD of vertices and an RDD of edges.

GraphX also provides various operators for manipulating graphs (e.g., sub-graph and mapVertices) and a library of common graph algorithms (e.g., PageRank and triangle counting). Note that in GraphX, graphs are directed, edges are always directed defined with a source and a destination (meaning that you have to create two directed edges to create an undirected graph or transform some conditions in the message passing algorithms to send message in both direction). In this thesis, multiple operations have been used. For example, the *sub-graph* operation which filter graphs by applying a filter on the RDD of edge and RDD of vertices. Another one is the *aggregateMessages* operation which sends messages to vertices through the edges conditionally and returns an updated RDD of vertices. The most used operation done on graph is *Pregel*, it is used to create the message passing algorithms developed in this thesis.

5.3.1 Pregel

By definition, graphs are recursive structural models. Indeed, the vertices properties depend on their neighbors' properties which depend themselves on their neighbors' properties. Consequently, the major graph algorithms are iterative and compute at each step the vertices properties until satisfying a fixed condition. Multiple graph-parallel abstractions have been developed in order to express those iterative algorithms. GraphX proposes a variant of the Google Pregel API.

In GraphX, the Pregel algorithm is a bulk-synchronous parallel messaging abstraction constrained to the topology of the graph. It is also a fault-tolerant framework for the distributed computation of graph algorithms over multiple machines. It is like the MapReduce re-thought for graph operations.

The Pregel algorithm performs a series of *super steps* in which vertices receive an aggregation of their inbound messages from the previous super step, computes the new properties of each vertex, and then sends messages to neighboring vertices in the next super step. Messages are computed in parallel as a function of the edge triplet[6] and the message computation has access to both source and destination vertex attributes. Vertices that do not receive a message are not taken into account within a super step. When the Pregel algorithm is finished, it returns the final graph (if no message are remaining or if it has reached a fixed maximum number of iterations defined by the user).

This being said, it will be easier to understand Pregel by looking at how it is constructed and with a simple example.

Here is the type signature of the Pregel operator

```
Function pregel (A) :
    (initialMsg: A,
    maxIter: Int = Int.MaxValue,
    activeDir: EdgeDirection = EdgeDirection.Either)
    (vprog: (VertexId, VD, A) ⇒ VD,
    sendMsg: EdgeTriplet[VD, ED] ⇒ Iterator[(VertexId, A)],
    mergeMsg: (A, A) ⇒ A)
    return Graph[VD, ED]
```

where the elements are defined as:

- initialMsg: an initial message each vertex receives on the first iteration;
- maxIter: a number of maximum iteration may be set up (optional);
- activeDir: the direction defining which edges will be active in the next super step;
- vprog: the function that changes the vertices in function of the received message;
- sendMsg: the function that takes every active edges and sends messages to the targeted vertex;
- mergeMsg: the function that aggregates the received messages.

[6]An edge triplet is composed of an edge and its two linked vertices defined as source and destination.

The idea of direction is not easy to understand; the active direction defines which edges incident to a vertex that has received a message in the previous super step will be active (i.e. accessible) in the function to send message. For example, if this is EdgeDirection.Out (resp. EdgeDirection.In), only out-edges (resp. in-edges) of vertices that received a message in the previous round will run. EdgeDirection.Either is the default argument which will run sendMsg on all edges of the vertices that have received a message previously. If EdgeDirection.Both is used, sendMsg will only run on edges where both vertices received a message.

Note that it is a little bit different from the Pregel version of Google where the messages are send from the vertices and a priory in function of the received messages. Instead, here, messages are only accessible for changing the vertices and are no more available when sending the message. It is tricky as you have to keep the message as an attribute of the vertex to use it afterwards to send a message. This has been done to have better performance but may seem less intuitive.

Pregel is used in the inspired GHS algorithm detailed (see Section 6.1.2.3), for the Markov tree construction (see Section 6.1.3) and the inference algorithm (see Section 6.2) described in next Part II about distributed algorithms. Therefore, all functions in pseudo-code are in fact using Pregel to send messages, they are denoted by the three components: "Message reception" (vprog), "Sending a message" (sendMsg) and "Aggregation of the received messages" (mergeMsg). Note that the initial message is mandatory but is often used only to launch the algorithm and make all edges active without any real meaning (this is why this step is skipped in the following example).

A simple and didactic example is given to understand the concept of Pregel.

Imagine a group of people, for some strange reasons, one of them becomes a zombie. Fortunately, in this group, someone knows how to heal a zombie and when he heals one, he also teaches him of to become himself a healer. The rest of the group is just normal people.

A zombie will try to bite all normal people that are in his field of view represented by the edges; when a zombie is the source of an edge and a normal person is the destination it means that the zombie will bite him. A zombie can bite multiple normal people and do not bite healer. A healer will try to heal all zombies connected to him (healers have special power to detect zombies) and those healed zombies will become healer themselves.

This scenario can be represented with a graph and Pregel (initialized with meaningless initial message, no maximum iteration and the direction is set to ”Either”). Figure 5.6 shows the scenario using Pregel. Pregel is detailed more formally in the legend at the bottom of the figure.

In the first super step, all the edges are actives (as all vertices have received the initial message) and only the first zombie will bite three normal people. Thus four people are now zombies.

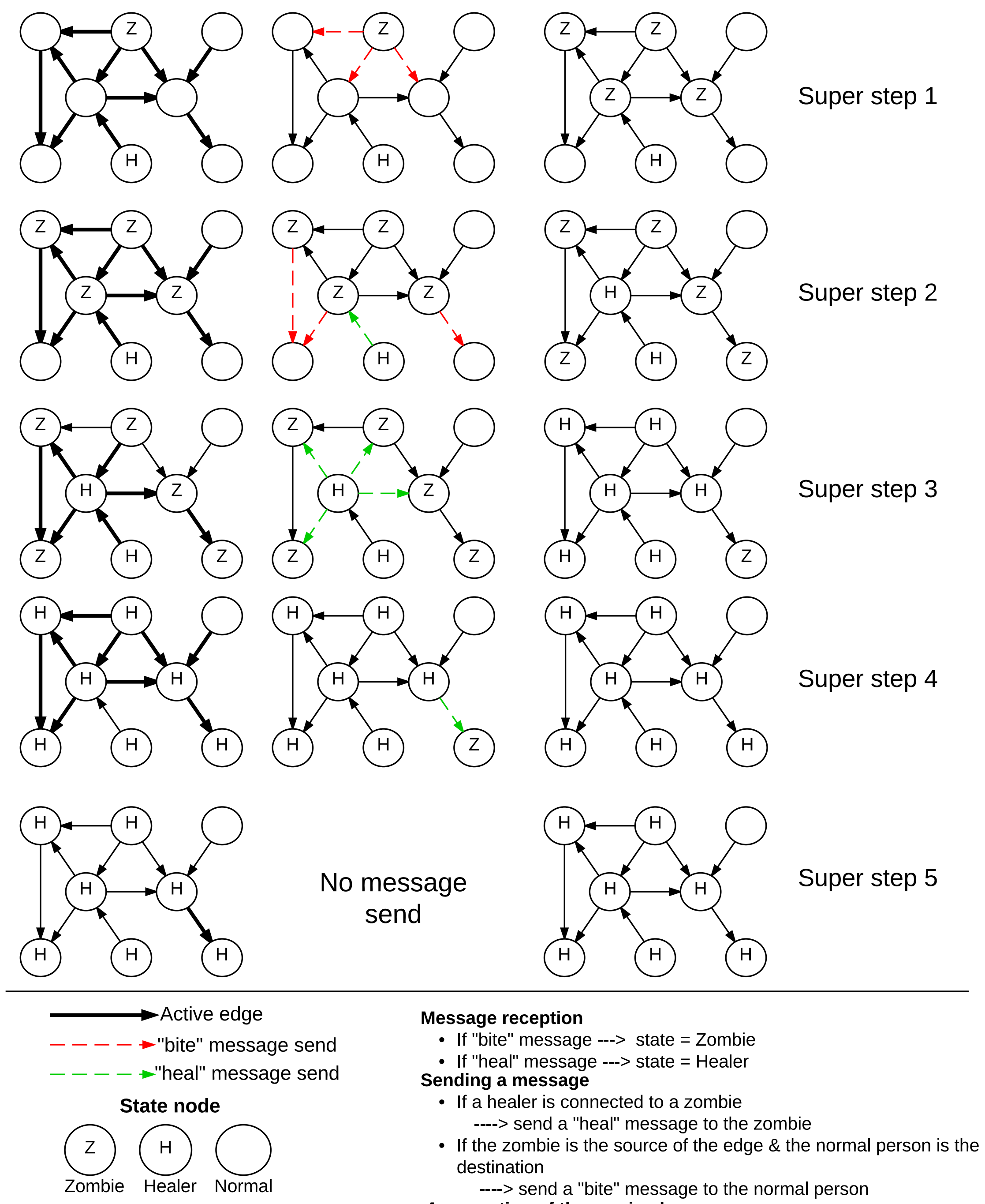
In the second super step, all the edges of vertices that have received a message are active. The zombies will continue to bite people but in this case the healer will heal one zombie. Four messages are send. A zombie is healed and become a healer and two people are ”zombified” (note that one normal person is biten by two zombies, the ”bite” message are aggregated into only one message). Therefore, there are two healers and five zombies.

In the third super step, only some edges are active. The new healer will heal four zombies. There are now six healers and only one zombie.

In the fourth super step, only some edges are active. one of the new healers of the previous super step will heal one zombie. At this point, there is no more zombie.

In the fifth and final super step, the only active edge is one connected to the last zombie which is now healed. There is no more zombie and thus everyone is safe now, the healers do not have any zombie to heal. Thus, it meands that Pregel is finished.

Now that Spark and GraphX have been described, it closes this chapter and this part. The next part will define the distributed algorithms developed in this thesis.



Message reception
- If "bite" message ---> state = Zombie
- If "heal" message ---> state = Healer

Sending a message
- If a healer is connected to a zombie
  ----> send a "heal" message to the zombie
- If the zombie is the source of the edge & the normal person is the destination
  ----> send a "bite" message to the normal person

Aggregation of the received messages
- If two "bite" messages ----> aggregated in one "bite" message
- If two "heal" messages ----> aggregated in one "heal" message

- No initial message
- No maximum number of iteration
- Stop when no message are send
- Direction is set to "Either"

FIGURE 5.6: Example of the use of Pregel.

Part II

Distributed Algorithms

Chapter 6

Markov Trees in a Distributed Environment

In this chapter, the different distributed techniques to learn Markov trees implemented in the thesis are presented. Like the previous parts, algorithms are described, sometimes illustrated, their complexity are evaluated and a pseudo-code is given to support the description.

6.1 Learning Trees

6.1.1 Chow-Liu Algorithm

In order to use the power of the graphical API graphX, the mutual information, computed through distributed objects (RDDs), is used to create a complete graph.

Indeed, samples of variables contained in files are directly used to create an RDD containing an array of observations [1]. Then this RDD is transformed to create a pair RDD containing variable labels as keys and samples of variable as values. From this RDD, Cartesian product and filter operations are applied on it to create a new pair RDD containing only pair of variable labels as keys and pair of samples as values. The mutual information is computed for each key from the pair of samples. Finally, this pair RDD

[1]See Section A.2 about representation issues.

is directly used to create the complete graph as keys correspond to two linked vertices (a source vertex and its destination) and values correspond to the mutual information which are the edges' weights.

Thus, the vertices of the graph are contained in a RDD and correspond to the variables. The edges are also contain in a RDD and their weights correspond to the mutual information. From this, to perform the Chow-Liu algorithm, we have to construct the minimum weight spanning tree from this graph. This is detailed in its own section 6.1.2 as it is an important part of this thesis.

6.1.2 Minimum Weight Spanning Tree

In order to create the minimum weight spanning tree, the previous techniques explained in Section 3.3.3 have been transformed to be performed in a distributed environment, partially or entirely. This is explained in the next sections 6.1.2.1 and 6.1.2.2. Of course, those algorithms have not been thought to be performed in a distributed environment but it is still an entrance door to understand how to create a minimum weight spanning tree. Moreover, a message passing algorithm [2] has been implemented too and is described in the section 6.1.2.3.

6.1.2.1 Kruskal, Prim and Boruvka Partially Distributed

As explained in previous Section 3.3.3, each of those centralized algorithms have a step which is well-suited to be performed in a distributed environment (or at least in parallel). Those algorithms are called *partially distributed* because after some operations (e.g. filter, join, reduceByKey) done on RDDs, the edges and vertices are collected into the local drive. Concerning Kruskal's algorithm, it is the "sorting phase": the edges are stored and sorted in a RDD then retrieved on the local drive to perform the rest of the tasks. Regarding Prim's and Boruvka's algorithm, it is the "find" operation: the distributed graph is filtered to only keep the required vertices and the "find" operation is performed on each of them. In the case of Prim, it would retrieve one edge on the local drive and add the new vertex in the tree. Concerning Boruvka's algorithm, it would retrieve a set of edges for each sub-tree on the local drive and then group each sub-trees thanks to the retrieved set of edges.

[2] Convenient to be performed in a distributed environment.

This gives quite simple algorithms only partially computed in a distributed environment and on the local drive. Their performance and scalability are tested in Section 9. Note that transferring object from its local drive to the distributed environment (and vice versa) add an extra cost of computation. Moreover, collected data such as the edges may be too large for the local drive memory and would create memory issues. For this reason, those methods could not be really scalable, they are only used to compare time performance with the message passing algorithm explained in Section 6.1.2.3.

6.1.2.2 Kruskal, Prim and Boruvka Fully Distributed

In order to perform those algorithms in a distributed environment, a lot of constraints are to be taken into account. For example, in Prim's algorithm, we have to join the vertices and the corresponding edges to search the minimum weight edge.

Hence, a lot of transformations are done on RDDs, Spark have to maintain the lineage information. For intern reasons in Spark, I have experienced stack overflows [3] when the number of variables becomes too large for Spark (around 300-500 variables). Indeed, long lineage causes a long/deep Java object tree (DAG of RDD objects), which needs to be serialized as part of the task creation. When serializing, the whole DAG needs to be traversed leading to the stack overflow error.

In order to solve this problem, performing actions would force Spark to actually compute the RDD and ”refresh the lineage information”. I tested this solution, it ”worked” but the computation times was so much increased that they were no more competitive with the inspired GHS algorithm explain in Section 6.1.2.3 [4].

To sum up, those algorithms, under the constraints of the distributed environment, require a lot of ”join” and ”union” operations and thus a lot of messages have to send between workers. Moreover, it is not well-suited for Spark for the reason explained before. In fact, initially, those algorithms were not designed [5] and, in the scope of my researches, do not perform very well with Spark.

[3]This error has been also experienced by others, see stack overflow caused by long lineage.

[4]Note that for smaller number of variables, those algorithms were good but still slower than the inspired GHS algorithm.

[5]Some twisted version of those algorithms have been developed to run on distributed environment but not enough and clear information about them has been found or the scientific papers were paying.

6.1.2.3 Gallager, Humblet and Spira Inspired Algorithm (GHS)

The GHS algorithm is a message passing algorithm to construct a minimum spanning tree introduced in 1983 [8]. It is particularly well-suited for the distributed environment and thus for Spark with its Pregel implementation. Note that the algorithm that has been implemented is inspired of the GHS algorithm but differs as the GHS makes the assumptions of asynchronous messages passing while Pregel forces to send messages in a synchronous way, super-steps per super-steps. Note that in the distributed environment things are more complex since there is no "entity" that knows the topography of the entire graph.

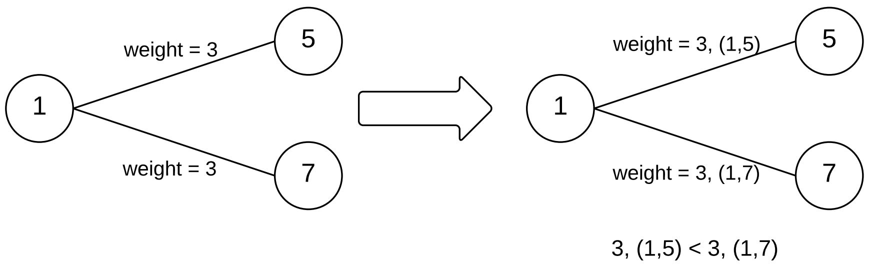
First, in order to describe the algorithm, four terms are to be defined:

- fragment: a sub-tree defined as a connected set of nodes and edges of the MWST, characterized by its id;
- outgoing edge: an edge connecting one fragment to another;
- frontier node: a node linking two different fragments used to propagate update messages in the fragment with the lowest id.
- node: a simple node of the graph representing a variable, characterized by its id. It also contains its fragment's id and the fragment's id of its previous connected fragment which will be useful to update the fragment id's of the nodes which belongs to the update fragment.

The algorithm is built on four parts:

1. the search for each node of their minimum weight outgoing edge;
2. the search in each fragment of the minimum weight previously selected outgoing edges;
3. the update of the fragment's id of the frontier nodes;
4. propagation of the fragment's id of the nodes in their respective fragment through their frontier nodes.

The first thing to do in order to create a unique minimum weight spanning tree is to define how to compare edges, especially edges with the same weight. To distinguish the weights' edges, we can use the concatenation of the neighbouring nodes' id.



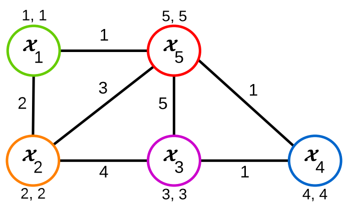
The second thing to do is to decide how to compare two fragments. When a fragment is connected to another through an outgoing edge, the updated fragment will be the one with the lower id.

From that, initialize the nodes and perform those four steps iteratively until all the nodes belong to a unique fragment. Note that the nodes contain its fragment's id and its previous connected fragment's id is used to differentiate frontier nodes and simple node to update their fragment's id.

Figure 6.1 illustrates a simple example of the inspired GHS algorithm.

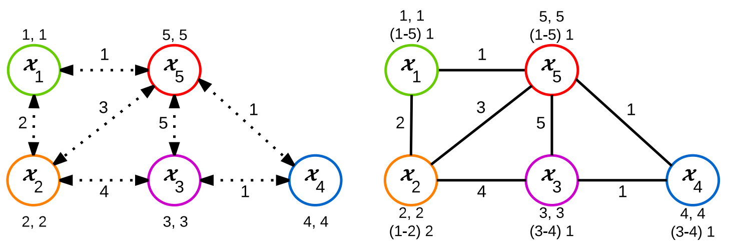
Complexity of the inspired GHS algorithm and its pseudo-code in Algorithm 9 are discussed in the next paragraph.

First call of the GHS function



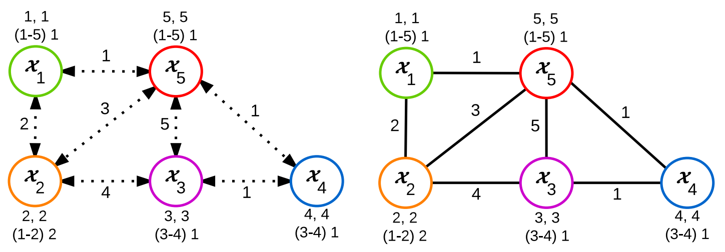
Initial fragment
{1}
{2}
{3}
{4}
{5}

Step 1



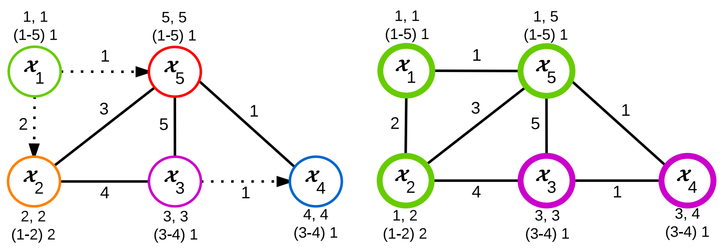
Fragment	Node	Minimum weight edge between fragments
{1}	1	1-5
{2}	2	1-2
{3}	3	3-4
{4}	4	3-4
{5}	5	1-5

Step 2



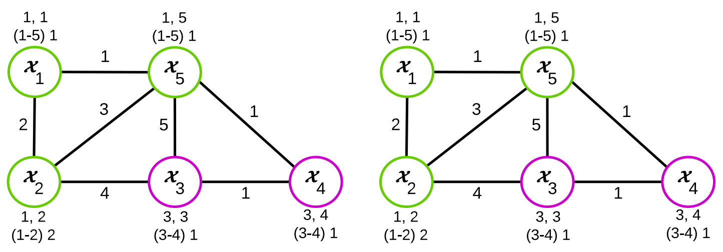
Fragment	Node	Minimum weight edge in fragments
{1}	1	1-5
{2}	2	1-2
{3}	3	3-4
{4}	4	3-4
{5}	5	1-5

Step 3



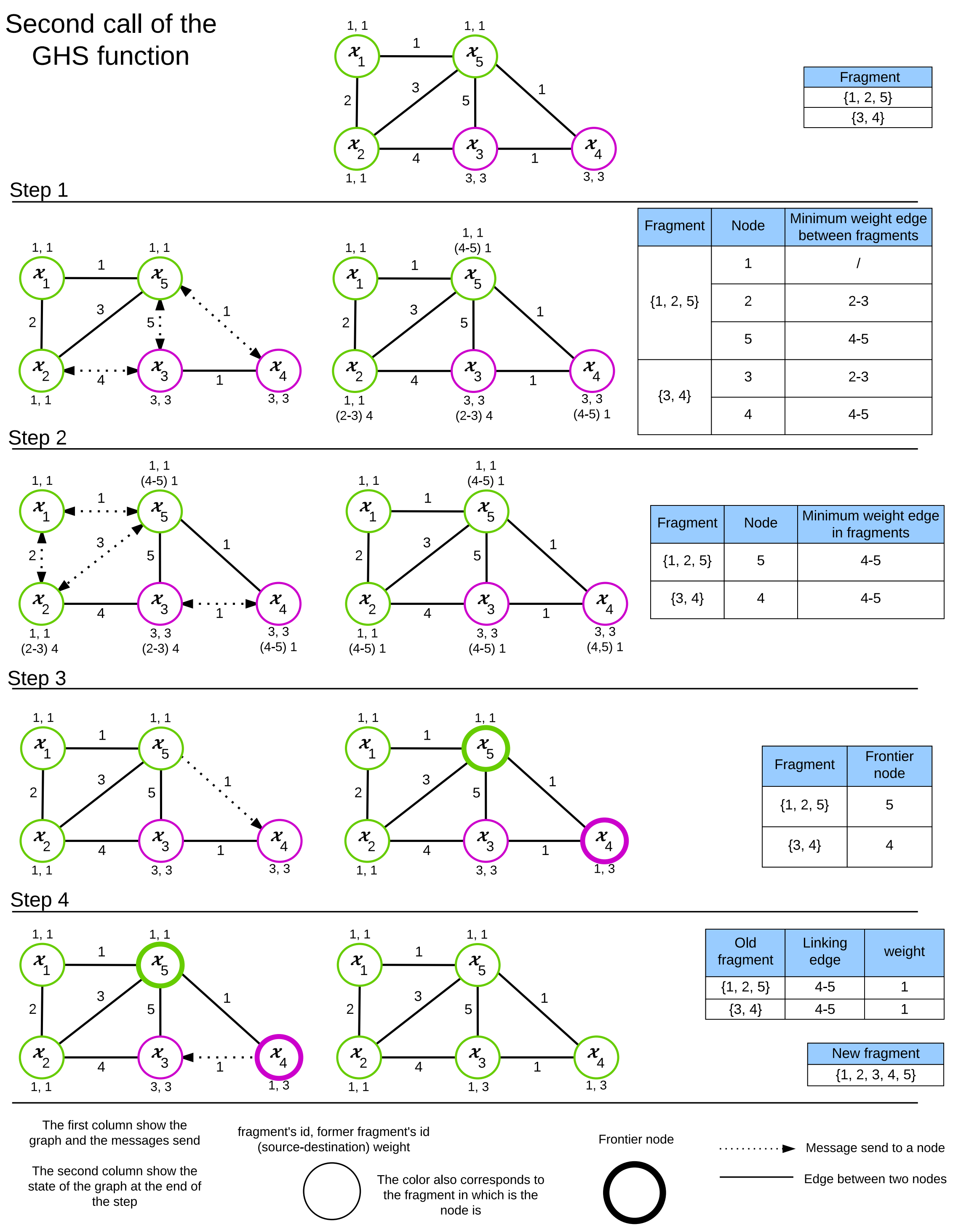
Fragment	Frontier node
{1}	1
{2}	2
{3}	3
{4}	4
{5}	5

Step 4



Old fragment	Linking edge	weight
{1}	1-5	1
{2}	1-2	2
{3}	3-4	1
{4}	3-4	1
{5}	1-5	1

New fragment
{1, 2, 5}
{3, 4}


Second call of the GHS function
Step 1
Step 2
Step 3
Step 4
The first column show the graph and the messages send
The second column show the state of the graph at the end of the step
fragment's id, former fragment's id (source-destination) weight
The color also corresponds to the fragment in which is the node is
Frontier node
Message send to a node
Edge between two nodes

Fragment
{1, 2, 5}
{3, 4}

Fragment	Node	Minimum weight edge between fragments
{1, 2, 5}	1	/
	2	2-3
	5	4-5
{3, 4}	3	2-3
	4	4-5

Fragment	Node	Minimum weight edge in fragments
{1, 2, 5}	5	4-5
{3, 4}	4	4-5

Fragment	Frontier node
{1, 2, 5}	5
{3, 4}	4

Old fragment	Linking edge	weight
{1, 2, 5}	4-5	1
{3, 4}	4-5	1

New fragment
{1, 2, 3, 4, 5}

FIGURE 6.1: Example of the GHS inspired algorithm. The iterative function is called twice, going through the 4 steps each. At each call, the chosen edges are kept but it is not shown on the example to save place.

Complexity and Algorithm

The complexity of the algorithm is expressed in terms of messages passed. The first step sends messages from nodes of different fragments, it requires E messages to be sent. Then the second step sends messages inside the fragments, it requires E messages to be sent. The third step computes the frontier nodes, it requires E messages . Finally, the fourth step updates the nodes of connected fragments with the lower id, it requires also E messages. Those four steps are done iteratively and for each pass, the number of sub-trees (i.e. groups of vertices) is reduced by a factor two. Therefore the complexity is $\mathcal{O}(4 \times E \log V) = \mathcal{O}(E \log V)$.

Algorithm 6.1.1: Inspired GHS (Gallager, Humblet and Spira)
Input : Symmetric matrix M of positive weights
Output: minimum Weight Spanning Tree

Initialize : a graph $\mathcal{T}$ where the vertices corresponds to the variables and the edges' weight are the mutual information between two variables

```
Function iterative_GHS ($\mathcal{T}$) :
    /* compute the minimum weight edges between two fragments          */
    $\mathcal{T}$ = minimum_weight_between_fragments ($\mathcal{T}$)
    /* compute the minimum edge in each fragment                       */
    $\mathcal{T}$ = minimum_weight_in_fragments ($\mathcal{T}$)
    if all edges are in the same fragment then
        return $\mathcal{T}$
    end if
    else
        /* update fragment's id of the nodes linking the fragments     */
        $\mathcal{T}$ = update_frontier_nodes ($\mathcal{T}$)
        /* update fragment's id of nodes in the updated fragment       */
        $\mathcal{T}$ = update_fragments ($\mathcal{T}$)
        iterative_GHS ($\mathcal{T}$)
    end if
end
```

```
Function minimum_weight_between_fragments ($\mathcal{T}$) :
    maxIter = 1
    Message reception :
        keep the minimum weight edge
    Sending a message :
        if outgoing edge then
            send to each other the edge weight their respective connected fragment's id
    Aggregation of the received messages :
        keep the message with the highest weight
```

```
Function minimum_weight_in_fragments ($\mathcal{T}$) :
    maxIter = 1
    Message reception:
        compare the message and the node
        keep the one with the highest weight
        if the nodes and the message have the same weight then
            keep the one with the lowest fragment's id
    Sending a message :
        if an edge links two nodes with the same fragment's id then
            send to each other their connected edge weight and connected fragment's id
    Aggregation of the received messages :
        keep the message with the highest weight
```

```
Function update_frontier_nodes ($\mathcal{T}$) :
    Message reception :
        update fragment's id by the message's id
    Sending a message :
        compare two connected frontier nodes
        send the higher fragment's id to the one with the lowest fragment's id
    Aggregation of the received messages :
        keep the message with the lowest fragment's id
```

```
Function update_fragments ($\mathcal{T}$) :
    Message reception :
        update the fragment's id by the message's id
    Sending a message :
        if edge links node1 with the same fragment's id than the last fragment's id of
        node2 and node2 has higher fragment's id then
            send to node1 the lowest fragment's id of the other one
    Aggregation of the received messages :
        keep the message with the lowest fragment's id
```

6.1.3 Markov Tree Set Up

In order to transform a tree into a Markov tree, we have to orient the edges outwards the root which is chosen randomly. It is done like a breadth-first search in a message passing fashion using Pregel. From the root, the edges are oriented downwards level per level, sending messages from an upper level to the next lower one.

From this Markov tree, we have to estimate the parameters. As explained before, the estimate chosen is the Bayesian estimation. It is computed like the mutual information with a Cartesian product but the filter operation only keeps the pair samples composed of a parent node and its respective children. Then the parameters are estimated and the new PGM corresponds to a Markov tree totally set up (to practise inference for example).

Complexity of setting up a Markov tree and its pseudo-code in Algorithm 14 are discussed in the next paragraph.

Complexity and Algorithm Here, a message is sent to each vertex except the root. Thus the complexity in terms of messages sent is $\mathcal{O}(V-1)$. Note that the complexity of learning the parameters of one variable is linear in function of the samples' size.

Algorithm 6.1.2: Markov tree construction

```
Input   : data set $D$, undirected tree $\mathcal{T}$ whose vertices are labeled by the variable set $\mathcal{X}$
Output: Markov tree
root = random vertex
breadth orientation downwards from the root
$\theta$ = learn_parameters($\mathcal{T}, D$)
return ($\mathcal{T}, \theta$)
```

6.2 Inference in Markov Trees

In order to perform inference on Markov trees in a distributed environment, the belief propagation, explained in Section 3.2.1 is well-suited as it is a message passing algorithm. Therefore, Pregel will be used to create two main functions, one to send λ messages and another to send π messages.

Before that, evidences are initialized and used to initialise λ & π of each variable $\mathcal{X}$ of the Markov tree. In addition of evidences, nodes have to contain the algorithm level (as Pregel do not offer this possibility and as it was not recommended to change Pregel in the source file and rebuild the library) to know if they can send message or not as the first phase requires to send messages per level.

From that, the first phase can begin. λ messages are send with Pregel. From the lowest level of the tree, the leaves send messages to intern nodes which will themselves send messages to intern node until the λ message reach the root (see earlier Figure 3.4).

When the first phase is finished, the second phase has to be initialised. Indeed, all nodes have to collect the $\lambda(\mathcal{X})$ of their children as it is needed to compute $\pi(\mathcal{X})$. This is simply done in a distributed fashion by using a function in GraphX which create a new set of vertices containing the parameters of their children (i.e. all vertices that are the destination of an edge).

Then, the second phase can be run. This is done exactly the same as explained earlier (see earlier Figure 3.5). π messages are send with Pregel. From the root to the leaves level by level, the π messages are propagated and the belief are computed.

Complexity of inference in a Markov tree and its pseudo-code in Algorithm 15 are discussed in the next paragraph.

Complexity and Algorithm The complexity here is proportional to the number of messages sent. In a tree of V vertices and $V-1$ edges, in the first phase, $V-1$ messages are sent. Then, the vertices have to collect the parameters of their children which require also to send $V-1$ messages. It is the same in the second phase. Thus, the belief propagation has a complexity of $\mathcal{O}(3 \times (V-1) \simeq \mathcal{O}(V-1)$ in term of communication.

Algorithm 6.2.1: Belief Propagation

```
Input  : Markov Tree and a set of evidence $\mathcal{E}$
Output: Inferred Markov Tree
Initialize:
/* when $\mathcal{X} \in \mathcal{E}$, the position of the 1 corresponds to the value given to
$\mathcal{X}$ thanks to the evidence                                         */
    • If $\mathcal{X}$ is a leaf and $\notin \mathcal{E} : \lambda(\mathcal{X}) = [1...1]$
    • If $\mathcal{X}$ is the root and $\notin \mathcal{E} : \pi(\mathcal{X}) = \mathbb{P}(\mathcal{X})$
    • Else if $\mathcal{X}$ is a node and $\in \mathcal{E} : \lambda(\mathcal{X}) = \pi(\mathcal{X}) = [001...0]$
/* sending lambda messages by level, from the leaves to the root            */
$\mathcal{T}$ = lambda_message($\mathcal{T}$)
/* All nodes collect the $\lambda(\mathcal{X})$ of their children                          */
$\mathcal{T}$ = collect_lambda_children($\mathcal{T}$)
/* sending pi messages from the root to the leaves and computation of
the belief                                                                  */
$\mathcal{T}$ = pi_message($\mathcal{T}$)
return $\mathcal{T}$
```

```
Function lambda_message ($\mathcal{T}$) :
  Message reception :
    if $\lambda(\mathcal{X})$ not initialised by an evidence then
      /* the message = all the multiplied $\lambda$ children messages */
      $\lambda(\mathcal{X}) = message$ where $\lambda(\mathcal{X}=x) = \prod_{C \in Child(\mathcal{X})} \lambda_C(\mathcal{X}=x)$
  Sending a message :
    foreach level, from the lowest to the highest (root) do
      foreach child $C$ of $\mathcal{X}$ do
        send to the unique parent: $\lambda_C(\mathcal{X}=x) = \sum_{child} \mathbb{P}(C=c|\mathcal{X}=x)\lambda(Y=y)$
  Aggregation of the received messages :
    multiply the $\lambda$ messages
```

```
Function pi_message ($\mathcal{T}$) :
  Message reception :          /* received from the only parent P of $\mathcal{X}$ */
    $\pi(\mathcal{X}=x) = \sum_{p} \mathbb{P}(\mathcal{X}=x|P=p)\pi_{\mathcal{X}}(P=p)$
    $\lambda(\mathcal{X})\pi(\mathcal{X})$          /* compute the belief */
  Sending a message :
    foreach level, from the highest (root) to the lowest do
      foreach unique parent $P$ do
        send to each child $\mathcal{X}$: $\pi_{\mathcal{X}}(P=p) = \pi(P=p) \prod_{U \in Child(P) \setminus \mathcal{X}} \lambda_U(P=p)$
  Aggregation of the received messages :
    keep the only message received from the parent
```

Chapter 7

Learning Mixture of Trees

In this chapter, the techniques to learn mixture of Markov tree are approached. Two main techniques are presented:

1. the mixture of Markov tree constructed with the Chow-Liu algorithm with bootstrapping (MCL-B).
2. the mixture created with the expectation-maximization concept (MT-EM).

The first approach is used to decrease the variance of the model while the second one is used to decrease the bias. Those two approaches have been studied in the thesis of François Schnitzler. I chose to implement the first one as it was one of the most accurate techniques while being one of the most computationally expensive (ideal to be transposed in distributed environment to study its performance). It was also chosen because it is a construction of m independent Markov trees which is well suited for parallel or distributed programming. The second one has been to have a technique to reduce the variance.

7.1 MCL-B algorithm

As said before, the MCL-B algorithm stands for "Mixture with Chow-Liu and Bootstrapping". Therefore each tree is constructed with the Chow-Liu algorithm with bootstrapped data.

As it is not allowed to construct an RDD containing multiple graphs each representing a Markov tree (i.e. RDD of RDDs), each Markov tree is contained in a simple array.

In order to perform bootstrap on RDD, the technique is quite the same as in a traditional way. A sequence of random numbers is generated and a map operation is done on the RDD in order to create bootstrapped samples.

Note that parameters are learned from the original learning set and the tree is build on the bootstrapped replicates.

Complexity of creating a mixture and its pseudo-code in Algorithm 14 are discussed in the next paragraph.

7.1.1 Complexity & Algorithm

Each tree will be computed sequentially, therefore, a mixture of m trees will be m times more complex. The complexity will be clearly determined by the Chow-Liu algorithm composed of the computation of the mutual information and the MWST algorithm (the inspired GHS in our case).

Algorithm 7.1.1: MCL-B

```
Input  : Data $D$, size of the mixture $m$
Output: Mixture of Markov Tree
Initialize: • $\boldsymbol{\mathcal{T}} = \emptyset$   • $\boldsymbol{\theta} = \emptyset$   • $\lambda = [1/m]_{1\times m}$
for $k \leftarrow 1$ to $m$ do
    $D' =$ bootstrap($D$)
    $\boldsymbol{\mathcal{T}}[k] =$ Chow-Liu($D'_k$)
    $\boldsymbol{\theta}[k] =$ learn_parameters($\boldsymbol{\mathcal{T}}[k], D$)
end for
return ($\boldsymbol{\mathcal{T}}, \boldsymbol{\theta}, \boldsymbol{\lambda}$)
```

7.2 MT-EM algorithm

The MT-EM stands for Mixture Tree Expectation-Maximization. The idea is determine the weight of each tree of a mixture to reduce the bias [1]. The weights are considered as the marginal probabilities $\mu_k = \mathbb{P}(Z = k)$ where Z is a hidden variable which selects one distribution $\mathbb{P}_{\mathcal{T}_k}(\boldsymbol{\mathcal{X}}) = \mathbb{P}(\boldsymbol{\mathcal{X}}|Z = k)$:

$$\mathbb{P}_{\mathcal{T}}(\boldsymbol{\mathcal{X}}) = \sum_{k=1}^{m} \mathbb{P}(Z = k)\mathbb{P}(\boldsymbol{\mathcal{X}}|Z = k) \tag{7.1}$$

By alternating between estimating a distribution of the hidden variable Z for each observation and optimizing both $\mathbb{P}(Z = k)$ and $\mathbb{P}(\boldsymbol{\mathcal{X}}|Z = k)$ the estimate of $\mathbb{P}(\boldsymbol{\mathcal{X}}, Z)$ is optimized. This technique is described in Algorithm 19 where $\gamma_k(i)$ can be interpreted as the probability that the hidden variable $Z = k$ for a certain observation $\mathbf{x}_{D_i}$ and according to the current estimate of $\mathbb{P}(\boldsymbol{\mathcal{X}}, Z)$. Therefore, for each observation and for each tree a weight $\gamma_k(i)$ will be associated. From those weighted observations per tree, a mixture of tree is created thanks to the Chow-Liu algorithm.

Unfortunately, due to performance issue, the distributed computation of the γ_k and the computation of new mixture each pass until convergence takes too much time and thus I have been unable to test it.

Besides the fact that some operations to compute the weights are relatively computationally expensive, I have been debugging for some times the MT-EM algorithm to make it runnable in an acceptable time. I found that the problem was caused by one bad distributed design choice. I have been regrouping all the elements in partitions of multiple RDDs and reduce them into only one partition which created a bottleneck and made the algorithm way too slow.

The pseudo-code of the MT-EM algorithm is still given in Algorithm 14.

Algorithm 7.2.1: MT-EM: CL mixture of Markov trees

```
Input  : Data $D$, size of the mixture $m$
Output: Mixture of Markov Tree : $(\boldsymbol{\mathcal{T}}, \boldsymbol{\theta}, \boldsymbol{\mu})$
Initialize: $(\boldsymbol{\mathcal{T}}, \boldsymbol{\theta}, \boldsymbol{\mu})$
repeat
    for $k \leftarrow 1$ to $m$ do
        for $i \leftarrow 1$ to $N$ do
            $\gamma_k(i) = \frac{\mu_k \mathbb{P}_{\boldsymbol{\mathcal{T}}[k]}(\mathbf{x}_{D_i})}{\sum_{k=1}^{m} \mu_k \mathbb{P}_{\boldsymbol{\mathcal{T}}[k]}(\mathbf{x}_{D_i})}$
        end for
    end for
    for $k \leftarrow 1$ to $m$ do
        $D'_k = (\mathbf{x}_{D_i}, \gamma_k(i))_{i=1}^{N}$
        $\boldsymbol{\mathcal{T}}[k] = \texttt{Chow-Liu}(D'_k)$
        $\boldsymbol{\theta}[k] = \texttt{learn\_parameters}(\boldsymbol{\mathcal{T}}[k], D'_k)$
        $\mu_k = \frac{\sum_{i=1}^{N} \gamma_k(i)}{N}$
    end for
until convergence
return $(\boldsymbol{\mathcal{T}}, \boldsymbol{\theta}, \boldsymbol{\mu})$
```

Part III

Experimental results

Chapter 8

Correctness

This chapter will focus on the accuracy of the Markov tree and the corresponding mixtures implemented in this thesis [1]. The idea is to measure the correctness of the resulting distribution $\mathbb{P}_{\mathcal{T}}$ thanks to a Monte-Carlo approximation of the Kullback-Leibler divergence with respect to the target distribution $\mathbb{P}$, given by :

$$\hat{D}_{KL}(\mathbb{P} \parallel \mathbb{P}_{\mathcal{T}}) = \frac{1}{N'} \sum_{x \simeq \mathbb{P}}^{N'} \log_2 \left(\frac{\mathbb{P}(\mathbf{x})}{\mathbb{P}_{\mathcal{T}}(\mathbf{x})} \right), \tag{8.1}$$

where N' corresponds to the number of observations (50000 in our case) in the test set.

The evaluation has been done using one target distribution: one synthetic DAG-200-5 (see Section A.1) meaning that the data are generated from a DAG with 200 variables and maximum 5 parents per variables. Samples from the same DAG have been used to perform the different evaluations of the divergence.

In order to compute the divergence, a learning and test set generated from this DAG have been used. As explained in the chapter about machine learning, the idea is to learn from the learning set, infer the probabilities of each observation from the test set and then compare those probabilities to the validation set which contains the probabilities computed over the target distribution.

From that, two tests have been made: the first one aims to characterized the divergence with an increasing number of samples thanks to fixed size mixture of Markov trees. The

[1]The source code is on github, see github.com/juliennix/TFE

second one aims to characterized the divergence with an increasing number of trees in a mixture while the number of samples is fixed. Note that, as Spark do not keep the order when transforming a file into an RDD, the indexes of each line of the test file and validation file have to be kept to compare comparable elements.

Figure 8.1 illustrates the first evaluation using a mixture of 5 Markov trees. Three different learning sets generated from the same DAG have been used. We can see that the mean of the divergence (red line) decreases from 100 samples to 4000 samples. It means that inferred probabilities of the observations in the test set are more close to probabilities of those observations determined by the real distribution (i.e. probabilities in the validation set). Then, it can be seen that the divergence increases. It may be caused by the fact that the model is overfitting meaning that the learned model fits too much the learning set, becoming less general and thus increases the error made on the test set.

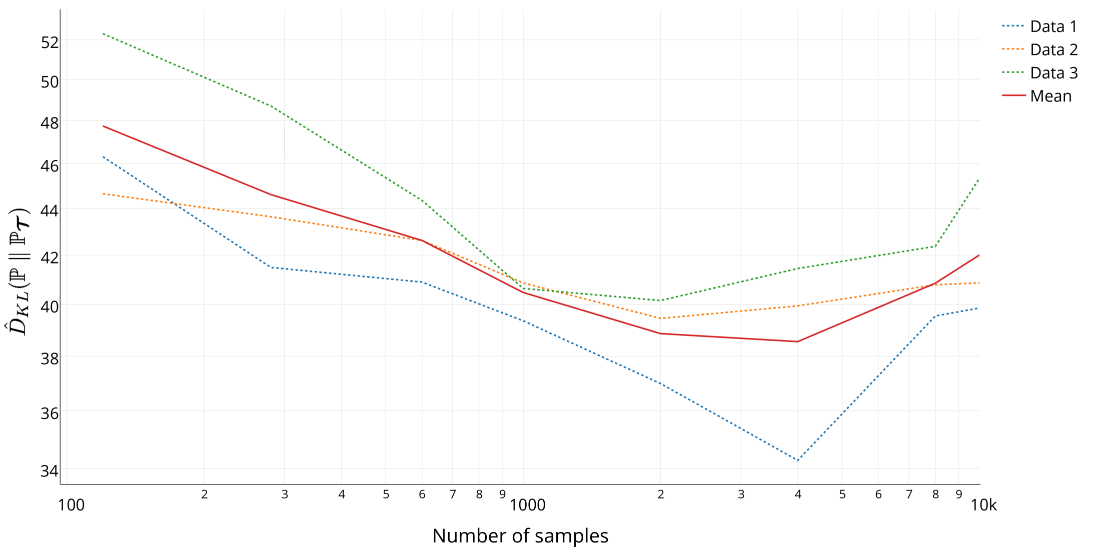


FIGURE 8.1: Illustration of the behavior of the divergence when the number of samples grows with a mixture of 5 trees

Figure 8.2 illustrates the impact of the number of tree in a mixture to model the target distribution. We can see that for small samples size (120 and 1000) the divergence decreases when the number of tree in the mixture increases and it seems to converge to $\simeq$ 44 for the 120 samples learning set and to $\simeq$ 40 for the other two learning sets. This shows that for small learning sets, bootstrapping has a great effect, decreasing the variance and hence, probabilities of the observations are more accurate. Regarding the larger learning set of 10000 samples, the divergence increases a little bit. Indeed, the variance is low because a large number of samples is used. However, the bias of the bootstrapped replicates may increase. This may be caused by the fact that randomization induced by the bootstrapping (as some objects do not appear and other appear several times) will create wrong dependencies between variables and thus induce error.

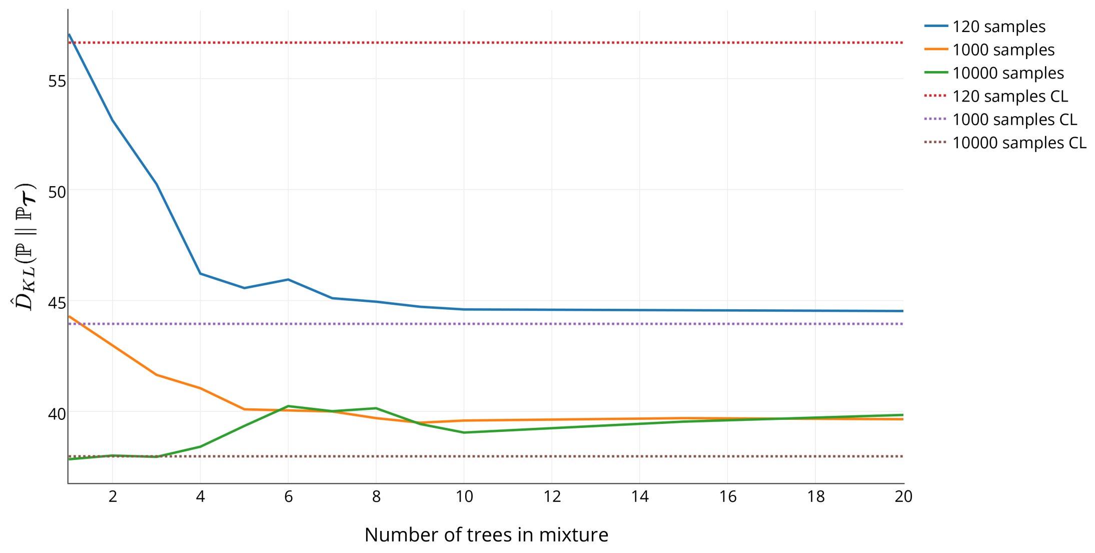


Figure 8.2: Illustration of the behavior of the divergence with fixed size samples when the number of trees in a mixture grows compared with the divergence of one tree build with the Chow-Liu algorithm

Now that the correctness of the model has been evaluated, time performance and scalability of the methods will be presented.

Chapter 9

Time Performance & Scalability

This chapter is about the performance and the scalability of the different methods presented in this thesis. Firstly, the results of the different tests produced on my computer (4 cores 2.53 GHz, 4Go RAM) will be discussed. I have been using mostly the Spark shell during my development phase on my computer to implement and test my algorithms. Secondly, the results produced on clusters will be discussed and compared to the results of my machine.

9.1 Local Tests

The first local test has been performed to compare the computation time of the different MWST algorithms. The idea here is to look if the GHS method is competitive with the traditional methods done partially on the local drive. Of course, methods done on local drive will be faster as they are dependent of constraints related to distributed computing but they will not be adequate for a very large number of variables. Indeed, the traditional algorithms are only partially distributed meaning that when the edges or the vertices will be collected on the local drive to perform the "centralized" part of those algorithms, odds are that the local drive will not be able to deal with so much data and will throw a memory error. Anyway, comparing them could give an overview on how the GHS algorithm behaves and if it runs in an acceptable time.

Note that the tests performed here are done with a learning set of 500 samples.

Figure 9.1 shows the results. We can see that the GHS algorithm is a little bit longer but still run in a reasonable time. GHS algorithm takes more time as said before because of the scheduling, the coordination,..., and because it runs on my computer which is not well-suited for distributed computing. All these constraints slow the algorithm but it can be used to treat much more data than the 3 others. Here, we can say that the GHS algorithm is competitive in comparison with the three algorithms done partially on the local drive and on the cluster.

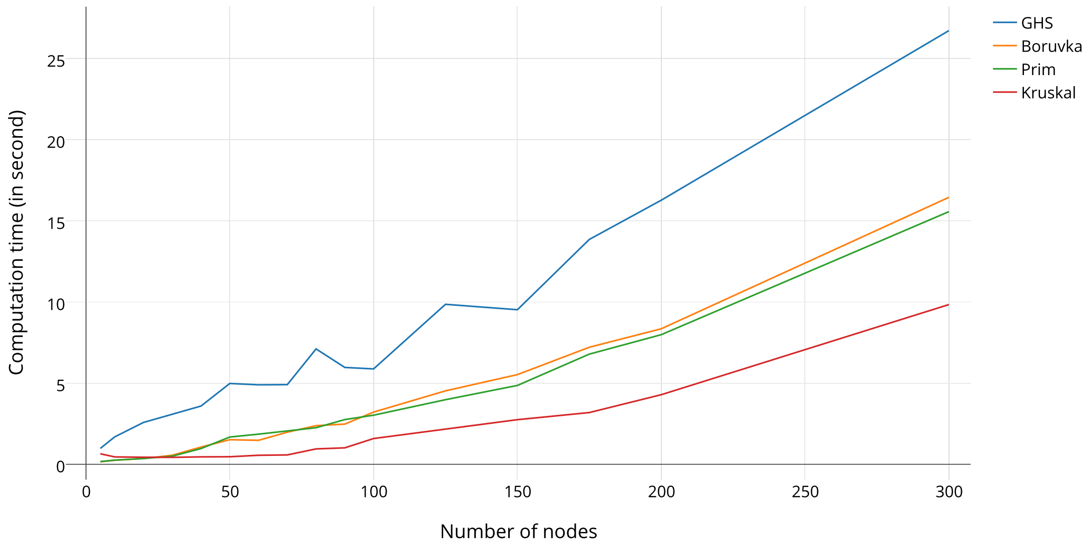


FIGURE 9.1: Comparison of the computation time for the different MWST algorithms depending on the number of variables (nodes)

Figure 9.2 illustrates the evolution of the computation time depending on the number of cores used. We can see that the computation time is slightly reduced. It has to be noted that the difference between 3 core and 4 cores in terms of CPU resource is not of 1 core as the computer have to keep CPU resources for other tasks. Thus, the difference between 3 and 4 cores are about a few percentage of CPU usage. Moreover, computation times are mean of 10 runs of the tested algorithms, as in fact, the results were fluctuating quite a lot (sometimes twice as much, I even had a peak of ten times the usual computation time). This shows that the methods are scaling but not proportionally of the number of cores used.

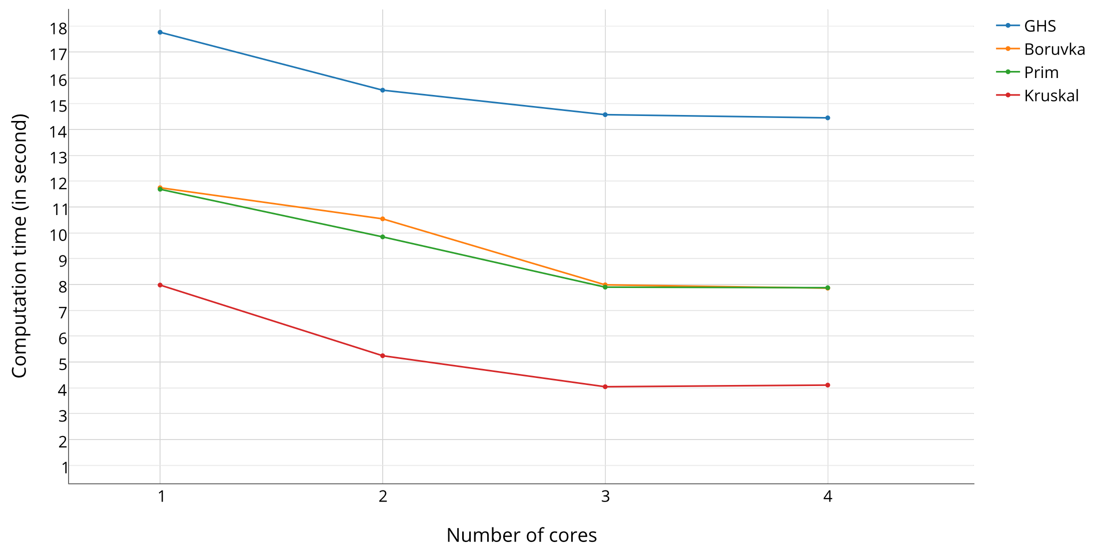


FIGURE 9.2: Comparison of the different MWST algorithms in function of the number of cores used

9.2 Cluster Tests

This section will present and discuss the different results computed on the provided cluster.

The tests are done on a cluster provided generously by the company Cray ©via a "sponsored account" with the company DataFellas[1]. This cluster is composed of 48 compute nodes, 96 processors, 1536 cores, 6TB of RAM and 206TB of storage [2]. Moreover, the cluster is not dedicated thus the results may fluctuate. This is why results are in fact an arithmetic mean of a set of results of multiple tests. Note that Spark runs on YARN.

Table 9.1 denotes the computation times of the different implemented methods in function of the number of variables. Those tests have been done with 20 executors, 5 cores and 5Gb of RAM each, the local drive has 1Gb of RAM. Each variable has 500 samples.

It can be seen that the computation time of the mutual information, GHS and the creation of Markov trees increase as the number of variables grows. The inference increases too but have not succeed analyzing 10000 variables. Indeed, the inference algorithm for 10000 variables throws a "Garbage Collector overhead limit exceeded"

[1]Company run by Andy Petrella (@noootsab) and Xavier Tordoir (@xtordoir).
[2] Web page of the Cray configuration

meaning that the garbage collector is taking an excessive amount of time and recovers very little memory.

Note that the previous partially algorithms would not have been able to deal with 1000 variables without triggering a OOM[3] error when retrieving the edges on the local drive. This shows the utility of developing distributed solution. Besides the amount of data treated, the time performance are not particularly ”fast”.

Table 9.1: Computation time of different methods in function of the number of variables

Method \ Number of Variables	10	100	1000	10000
Mutual Information	1"	2"	42"	5' 14"
GHS	17"	28"	1'27"	10' 30"
Markov Tree Creation	7"	12"	45"	8' 5"
Inference	8"	1' 2"	4' 32"	*

Table 9.2 shows the computation time of the different implemented methods in function of the number of executors with 10 Gb of RAM each and with a fixed learning set containing 1000 variables of 500 samples.

Results show that the studied methods scale with the size of the provided hardware but not in a proportional way. Indeed, the computation time of the studied methods decreases but not proportionally to the number of executors. It may seem weird as most of them (the creation of Markov trees for examples) are just using the Pregel methods to perform operation. One first explanation would be that more executors would increase the scheduling time.

Therefore, from those results, I tried to optimize the intern parameters of Spark and the application itself. It is rather complicated mostly because the optimization can be done through multiple ways (see Section 10). In order to improve the presented results, I tried to tune the parallelism level by:

[3]Out Of Memory

Table 9.2: Computation time of different methods in function of the number of executors

Method \ Number of executors	1	10	50	100
Mutual Information	2' 10"	1' 32"	1' 5"	45"
GHS	1' 40"	1' 25"	1' 14"	1' 2"
Markov Tree Creation	1' 58"	1' 27"	1' 10"	1' 7"

1. changing the number of partitions for the initial RDD created from the learning set;
2. changing the number of blocks per file in the HDFS;
3. changing the number of executors by controlling the number of cores used per executor.

For the inference, as I have seen irregularities from the garbage collector information, I have try to tune the memory by removing some cache operation, changing the cache option to serialize objects, increasing the number of memory for each executors, decreasing the size of a blocks file in HDFS. Unfortunately, none of those attempts has been really successful and it is even more complicated because they are particularly time consuming to test as the Spark tool has to be re-launched for each test to change its parameters (from five to ten minutes I would say). The strange thing is that even when no cache operation is performed, it throws a error after a while without a clear error message, just saying that Spark has been shutdown with DAG scheduling error but, in fact, this is certainly related to a GC error.

As I have used Pregel implemented for Spark, I must admit that I was expecting faster and scalable results. I have noticed in the Spark UI that when the number of variables increases, Pregel is more and more time consuming and for two main reasons: the first one is that scheduling increases when the number of executors increases. Thus, it slows all the Pregel iterations and there is a lot of small operations per iteration. The second one is that the garbage collector has more and more objects to remove for Pregel as each iteration will create message objects and new vertices objects.

I have also find other weird processes management related to internal functioning of Spark: when the level of parallelism is increased, the different tasks given to each executor may not be equal. For the computation of the mutual information, few executors (around 6%) receive around 95% of the total input and of course perform much more tasks which creates a bottleneck. I do not know what are the intern causes of this phenomenon as the initialized partitions of the input RDD are uniformly distributed (this may be caused by a bad repartitioning of the initial RDD after doing a Cartesian product). Anyway, the bad distribution of the tasks for each executor gives clues on why the different implementation do not scale greatly.

Unfortunately, the different ways to increase time performance and scalability of the application did not solve the main bottleneck which is the non-fairly tasks distribution for each executors. This lack of performance is difficult to characterize as it may come from multiple sources and would require to test each transformation and action to see how the RDDs are actually computed, how they are actually repartitioned by Spark.

As explained in Section B.2, I have had some difficulties using Spark. (see also on the web where few scientists have exactly similar problems using Spark [4]). Hence, I have few remarks about testing HPC applications and debugging them with Spark.

First, it is true that Spark has an elegant documentation which gives the feeling to program locally (map/reduce/reduceByKey...). Unfortunately, when an application is launched, the different processes done in it are rather opaque and the source of a problem may be difficult to find; this is why Spark provides a user interface to see the state of the different jobs, their stages and the different tasks with some metrics (like scheduling time, GC time and serialization time). This is a great way to debug, I did not used it at the beginning on my computer but in fact, it is not possible to really understand how your application runs without it.

However, I have to say that the lazy computation makes the understanding of the progress of an application complicated and unintuitive. Indeed, it is used for efficiency purposes but makes debugging and finding bottleneck in a program difficult as it groups

[4]For example, a Doctor in computer sciences (see here its CV) with few years of experience in distributed computing (with other people working in the same laboratory) have had some difficulties using Spark and wrote it down on the web

transformations and schedules them internally. In order to test correctly, it is mandatory to trigger an action on a RDD to materialized it. It would be great to have a debug mode in which we could simply see the transformations or even the DAG of transformations and force computations of those transformed RDDs with few metrics about them directly.

Another great improvement for debugging would simply be to have more information about RDDs in the Spark UI. For example, the number of partitions of an RDD and what is in those partitions. This may give a great way to have an overview on the distribution and how the distribution is really done by Spark.

A last thing about testing and debugging is that there is plenty way of tuning Spark and your application. Chapter 10 describes this aspect of Spark. Indeed, even if you have the impression that you are programming on your laptop, you are not. To implement great and fast applications, attention must be paid on serialization, caching, memory management, hardware provision... There is also some settings that may be great to set and not used with their default values, e.g. buffers and heap size, the number of executors, the number of partitions for RDDs... and as there is no systematic way to set all the parameters for a specific problem of an application, you have to test cleverly those parameters choosing them wisely.

The next Part will be the final one. Its first chapter is focused on how an application can be optimized and of course the one of this thesis. It also describes some alternatives to create mixture. Finally, the second chapter is the conclusion of this thesis.

Part IV

Discussion

Chapter 10

Optimization

In this chapter, multiple ways to improve the use of Spark or the implementation itself are described. First, the ways to improve the use of Spark are discussed.

10.1 Spark Tuning

This section will try to provide clues on how to tune a Spark application. Basically, Spark has been designed so that default configuration works directly in an efficient way, "off the shelf". However, the user is free to tune some configurations of Spark and is forced to if performance are needed.

Level of Parallelism As said before, RDDs are immutable and split into a set of partitions containing a part of the data. Of course, the number of partition for an RDD will determine how it will be divided.

Hence, when Spark schedules and runs tasks, a single task is created for each partition. This task will be processed by default on one single core in the cluster. Basically, Spark determines the degree of parallelism and this is adequate for many use cases.

Of course, performance will be affected by the level of parallelism. In order to improve performance, it may not be too high because it would create small overhead associated with each partition and those may become an issue. Indeed, with too much partitions, a lot of small tasks will be computed but will actually do nothing and slow down the process. The degree of parallelism may not be too small as, of course, if you have 500 cores to perform a stage with only 50 tasks, many of the core will idle most of the time. A good practice is to set the number of partitions at an integer multiple of the number of executors.

The number of parallelism has been tested and provided slightly better results for the computation of the mutual information but it should be benchmarked in order to find the best degree of parallelism for a specific problem instance.

Changing the size of a block of a file in a HDFS is also a way to increase parallelism. The size of blocks has been decrease to 1 Mb to perform tests, it increases the parallelism and gives better results. [1]

Serialization One way to increase the performance is to change the way object are serialized.

Indeed, when a lot of data is transferred over the network or simply when those data are put on the disk drive, Spark has to serialize objects into a binary format.

Initially, the Java built-in serializer was used, then, as different classes where composed of collection and of others complex objects, another serialization library has been chosen: Kryo. Indeed, Kryo is a fast, compact and efficient object graph serialization framework for Java and is easy to use. Note that it cannot serialize all types of objects. As multiple objects are constructed with Map collections which is a pointer-based data structure, shifting to Kryo also provided small benefit in terms of time performance and memory usage.

Memory Management Memory management is very important as large data are processed. Spark uses memory in different approaches.

Inside of each executor, memory is used for RDD storage, in shuffle and aggregation operations and for the user code.

Concerning RDD storage, it is possible to cache an RDD. When an RDD is cached, its partitions would be stored in memory buffers. There is a limit on the volume that can

[1] It is difficult without a real benchmark with multiple tests on multiple applications to say how much it has increased the performance but I would say that it gives around 20% better results in terms of time computation.

be cached and when the limit is reached, older partitions are dropped. Some big RDDs which are used multiple times, like the MWST or the tree itself, are cached and also intermediate RDDs in order to avoid recomputing them. Note that caching has been done at its default level without serializing objects, in memory, nothing on disk.

Concerning the shuffle operations, intermediate buffers are created in order to store the resulting shuffle data. Those buffers are not the actual buffer for the results of an aggregation but additional ones that store intermediate results. It is possible to limit the size of those buffers but they are not suspected to be the cause of the lack of performance.

Regarding the user code, memory usage is important when user function requires a relatively big amount of memory (e.g. for big lists or arrays). The user code has access to a limited part of the JVM heap memory as it is used by Spark to store the RDDs and the shuffle storage. This may not be the cause of the lack of performance but it is always great to verify the amount of data needed for user functions to ensure that no potential problem of memory on executors may be triggered.

The garbage collector is another source of performance loss, especially with large file and low RAM resources or simply if too much RDDs are cached in memory.

Therefore, a particular attention must be paid to memory management. Managing memory is more complex as multiple sources of potential errors have to be taken into account.

Hardware Of course, the performance of an application is dependent on the hardware. The main parameters that have to be taken into account are the amount of memory and cores given to each executor, the number of executors and the size of local storage disks. Therefore, the hardware and its architecture is an important factor of performance.

Tuning a Spark application requires to test multiple configurations. It is quite complex as the configuration is function of the four basic parameters explained just before and finding the right needed resources for an application may require to test a relatively great amount of possibilities. The best way to test is to use the Spark UI and watch what are the stages that are slow and why (deserialization, scheduling, garbage collect, input size, shuffles...)

10.2 Implementation improvements

In order to improve the implementation, different ideas may be tried. First, dealing with mutual information near from zero could be very great. Indeed, it would create less dense graph and, by using some methods to build MWST on less dense (i.e. at a certain level of sparsity) graphs proposed in the PhD thesis of Meila [3]. The constructed models could also be better as they would not encode too small dependencies between variables.

Another way to build a mixture can also be an option. In this thesis, the MCL-B algorithm has been chosen as it was one of the most accurate but costly. This was a basic choice based on creating accurate models but it requires to recompute the whole mutual information on a bootstrapped replicated which is a very computationally expensive methods.

Other techniques based on edges re-sampling could have been a great choice as the computation of the mutual information is done only once. For example, the methods CES (Cluster-based Edge Sampling) and the RES (Random Edge Sampling) proposed in the thesis of François Schnitzler would be another great choice.

Finally, Spark provides a lot of built-in functions and advanced programming techniques, just to cite them: working on a per-partition basis, using broadcasting variables, defining custom partitioners, using accumulators; with more experience, optimizing the code of certain methods with more appropriate operations could be great.

Chapter 11

Conclusion

This thesis has explored the fields of distributed computing, machine learning, probabilistic graphical models... and has shown some algorithms to create mixture of Markov trees in a distributed environment.

The first idea was to use the Chow-Liu algorithm and randomize it to construct Markov tree and then mixtures of them which would be used to perform better density estimation.

Before even starting to use Spark, I have learned how to use the programming language Scala and its particularities (object-oriented and functional programming). Then I installed Spark, configured it and tried some simple examples, doing the Spark tutorial... Only after this introduction phase, I began to implement the application of the thesis.

The first step to implement the Chow-Liu was to define a way to compute mutual information in a distributed environment using the Spark tool. From the mutual information, a graph using the GraphX library was built instead of using a matrix containing the mutual information. Then, the next step was to find a technique to build a minimum weight spanning tree from this graph in a distributed environment. At first, some traditional MWST algorithms have been tried, being partially distributed, they were the base to understand how to implement the same principle that would have been transposed in a distributed environment. Then, those algorithms have been transformed to be fully distributed but were not really thought for this kind of environment. Finally, a message passing algorithm inspired from the GHS algorithm and using Pregel has been implemented, being the reference algorithm to build a MWST in this thesis.

When the Chow-Liu algorithm was completed, the algorithms to build a single Markov tree and then compute its parameters were implemented to run on a distributed environment. Pregel was used once again.

Then, the way to randomize the Chow-Liu algorithm was the "Perturb and Combine" framework. Its principle was to bootstrap the learning set and learning Markov trees structure on it while the parameters were learned on the original learning set. This technique was chosen to build mixtures of Markov tree.

An algorithm to perform inference on Markov tree was implemented using Pregel too.

Building a mixture with such methods is well-suited for distributed programming as each tree is learned independently. Mixtures of models were used to perform density estimation and the method has been validated and evaluated giving great results. Their time performance and scalability were also tested and were not very fast but could handle much more variable than a traditional centralized system. Optimization for increasing performance is required.

Distributed computing is a totally different way to program and compute. It seems a great way to deal with tones of data but much more elements to design programs are to be taken into account. There is also a lot of optimizing processes depending on the application that is very different from centralized programming; how data are distributed, how tasks will be split into workers and are they fully used or not, how are the tasks scheduled, is there congestion into the network... From there, much more metrics have to be considered. Thus, the optimization of a distributed environment and of an application (or even its debugging) is a more complex task which require to have a great overview on multiple aspects of computer engineering.

Therefore, is distributed computing a bad think ? No, but it is just initially more complex and require experience to be fully understood.

Concerning the creation of mixture of Markov trees in a distributed environment, it can be said that they are well-suited for distributed programming, they are easy to constructed and give the possibility to encode complex probabilistic distributions.

All that aside, according to me, it is still not always really clear how Spark works internally. This is one characteristic of distributed systems, the distribution is hidden for the user and the programmer of the application. This is like a black box that can be used seamlessly. I also had a lot of difficulties to find other information than the one provided by "Spark" itself which are always describing Spark for its particular uses. Multiple slide presentations presenting internal functioning of Spark are present on the web but are not self-sufficient according to me. I also thing that improving the user interface and debugging tools while explaining clearly the way of implementing correctly a Spark application[1] is very important, this could give real pleasure to build applications without always asking yourself where the problem could be or how to improve performance, felling like you are stabbing in the dark.

I am still quite mitigated about the recent Spark tool, more precisely, for high performance computing on large graph with its Pregel implementation. Spark and its Pregel implementation requires a lot of computation resources and especially RAM to run efficiently.

Moreover, GraphX did not seem very popular next to the other components of Spark but it is still quite in its early stage, thus it is comprehensible. A new version of Pregel with asynchronous message is apparently in development.

Spark is a growing technology, built to deal with enormous amount of data. Of course, only the future could tell how it will evolve but one thing is sure, new techniques of programming and dedicated hardware will be developed in order to always raise the power of computation, pushing the boundaries of computation.

[1]For example, it would be great to directly give a per partition approach of an application in the tutorial of Spark.

Appendix A

Experimental Data

A.1 Data Source

In order to perform experiments, data sets were needed. The data sets used have been created and used in the PhD. thesis of François. Indeed, the data sets correspond to randomly generated DAG structures [1]. As said before, a PGM is composed of a graph structure and its parameters, both were randomly generated and specify a distribution. Those kinds of model are often called *synthetic distributions*.

He provided learning sets, test sets and validation sets. Learning and test sets were generated from DAG structures. They are denoted by DAG-n-k where n is the number variables contained in the DAG and where k is the upper bound of parents a variable can have. Hence, by constraining the number of parents and as the number of parameters grows exponentially with the number of parents, it was possible to store them in memory.

A.2 Data Representation

Data representation and also the way those data are stored may seem anecdotal but may, in fact, lead to implementation difficulties and bad performances in term of computation time and memory usage. Two main problems can be drawn: how to represent my data and how to store them.

[1]Directed Acyclic Graph.

For the first problem, sometimes, the chosen representation may seem great at the beginning of a particular project and then, for some reasons, become source of trouble. In this thesis, we have to store observations of variables. Two simple ways to represent those data could be either that each line corresponds to a group of observations of all the studies variables or that each line corresponds to the label of a variable and all its observations (i.e. one sample). Note that the variable samples have to be labelled in the second representation; indeed, as the order of the samples is not guaranteed, it will not be possible after creating the new RDD to know which sample correspond to which variable. Note that the same principle can be applied for the test and validation sets used in Section 8 as each observation (i.e. line in the test file) of the test set has to be compared with the corresponding probability (i.e. line in the validation file) in the validation set.

I had chosen the second representation (with data sets simply generated with the random number generator from the Scala library) and the provided data sets used the first representation. This is kind of problematic because Spark reads file by lines and creates the RDDs from them, the representation leads me to design algorithms according to the representation and the Spark constraints. For example, the way mutual information is computed (Cartesian product, filter...) would have been different with the first representation (hypothetically as I have not implemented it: a reduce function to compute the co-occurrences and occurrences and a map to compute the mutual information).

Therefore, I had to transform the provided data into the second representation to use them with my algorithms. Of course, it increases the computation time and the memory usage.

For the second problem, the way data are stored (i.e. the file format) may be a cause of bad performance and especially when petabyte of data are to be stored. So far, the data used to create RDDs comes from a native collection or regular files, but odds are that the loaded data do not even fit on a single machine (and the data output of your application neither).

Therefore, it is mandatory to see how to deal with this issue. I will not go into much details but Sparks deals with wide range of input and output sources. It can manage a wide formats range: from text (unstructured), JSON wich (semi-structured; most libraries require one record per line), SequenceFiles (structured; used for key-value data), ADAM (columnar file format that allows efficient parallel/distributed access to genome) or even CSV.

In this thesis, text files are used and distributed in a HDFS. As said before, when a single text file is load to create an RDD, each input line becomes an element in the RDD (thus remind that representation has its importance). To load/save data from/to a distributed file storage system like HDFS, Spark has simple functions which manage those operations. Note that Spark can also use compressed format file.

With this section, some problems were highlighted just to invite programmers not to skipped them.

Appendix B

Personal Experience and General Information

B.1 The Programming Language: Scala

When a new language comes out, two reactions are possible: Either "Yeah ! new stuff" or "Oh, no! A new one, again". The reaction mainly depends on the programmer but anyway, the aim here is to figure out if this relatively new language (2003) is well-suited for the distributed programming.

As its name suggests, Scala stands for scalable language. The language is so named because it was designed to grow with the demands of its users. Scala can be applied to a wide range of programming tasks, from writing small scripts to building large systems [9]. Scala runs on the Java platform and operates with in a seamless way. Scala also mixes object-oriented and functional programming concepts.

Learning Scala was interesting, it is a nice language and quite easy to learn. However, at the beginning I had some problems in order to read code example and still a little bit now with more experience (even though I have the feeling that I have greatly improve my Scala skills since the beginning). This is because Scala has a very concise programming style; people creates blocks of code and it is difficult sometimes to fully understand what the code is supposed to do (even more without comments of course). Anyway, if someone asks me to introduce him to functional programming language, I would certainly recommend Scala.

B.2 The Distributed Environment and Spark

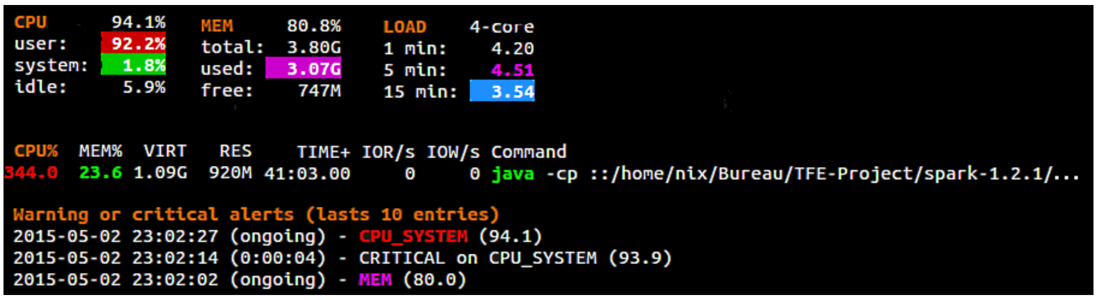
Distributed Computing Field and its accessibility In order to perform the tests of this thesis on a real distributed environment, multiple operations have been done and some basic notions have to be understood. The first operation is, of course, installing Spark. This part is quite well documented and ”easy” to access. Spark is still quite recent but the documentation is good at the beginning of the learning process. There is a tutorial, a good API documentation, even a book which has been made because people where ”complaining” about the lack of documentation. I had the feeling that for more complex issues or information about how does Sparks works internally, there is a lack of documentation. I have found a lot of blogs where people talk about some tests they have done with Spark or some mailing lists but nothing really formal. I think that one thing that bothered me is that there is lot of possibility with Spark, a rich API but in order to code efficiently, you have to understand how Spark really works, not use it as a black box as said in the conclusion and for that, there is not a lot of documentation. Anyway, Spark is a growing tools which will be more and more developed and documented.

Spark and its Use Spark has a great API, difficult to understand at first but which become more pleasant as you get familiar with. Spark can be simulated and thus simulates a distributed environment on a single machine using the multiple cores of it. This is easy to run, test applications directly in shell and it is stable enough. There is also a Spark user interface (SparkUI) which gives essential information about what Spark is going: jobs, stages, metrics like the computation time, the memory usage, the scheduling time... This is well done and easy to understand but could give more debugging tools.

Of course, as the computer simulating Spark may be used for a lot of other tasks, the results in terms of computation times may fluctuate, perturbed by the fact that not enough RAM or CPU resources are available.

According to me, the results fluctuates a little too much but maybe my computer is getting old. The same operation may take 50 ms one time and 1 second the next time. Moreover, I have experienced multiple shutdowns of the Spark tool without any messages. Therefore, it was impossible to understand what and why it happened. I have not found any specific explanation about those shutdowns but some others people have experienced the same thing.

Anyway, when some tests are running, they use the number of cores allocated as explained in Section 9. Sometimes, it may drain all the resources of your computer, thus, be careful when tuning the parameters of Spark (especially if other important tasks are running). Here is a screenshot showing the use of the CPU and RAM of my computer (4 cores and 4Go RAM):



```
CPU       94.1%   MEM      80.8%   LOAD     4-core
user:     92.2%   total:   3.80G   1 min:     4.20
system:    1.8%   used:    3.07G   5 min:     4.51
idle:      5.9%   free:     747M   15 min:    3.54

CPU%  MEM%  VIRT   RES     TIME+ IOR/s IOW/s Command
344.0  23.6 1.09G  920M 41:03.00     0     0 java -cp ::/home/nix/Bureau/TFE-Project/spark-1.2.1/...

Warning or critical alerts (lasts 10 entries)
2015-05-02 23:02:27 (ongoing) - CPU_SYSTEM (94.1)
2015-05-02 23:02:14 (0:00:04) - CRITICAL on CPU_SYSTEM (93.9)
2015-05-02 23:02:02 (ongoing) - MEM (80.0)
```

It is clear that the four cores are used reaching the 350% of CPU usage for Spark only (400% being the maximum). A big part of my RAM is also used, it reaches 23.6 % here. Depending on the way Spark has been configured, the amount of allocated RAM can be increased too (once again, be careful).

There is also another thing that I found not very easy is debugging, sometimes the errors are difficult to read referring to intern elements of Spark (like the DAG scheduler) and "googling" the error did not help me to understand what were those errors referring to.

Amazon and the EC2 Cluster Amazon offers a large panel of web services through its department names Amazon Web Services (AWS). It provides a large set of global compute, storage, database, analytics, application, and deployment services that help organizations to move faster, lower IT costs, and scale applications.

AWS is the biggest provider of cloud computing services for now, it is followed by Google, Microsoft of even IBM but to a lesser extent (AWS has more than a third of the world market share while the others share the rest). It must be said that the overall market is growing rapidly. The firm estimates the annual growth at around 50%.

This being said, we can jump into the description about what AWS offer in terms of high-performance computing (HPC) applications. The creation of new clusters for HPC is relatively recent (July 2010), and Amazon has deployed those new clusters name EC2.

From that, multiple pricing policies have been created in function of the required resources [1]. The idea is that a consumer does not have to think too much about what hardware to buy, neither how to configure them and above all, the consumer can adjust the resources that he needs. He also pays depending on what he uses per hour or can subscribe to a contract for one or more years. The cost can rise from \$0.013/hour for the minimum configuration to \$5/hour for the maximum one[2]. There is also free trial proposed to test the services, giving access to limited resources of course. Note that multiple instances of those clusters can be launched to run independently multiple applications.

This gives an idea of what are the offers for HPC but there are also many other services provided by AWS, many more applications are using the S3 storage solution for example.

[1]See an introduction video here: introduction to EC2 and HPC video here introduction to hpc

[2]The minimum corresponds to a virtual CPU and 1GiB of RAM while the maximum correspond to 36 virtual CPUs (see Intel Xeon Processor E5-2660 v3), 1 or multiple GPU(s) and 100 GiB of RAM.

Bibliography

[1] François Schnitzler. *Mixtures of Tree-Structured Probabilistic Graphical Models for Density Estimation in High Dimensional Spaces.* PhD thesis, University of Lige, Department of Electrical Engineering and Computer Science, 7 2012.

[2] P. Naïm, P.H. Wuillemin, P. Leray, O. Pourret, and A. Becker. *Réseaux bayésiens.* Algorithmes. Eyrolles, 2011. ISBN 9782212047233. URL http://books.google.fr/books?id=7d_Jq2ehb0oC.

[3] Marina Meila and Michael I. Jordan. Learning with mixtures of trees. *Journal of Machine Learning Research*, 1:1–48, 2000.

[4] Jr. Kruskal, Joseph B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):pp. 48–50. URL http://www.jstor.org/stable/2033241.

[5] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401, 1957. ISSN 1538-7305. doi: 10.1002/j.1538-7305.1957.tb01515.x. URL http://dx.doi.org/10.1002/j.1538-7305.1957.tb01515.x.

[6] Borvka Otakar. O jistm problmu minimlnm (about a certain minimal problem). page 3758, 1926.

[7] M. Hamstra, H. Karau, M. Zaharia, A. Konwinski, and P. Wendell. *Learning Spark: Lightning-Fast Big Data Analytics.* O'Reilly Media, Incorporated, 2015. URL http://books.google.be/books?id=9IPUlgEACAAJ.

[8] P. A. Humblet R. G. Gallager and P. M. Spira Prepared by: Guy Flysher & Amir Rubinshtein. A distributed algorithm for minimum-weight spanning trees. 16

(10), 2005. URL http://webcourse.cs.technion.ac.il/236357/Spring2005/ho/WCFiles/MST.pdf.

[9] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. 2011.

[10] Wehenkel Louis. *Thorie de l'information et du codage*. Centrale des cours de l'AEES, asbl, 2003.

[11] Richard E. Neapolitan. *Learning Bayesian Networks*. 2003.

[12] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. 2009.

[13] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1), January 1983.

[14] C. K. Chow and C. N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 14(3): 462–467, 1968.

[15] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. In *40th International Conference on Very Large Data Bases*. Stanford InfoLab, September 2014. URL http://ilpubs.stanford.edu:8090/1077/.

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.